FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR THESIS

Study program/Specialization: Bachelor of Science / Computer Science	Spring semester 2021 Open or Restricted access
Writer(s): Vegard Matre, Markus Aarekol Johannessen	
Academic Administrator: Tom Ryen Supervisor(s): Leander Jehl	
Thesis title: Vue vs. Vugu	
Credits: 20	
Keywords: Frameworks, Vue, Vugu, WebAssembly, Comparison	Number of Pages: 78 + appendix/else: Link to GitHub: Source Code Stavanger 15. May 2021

Contents

Abstract

Vue and Vugu are two frameworks used for developing web-oriented designs and applications, whereas Vue utilizes JavaScript, and Vugu utilizes Go and the fairly new technology WebAssembly. WebAssembly has emerged as a relevant technology to supplement JavaScript in developing web applications where JavaScript might be insufficient. For the application developed in this thesis, we wanted to investigate whether or not using WebAssembly would affect the performance of the application. Therefore we have implemented and evaluated a mirrored Sudoku application using the frameworks Vugu and Vue which are based upon these technologies. By creating the applications as equal as possible we have been able to compare the two frameworks against each other. This has allowed us to discover the advantages and disadvantages amongst them and helped us reach a conclusion. The objective was to investigate if Vugu, and hence WebAssembly would perform better than Vue under somewhat demanding circumstances such as in generating a Sudoku board with a corresponding unique solution. We found that the Vugu variant outperformed Vue in terms of performance in the developed Sudoku application. Although Vue has more resources and a broader community available, which in turn would make for a more efficient development life cycle, Vugu turned out to be the overall optimal choice for this application. Thus, confirming that WebAssembly, utilized by the Vugu variant, does have the capabilities that JavaScript is lacking.

Chapter 1

Introduction

Frameworks such as Vue and Vugu are software that is developed and used by developers to build applications. Frameworks provide the tools developers need to keep them from reinventing the wheel for every new project. They provide consistent starting points and organizational structure, which as applications grow larger, provides for better code performance.
Frameworks are software that is built, tested, and optimized by several experienced software engineers, programmers and companies. Therefore by utilizing them, one may rely on that the framework already has been tested and optimized for the various use cases that might appear while coding.
By building a new frontend web interface by only using HTML, CSS, and JavaScript without using a framework, finishing up a small project could be done relatively easily. But when a project grows larger and more complexity is introduced, one might find that following best practices and maintaining high code performance grows increasingly difficult. Especially when working in teams, maintaining an organizational structure becomes very important, as this helps for a wider understanding amongst the team when changes are to be introduced or bugs need to be tracked down.
The objective of this thesis is to compare the two frameworks Vue and Vugu, and to see if Vugu and WebAssembly can achieve better performance when used in web applications that require heavy computational operations. The comparison is done by building a Sudoku application and then implement it using the two different frameworks Vue and Vugu, followed by an evaluation of them. The application is built in Go and JavaScript, which the two frameworks are

built upon respectively, and then implemented with the corresponding framework. The performance of the application, as well as the development process, are then evaluated against each other so that a comparison between the two can be made. When creating a Sudoku application it is possible to opt for functions with time complexities that grow larger over time as the difficulty of the Sudoku board increases. As the applications are implemented with few differences, performance comparisons between Vue and Vugu can be derived from test results. Performance are an important part of the objective since applications with heavy computational operations, such as creating a Sudoku board, are one of the possible use-cases for WebAssembly and it should perform at least similar to established JavaScript frameworks such as Vue, to be a viable choice.

1.1 Outline

The rest of the thesis is outlined as follows:

Chapter 2 Introduces the reader to important terminology, frameworks, libraries, and algorithms used throughout the thesis.

Chapter 3 Describes how to setup and begin a project using either Vugu or Vue.

Chapter 4 Consists of the implementations of the application with both frameworks. The differences in the implementations are highlighted in this chapter.

Chapter 5 The in-depth comparison between Vugu and Vue is discussed. Performance tests of the two applications are introduced with coherent displayed results.

Chapter 6 The test capabilities in both Vugu and Vue are reviewed. Also, the support around the two frameworks are looked into and compared.

Chapter 7 Concludes the thesis and introduces thoughts regarding further work that could be done onto both the application and the Vugu framework.

Chapter 2

Terminology

This chapter introduces the important terminology, frameworks, libraries, and algorithms used throughout the thesis. The programming languages, JavaScript and Go, that Vue and Vugu are mainly built upon will be introduced and discussed. From there the frameworks will be presented. In the final part of the chapter, WebAssembly, which Vugu utilizes, and the algorithms used in the developed application will be introduced.

2.1 JavaScript

JavaScript (JS) is one of the most popular languages in the programming world. It is used as a scripting and as a programming language, and allows developers to implement complex features on web pages. Every time a web page does more than display static information, JavaScript is involved. It is a high-level programming language, which together with HTML and CSS form the three pillars of modern web development.

As a multi-paradigm language, JavaScript supports functional, event-driven, and imperative programming styles. It has application programming interfaces (APIs) for working, amongst others, with text, data structures, and the Document Object Model (DOM). When JavaScript code is placed within web pages,

i.e. in script tags, the browser's built-in interpreter locates the JavaScript code, reads through it, and executes it. Once the page is parsed and rendered, depending on the functionality defined by the implemented JavaScript, a user may now click on elements to invoke JavaScript functions to change the various views on the web page, and thus allowing for interactive web pages.

In 2021 JavaScript is arguably amongst the world's most broadly deployed programming languages. According to a Stack Overflow[1] survey in 2020, it is used by 58,3% of professional developers, making it a popular widely-used programming language.
According to the TIOBE index[2] which is an index that calculates the popularity and relevance of programming languages based upon search engine queries. The popularity of JavaScript over time, from the TIOBE index for JavaScript is shown in Figure 2.1 below, with its most recent peak achieved in 2018.

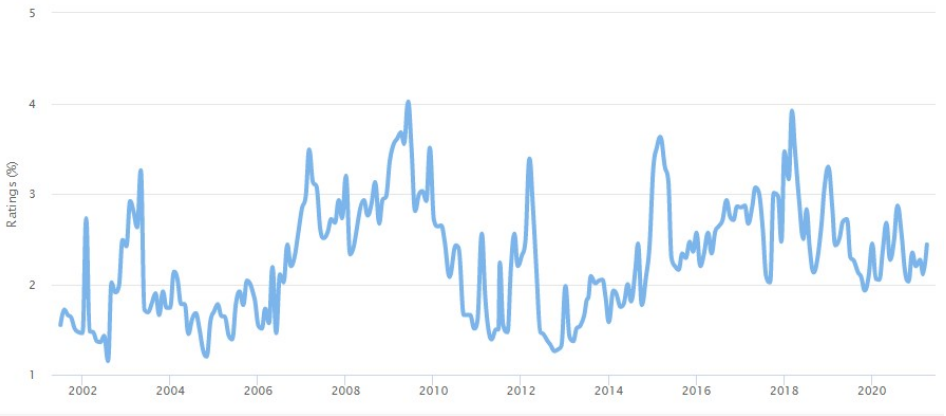


Figure 2.1: JavaScript popularity over time, higher % means more popular.

2.1.1 Node.js

Node.js is an open source JavaScript runtime environment that runs on Google Chromes V8 JavaScript engine, developed by Ryan Dahl in 2009[3]. Node allows JavaScript developers to run their scripts outside of web browsers, like server-side scripting, before the page is sent to the browser. In this thesis, Node.js was used under development of the JavaScript backend to test the code while building it, and later when executing various test cases to compare its performance versus the code written in Go.

2.2 Go

Go, also known as Golang, is an open source programming language, introduced by Google in 2009. It is a statically typed language and produces compiled machine code binaries. Go provides tools that allows for safely usage of memory, manage objects, collect garbage, and provide static typing along with concurrency. Go is a robust system level language used for programming across large scale network servers and big distributed systems. This has resulted in Go emerging as an alternative to C++ and Java for application developers. The purpose behind creating the language was to deal with the lack of pace and difficulties involved with programming for large and scalable servers and software systems.

According to the TIOBE index, Go was awarded the "Programming Language of the Year" in both 2009 and 2016[2]. This is an award for the programming language with highest the rise in ratings over the year[4]. According to the TIOBE index, the popularity of Go has been on the rice since. In the same Stack Overflow survey from 2020[1] mentioned above, Go scored a 62,3% and therefore surpassed JavaScript by nearly 4%. The popularity of Go over time is depicted in the TIOBE index for Go, in the Figure 2.2 below, showing its highest point in late 2016.

Figure 2.2: Golang popularity over time, higher % means more popular.

2.3 Vue

Vue is a modern open-source front-end JavaScript framework that provides useful facilities for building user interfaces and single-page applications, created by Evan You[5] in 2014. Unlike many other frameworks, developers can use Vue to enhance existing HTML, which enables users to use Vue as a drop-in replacement for libraries like JQuery. Vue may also be used to write entire Single Page Applications (SPAs) which allows the developer to create markup managed entirely by Vue. This may improve developer experience and performance when dealing with complex applications.

Like alternative frameworks, in Vue it is possible to create reusable blocks of markup via Vue components. Unlike popular frameworks such as React, Vue components are written using a special HTML template syntax. When more control than the HTML syntax allows is needed, Vue allows JSX or plain JavaScript functions to better define the components.

Typically when applications grows, a problem arises when the DOM has more nodes to traverse, which in turn leads to more elements and more scripts to communicate with, resulting in that the DOM grows slower and costs increasingly more processing power. Vue deals with this issue by utilizing complex algorithms to avoid re-rendering the entire DOM after any new change to the document. This is achieved by building a virtual DOM as an abstract of the original one. In turn this leads to less DOM API calls which greatly improves efficiency and resource management. To make use of Vue on existing sites, simply drop the Vue CDN into a <script> tag in the HTML. This will allow Vue on existing sites.

2.4 Vugu

Vugu is inspired by frameworks like Vue, React, etc. It is written in the programming language Go developed by Google. Vugu is more of a library than a framework according to the creators[6]. A framework could contain different libraries and the framework tells the developer where to put the code. A library is a set of tools the developers uses and decides where to use. In short a framework controls the flow of the application, while with a library the developer controls the flow. This is why Vugu is more of a library that can be utilized to create a web application. The developer mostly chooses where to use Vugu. Vugu enables developers to execute Go code in a web browser using WebAssembly (wasm). This is still an experimental technology and are in constant development. Vugu runs in most modern browsers and all the major browsers. Vugu has a Single Page Application (SPA) approach which means that the application can be split up in components. It is developed with the incentive of keeping the build environment simple and sane. Developers write user interfaces with HTML+CSS presentation and the facility of Go for interface logic. In Vugu there are .vugu files that are converted to .go filed. Then all .go files are compiled into a wasm module and executed in the web browser. The Vugu framework provides the functionality to efficiently synchronize HTML DOM on a web page based on the markup in the project files. DOM event handlers and breaking large parts into components are features supported by Vugu.
In short, Vugu allows for writing Go code to create web applications using WebAssembly.

2.5 WebAssembly

The development of the web platform has allowed for more advanced and demanding applications. This demands more efficient and secure code. JavaScript has shown not to be the best equipped language for this, due to i.e. the fact that it needs to be sent to the client side for execution. Because of this, engineers from the four major browser vendors worked together to design a portable low-level bytecode called WebAssembly[7]. WebAssembly does not commit to a single programming model. Compact representation, efficient validation and compilation, and safe to no-overhead execution is among the features that WebAssembly offers. It is an abstraction over modern hardware and thus making it hardware-, language- and platform-independent.

A WebAssembly binary file will run in every browser since it is the implementation in the browser that ensures the operating system specifics. Usually today JavaScript is delivered to the browser in text format. The files must then be analyzed and translated by the browser before it even runs the file. Therefore due to the analyzing and translation part of the JavaScript, WebAssembly might perform better than JavaScript for some cases.

As mentioned earlier one of the reasons behind WebAssembly is the more demanding applications. This could be games, visualization-, video- and audio software, augmented- and virtual reality. Another incentive to use WebAssembly is because it will be easier to express things like threads and SIMD. SIMD stands for Single Instruction, Multiple Data and is used to line up multiple chunks of data beside each other and run a single instruction to operate on every one of them at the same moment.

One important note is that WebAssembly is not to replace JavaScript, but can help fill the gaps where JavaScript is not optimal. JavaScript and WebAssembly should be used to fulfill each other. Where there are big chunks of data, WebAssembly is a potential candidate to use instead of JavaScript because of better performance.

2.6 Recursion

A function that calls itself directly or indirectly is the process known as recursion and the corresponding function is called a recursive function[8]. Recursion seems to create an infinite loop, but it is often used in such a way that this will not occur. This is accomplished by providing a condition that when met, it does not call itself anymore. The point of using a recursive function is to code some problems more easily and efficiently. A recursive function starts with having a base condition with a provided solution. The solutions to the bigger terms is expressed in terms of smaller problems. The bigger problem is then solved by solving these small problems until the big problem is solved.

Problems that may occur while using this method are i.e. stack overflow. Stack overflow is an error that occurs when a function uses more memory than available on the stack. This can happen if the base case is not reached, if it is not defined or if a condition to break the recursive function is not defined. Then it could cause a stack overflow error if the memory is exhausted by these functions on the stack.

Direct and indirect recursion are two types of recursion. Direct is when a function calls itself inside itself. It is called indirect recursion if a function calls a second function which then calls the first function again. By applying recursion to problems, solving the problems can be done more easily and efficiently than it without recursion.

2.6.1 Backtracking Recursive Algorithm

Backtracking can be defined as a general algorithmic method to solve a computational problem. This is done by testing all possible candidates step by step and if a candidate does not satisfy the constraints, it is discarded[9]. If a candidate is discarded the algorithm will return to the previous step (backtrack) and continue trying a different candidate. Figure 2.3 below shows a visual example of how this procedure works. This will continue until there are one or more solutions that satisfy the constraints.

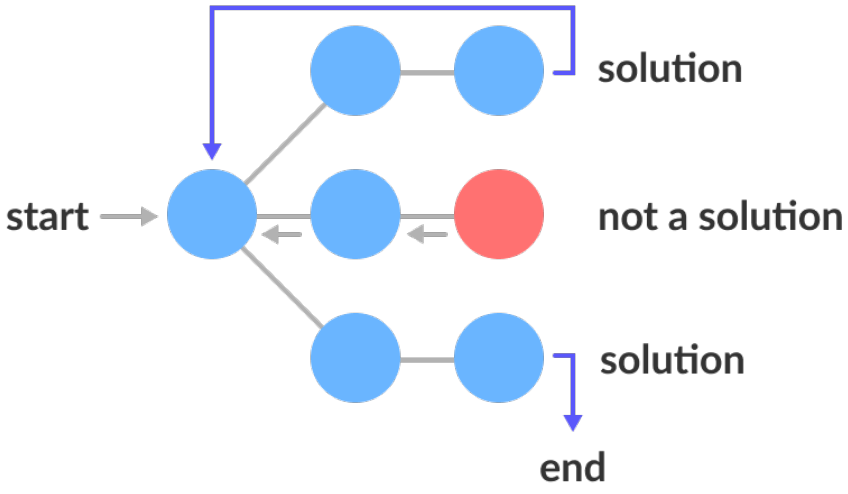


Figure 2.3: Concept of backtracking using a search tree.

Backtracking is used by having a set of partial candidates that in principle could have different solutions. The algorithm will try to solve the problem incrementally by testing all possible candidates to see if they satisfy the constraints. Conceptually, all partial candidates are represented as nodes in a search tree, the potential search tree. Each partial candidate is the parent to the candidate that differs from it by a single extension. The leaves of the tree are the partial candidates one cannot traverse further. The algorithm will traverse the search tree recursively, from root downwards, depth-first order. At each node, the algorithm will check if the node can become a valid solution, if not it will

skip the whole sub-tree (known as pruned). Otherwise, it will check the node itself to see if it is a valid solution, and if it is determined to be valid, it will recursively enumerate all sub-trees of the node. The two tests and the children of every node are specified by the user. This leads to that the actual search tree traversed by the algorithm is only a part of the potential search tree since the algorithm will skip the sub-trees that does not have a solution. The total cost of the algorithm is the number of nodes times the cost of obtaining and processing each node. This should be considered when choosing a potential search tree and implementing the pruning test.

In general, problems that are constraint satisfaction problems, have clear and well-defined constraints on any objective solution and can be solved by incrementally building candidates to the solution. If it is determined that the following can not act as a valid solution, the candidate is cancelled, and the algorithm backtracks to the previous step. Problems such as these can be solved using a Backtracking Recursive algorithm.

Other algorithms that can be used to solve similar problems are Dynamic Programming or Greedy Algorithms which are logarithmic, linear time complexity in order of input size, and hence will outperform the Backtracking Recursive algorithm since it is generally exponential in both time and space. But there still exist problems that only a Backtracking Recursive algorithm can solve. These are the problems where it is necessary to solve all sub-problems one-by-one, to find the correct solution.

2.7 Testing

In software development, testing is a method to verify that the software product provides the expected requirements and to ensure that the software product is without defects and bugs. There are two main reasons to use testing, to check the level of quality and acceptability and to identify problems with the software product[10]. Testing is an important part of the development cycle. Also, testing can be used for Test Driven Development (TDD). There exist different forms of testing for different reasons and it depends on what is being tested.

2.7.1 Unit Test

A Unit test is a form for testing the functionality of the written code. Unit tests are tests that test a unit component. Unit components can be functions, objects, or anything else that an end-user might depend on. A unit test tests the integrity of the unit components. Writing unit tests allows the developers to verify that the code meets the requirements and quality standards. By adding unit tests into the development from the early stages it could save the developers time and money, and it will also help the developers write better and more efficient code.

2.7.2 Performance Test

Performance tests are used to test the performance of the various features of an application. To test the features of the application mentioned in this thesis, the performance developer test tool built into every browser is utilized. The performance tool can be found in most modern browsers, such as i.e. Mozilla Firefox[11]. This tool gives insight to an application's general responsiveness, JavaScript and layout performance. Further down in this thesis, the performance tool will be used to measure the duration of DOM events triggered by events from the Vugu and Vue application. The results from the tool will then be used to give comparisons between the Vugu and Vue applications based upon the different features from the DOM events.

Chapter 3

Introduction to Vugu and Vue

This chapter shows how to setup and start a project with Vugu and Vue. The chapter also shows differences, advantages and disadvantages with using Vugu vs Vue. Relevant functionality is also discussed in this chapter. Because Vue is a much more known framework than Vugu, which still is in a experimental state, this chapter is much more focused on the Vugu part of getting started with a project, partly due to the fact that Vugu needs a more hands on approach of setting up project dependencies.

3.1 Getting started with Vugu

Getting started with a Vugu application is simple. First, install the vgrun tool from GitHub and use this tool to install all other tools that are designed to help creating a web application with Vugu. These tools are not necessary for Vugu programs to run or compile since Vugu is just Go. The vgrun tool is a wrapper that provides features like file watching. This will automatically build and page refresh when editing the application under development. Downloading the example project from GitHub is an easy way to start a project. This is a web application that can be compiled and executed in the browser without further work and from that point, the development of an application can begin.

```
C:\Users\Vegard\OneDrive\Utdanning\DATBAC\VuguVsVue\Vugu>go run devserver.go
2021/02/22 10:27:23 Starting HTTP Server at "127.0.0.1:8844"
WasmCompiler: Successful generate
WasmCompiler: Successful build
```

Figure 3.1: Running Vugu application at 127.0.0.1:8844 using go command.

```
C:\Users\Vegard\OneDrive\Utdanning\DATBAC\VuguVsVue\Vugu>vgrun -1 devserver.go
2021/02/22 10:35:11 Starting HTTP Server at "127.0.0.1:8844"
WasmCompiler: Successful generate
WasmCompiler: Successful build
```

Figure 3.2: Running Vugu application at 127.0.0.1:8844 using vgrun command.

Figures 3.1 and 3.2 show how to start the Vugu application using both the vgrun tool and simple Go commands. Default settings are using 127.0.0.1:8844 as the address. In the example project there are known files such as .gitignore, README.md, and go.mod. There is also a generate.go file that has a Go generate comment. This file are shown in Figure 3.3

```
1  package main
2
   run go generate | run go generate ./...
3  //go:generate vugugen -s
4
```

Figure 3.3: Comment that invokes the Vugu generator.

Another file is the root.vugu. In the example project, it is a simple vugu component to show how Go code and HTML elements can be combined. The component named root is by default the top-level component and is rendered inside the <body> tag. The devserver.go file is a simple development web server. This server serves the program to the server and it does not get compiled to WebAssembly. The file starts with an HTTP server and a WebAssembly compiler. On the Vugu homepage, there is also a playground where it is possible to learn and experiment before starting a project in Vugu.

3.2 Getting started with Vue

Due to Vue being a fairly known and established framework, setting up a starter Vue project has become a very simple procedure. All that is needed to set up a sandbox example is one .HTML file to hold both the Vue instance, and the Vue CDN within a script tag, one .CSS file, and one .JS file. To get started with a project, one may also make use of the Vue CLI which simply installs all of the dependencies needed for the specific project. At project end, any Vue project initiated with CLI can also easily be built for production with the npm run build command. The CLI then builds the project for distribution within a dist folder.

3.2.1 The Vue Command Line Interface

For building more complex applications, it is recommended to use the Vue NPM package. This allows for more advanced Vue features. To make the building of applications easier, there is a Command Line Interface to streamline the development process. The CLI ensures that the various build tools works together smoothly, to ensure that developers can focus on writing their app instead of having to invest time into various documentations when dealing with configurations.

Upon project creation the CLI offers various plugins that are npm packages which provide optional features to your project. One useful plugin are Eslint integration, which is a static code analysis tool, used for identifying problematic patterns found in JavaScript code. Another useful plugin that simply can be initialized with the CLI, used in this project, is the Jest testing framework. The Jest framework can be used to test both the JavaScript logic and the Vue components.

3.3 State management

As applications grow, being able to control the state of an application and handle messages, like i.e. the degree of difficulty set for a Sudoku board by a user becomes important. State management is a design pattern that both Vugu and Vue offers support for that can be used to handle these scenarios.

3.3.1 Vugu Wiring

Go has a code generating tool called Wire. This tool automates connecting components, often structs in Go, using Dependency Injection. Dependency Injection is a coding technique were functions and structs that developers depend on are abstractions[12]. This is an important design principle in programming as it keeps the code loose-coupled and easy to maintain. In Wire, dependencies between components are represented as function parameters. This is encouraging explicit initialization instead of global variables. The benefits of using Wire is that the container code is obvious and readable since the code is generated. Also, it is easy to debug since if there is a dependency that is missing or not being used, an error will occur during compilation. Since Vugu programs involve UI components that are created and destroyed in response to user interactions throughout the lifespan of the application, Vugu applications would not benefit from using wiring tools such as Wire, unlike many other Go programs. Hence, wiring applications at runtime needs an effective mechanism. This must ideally be with as much help and type-safety as possible from the Go compiler. In a Vugu application, wiring is done by providing a wire function that is called every time a component is created or re-used. This function is called often and should be kept clean and simple.

Vugu wiring allows for creating shared structs that can be injected into any UI component that needs a reference to it. In a new Go file the desired struct with data fields and functions are created. In this file, there also is a struct that points to the desired struct.

```go
package main

type Difficulty struct {
    Difficulty int
}

func (d *Difficulty) SetDifficulty(dc int) {
    d.Difficulty = dc
}

func (d *Difficulty) GetDifficulty() int {
    return d.Difficulty
}

type (
    DifficultyRef    struct{ *Difficulty }
    DifficultySetter interface{ DifficultySet(d *Difficulty) }
)

func (dr *DifficultyRef) DifficultySet(d *Difficulty) { dr.Difficulty = d }
```

Figure 3.4: Difficulty.go file used in Vugu wiring.

This works by having a difficulty.go file as seen in Figure 3.4. In this file, there is a Difficulty struct with a Difficulty integer data field. This integer represents the number of cells that are to be removed from a filled Sudoku board. Functions that belong to the Difficulty struct are also located inside this file. Vugu wiring also requires two tools. The first tool is a struct that is a reference to the Difficulty struct, called DifficultyRef. The second tool is an interface called DifficultySetter, which sets the Difficulty struct. In the setup.go file the SetWireFunc call is providing a function to wire up each component as it is created. Calling this function is handled automatically by Vugu for each component that is instanciated or re-used. Inside the Choosedifficulty.vugu file the Choosedifficulty struct embeds the DifficultyRef which is a pointer to a Difficulty struct. Then it implements the CounterSetter interface. This interface is checked for in the vuguSetup function inside of the setup.go file.

In the implemented Vugu application there is a component that allows the user to decide the difficulty. The difficulty value is then used in a different component. With Vugu wiring it is possible to create a shared object that makes it possible to get the difficulty value from another component. In this example, the Difficulty struct is used into two different UI components allowing two smaller and more focused components instead of a big component.

3.3.2 Vuex

Vuex is an official plugin for Vue.js, which can be installed as a feature when starting a Vue application using the Vue CLI. Vuex offers a centralized datastore within a Vue application that holds the shared application state. As applications grow larger and the child component of a component tree structure changes its state, by i.e. a button click to query a Sudoku board with an increased difficulty, instead of having to pass on the changed state chronologically upwards through all parent components, that new state is altered with mutations within the shared application state.

Similar to the Vugu application, the Vue application also has a component that allows the user to decide the difficulty of the board. Vuex state management makes it possible to share the difficulty, the new state, selected by the user in the UI to all the other components.

3.4 Routing

Routing is a common feature used in web applications to map the UI tree nodes to the URL paths and vice versa. Each branch is assigned a URL path, made reachable through the root at "./", i.e. reaching the rules component at; "./sudokuRules". Routing contributes to reducing accidental complexity that may occur when adding new routes to an application, by utilizing routing an intuitive pattern to solve complexity problems can be used. Routing also contributes to giving more structure to web applications, making it easier to understand, debug and potentially later, extend.

3.4.1 Vugu Router

One of the most common usages of a JavaScript framework such as Vue, is to create a Single Page Application. A central feature required to do this effectively is a router. Vugu is no different and has its own version of the Vue router, which is a package from github.com/vugu/vgrouter. This package provides functionality for the application to handle routing for server- and client-side, reading HTTP requests, and reading browser path.

The routes in Vugu are registered and when they are processed they are examined in the sequence they were registered. Routes can be either partial or exact matches and the code for that route will perform the desired action. Having one or more sections of the user interface being dynamic based on which route is the usual approach. The route handle, or handlers, that match for a given route assign those fields of type vugu.Builder and is then rendered via a vg-comp tag.

A Vugu router is implemented by using a wire function as seen in Section 3.3.1. First, an instance of a router is made and then the wire function will populate everything that wants a "Navigator" or router. This is done in the setup.go file and inside the vuguSetup function. The routes are also implemented inside this function. A root component is also created from the Root struct in root.vugu. In the root component, there is a data field called Body. When navigating to a route the root.Body is set to the URLs component. This could be a component for an entirely new page. This will then be rendered via the vg-comp tag which takes in the Body data field of type vugu.Builder. In practice, this works by having each route have a corresponding vugu component file. This component will be set to the Body in the root component and then shown on the page.

3.4.2 Vue Router

Similar to Vugu, Vue also has a router dedicated to navigating between the views of an application. The Vue router uses client-side routing, meaning that the routing happens in the browser itself using JavaScript. The application's web pages get loaded into a single HTML page so that the application does not need to reach out and request continuous responses from a server, called server-side routing. The server-side routing takes place when the user clicks a router link which then dynamically presents the requested view, as it has already been loaded into the browser. This keeps the browser from refreshing every time a new page is requested which in turn results in greater performance and user experience. This routing in between different views of an application could i.e. be when a user wants to check out the rules of a Sudoku game, then the user can simply click the «rules» link and the rules view with its corresponding components will be dynamically presented to the UI.

3.5 Components

A smart way to divide and organize Single Page Application (SPAs) while using frameworks is to make use of components. This works by splitting up the application into different parts where each part of the application can be represented by a component. This could i.e. mean splitting up the application into components containing a header, navigation bar, content bar, or any other feature. The fact that it is possible to reuse a component as many times as necessary allows developers to build applications organized as a tree of nested components. By using components it is easier to update and troubleshoot the application. Both Vue and Vugu support the use of components in a similar way. Figure 3.5 depicts how this could be organized in a Sudoku application.

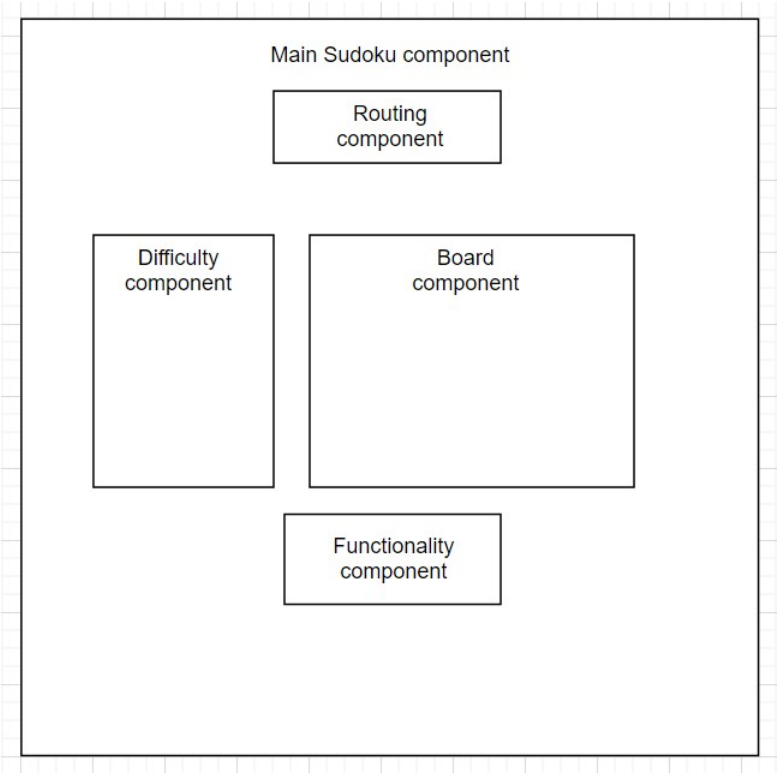


Figure 3.5: Visualization of the components architecture in the Sudoku application.

3.5.1 Vugu Component

Components in Vugu are individual .vugu files which are used to organize the user interface code. Each of these .vugu files is processed to produce a .go file, this .go file is a code generated file that ends with _vgen.go. All components are in fact just Go structs. The top-level component is the root component and it is this component that is rendered inside of the <body> tag. From this component, it is easy to add new components.

```
<div class="board">
    <main:Board></main:Board>
</div>
```

Figure 3.6: Using the Vugu component Board.

Figure 3.6 shows how to use a component. This is a Board component that are inside a tag. The component is in its own .vugu file and has a Go struct called Board which is inside a <script> tag. It is also possible to add functions belonging to this component in this tag. By using components the project gets easier to maintain and organize.

3.5.2 Vue Component

Vue component is one of the most useful features of Vue. Components help developers extend basic HTML elements to encapsulate reusable code. Each component can have its own state, markup and style. Components are stored in a .vue file. The file starts with a <template> written in HTML, serving as a directive to the framework on how to produce the final markup of the component based on its internal state, shown below in Figure 3.7.

```
<template>
    <div class="sudoku">
        <div class="row">
            <h2>Sudoku</h2>
```

Figure 3.7: Vue HTML holder.

Secondly the Vue component takes a <script> tag to hold the components JavaScript logic. Within this section, low level concepts such as properties and state reside. Properties are a set of variables to configure each component's behavior, provided by host application or parent component. The state is a data structure that provides the state of a component at a given time, often changed based on occurring DOM events. The backend Sudoku class is imported and made usable within the various .vue components here. This is shown in Figure 3.8.

```
<script>
import { sudoku } from '/src/sudoku.js';

export default {
    name: "Sudoku",
```

Figure 3.8: Vue JavaScript logic holder.

Lastly, the vue component takes a <style> tag for which it's CSS is placed. The logic for design features and looks is placed here. When the scoped keyword is placed within the style tag, the CSS is limited for the respective component only, shown below in Figure 3.9.

```
<style scoped>
.sudoku {
    width: 100%; max-width: 420px;
    margin: auto;

    font-family: Arial,sans-serif;
}
```

Figure 3.9: Vue CSS holder.

3.6 Summary

In short, since Vue is more established compared to the experimental framework that Vugu still is, Vue is faster to get started with. However, Vugu and Vue have many similarities in terms of application development. Both have a router, both have state management, and both make use of components. Hence if a developer is familiar with Vue, Vugu will be somewhat intuitive to learn. Vue has a CLI which can be used as an easy way to install all dependencies for a project. Unfortunately Vugu does not have an alternative to this for the moment, therefore setting up a Vugu project from scratch requires more steps. Because i.e. the Vue Command Line Interface, Vue is easier to start a project with compared to Vugu, but Vugu have many similarities that allows for an easier transition from Vue to Vugu.

Chapter 4

Implementation

This chapter shows how the Sudoku application works and how it is implemented. During the implementation of the application in this thesis, the similarity of the two implementations was emphasized so that the comparison of the two frameworks and languages were not to be influenced by different styles, techniques, or differences related to chosen data structures and algorithms. Therefore the Go backend with its corresponding Vugu frontend was implemented first. After implementing and testing the functionality of the various functions in Go and Vugu, and ensured satisfactory functionality, the JavaScript backend with corresponding Vue frontend was implemented as similar to the Vugu implementation as possible. There were of course some differences bound to happen, due to amongst other things, the fact that Go is a typed language, while JavaScript is an untyped language.

4.1 Sudoku

Sudoku is a game that most of the time resolves around a playing board that consists of size 9x9 cells. A 9x9 sized board will consist of nine 3x3 boxes that each holds 9 cells. Figure 4.1 shows an example board from the application with its corresponding unique solution. The objective of Sudoku is to fill the board with numbers ranging from 1-9. At the beginning of the game, the board is only partially filled. The difficulty of the given board depends on the amount of given numbered cells. To win the game, one has to successfully fill up the rest of the board without violating the following set of rules:

- Each column can not have duplicate numbers
- Each row can not have duplicate numbers
- Each box can not have duplicate numbers

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4.1: A Sudoku board with its corresponding unique solution.

4.2 Implementation of the Sudoku application

The idea for implementing the application with both Vugu and Vue is to try and keep it as similar as possible. This is to ensure the most valid result for the performance comparison of Vugu and Vue for this application. Because Vue is a much bigger framework than Vugu there exist more features in Vue. Therefore, the application was first made with Vugu and Go and then made with Vue and JavaScript. Not much focus was aimed towards creating optimized functions. Hence, some of the functions tend to use some time finishing. This was the chosen approach since this was more likely to lead to differences in the comparison of the Vugu and Vue implementation.

4.2.1 Part I Creating the filled board

As follows from the Sudoku games objective, the first part of the implementation is to create a board with filled values without violating any of the 3 rules mentioned above for Sudoku. There are different ways to accomplish this. One is to go through every cell in a brute force backtracking manner and fill it with a random number and check if the imprinted value is allowed. If the imprinted value breaks any of the 3 rules, then it gets removed and a new value is checked. A similar improved approach, used in this implementation, is to first fill all the cells in the Sudoku board along with one of the diagonals, i.e. left to right as shown below in Figure 4.2.

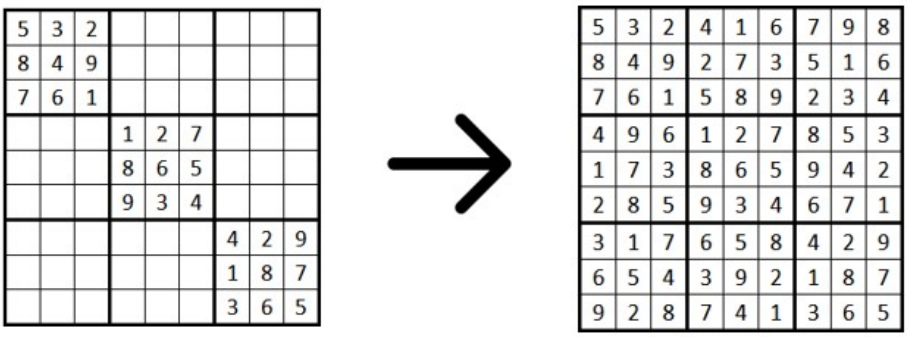


Figure 4.2: From diagonals to filled board.

This is a better approach because in doing so, the algorithm can legally skip checking if the imprinted value already exists in the row or column. It only needs to check if the new value to be inserted already exists within the actual 3x3 box. An empty 3x3 box can be seen in Figure 4.3.

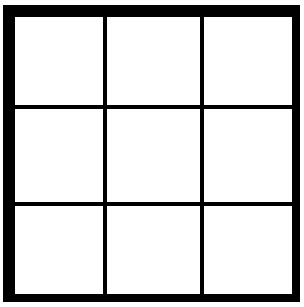


Figure 4.3: One of nine 3x3 boxes on a Sudoku board.

Then the rest of the board is filled using a recursive function. The recursive function then checks for every consecutive cell if the next imprinted value is valid for its row, its column, and its 3x3 box, before calling itself and moving into the next cell in order. As a result of this implementation logic, we achieve a filled board that satisfies the 3 rules for Sudoku, with each cell containing an integer ranging from 1 through 9.

4.2.2 Part II Removing values to create a starting point

After a correct board has been created, as the one from Figure 4.2 above, the next objective for the game is to create an unfinished Sudoku board from the solution presented in part I. This is where things get a bit complicated. In this part, values must carefully be removed one by one from the filled board. It is important that the cells that are to have their value removed are carefully selected, to ensure that the board still maintains its unique solution. This is because a Sudoku board with more than one unique solution is not considered a "real" Sudoku board.

The implementation logic manages this through a function called CreateUniqueBoard(k) which takes in the parameter k, which represents the number of values to be removed from the filled board in part I. Within the CreateUniqueBoard(k) function, a new function, Counter(), that utilizes a Backtracking Recursive algorithm, is called for each iteration where the Counter() counts the number of solutions after every removed value. If the total number of solutions surpasses 1, thus breaking the constraint of not having more than one solution, the removed value is placed back into original cell, and the algorithm moves to try a new random cell. This logic continues until the desired number of values given in the parameter k in CreateUniqueBoard(k) are cleared from the filled board. The higher the number parameter k is set to, the longer processing time it will take for the implementation to find a board that satisfies the rules while maintaining the unique solution.

The least amount of given cells a Sudoku board may have is 17[13]. Hence k can never be higher than 64. If k is assigned a higher number than 64, then it becomes impossible for the Sudoku board to maintain a unique solution. It is safe to say that a Sudoku board with the max difficulty, only 17 given cells, would be hard to solve without the help of a machine. The Figure 4.4 gives an example of what a user might see when choosing a board with the "Medium" difficulty. After the user clicks the button that sets the difficulty to "Medium", the function CreateUniqueBoard(k=48) would be called, and the user would be presented with the board below in the user interface.

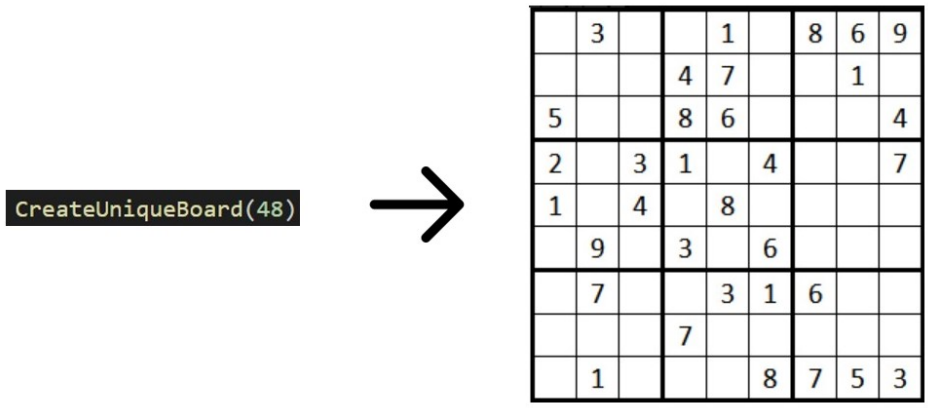


Figure 4.4: A Sudoku board generated with "Medium" difficulty.

4.2.3 Part III Solving the unique board

After the unique board, mentioned in part II, has been created the objective now is to make it back to the filled board from part I for the players to win the game. For this purpose, the implementation logic has another function that utilizes a Backtracking Recursive algorithm, called SolveBoard(), which keeps track of the next cell with an empty value. When the function finds an empty cell, it attempts to assign it a value before checking if the now assigned value is applicable without violating the rules. If the assigned value still follows the rules, then the implementation logic continues to the next cell in order. If however, the implementation logic discovers that it is not able to assign any correct values to the cell next in order, it backtracks to fix misplaced values before it. After the SolveBoard() function is complete, the result should be the same as the filled board from part I, which is the unique solution. The Figure 4.5 below shows how a generated Sudoku board from part II finds it unique solution when a user clicks the user interface to solve the board, invoking the SolveBoard() function.

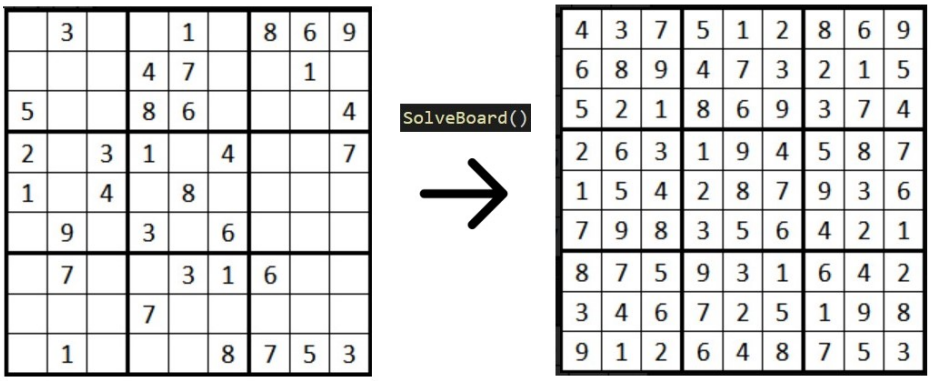


Figure 4.5: The Sudoku board with its unique solution.

4.3 Differences in implementing the Sudoku application

When implementing the Sudoku application in Vue and Vugu there were bound to be differences in the implementation since the two frameworks are based upon two different programming languages. Regardless of the differences, the same approach as described in Section 4.2 is used for both implementations. The differences in the implementations do not affect how the functions work, but is only present due to the differences in syntax between Go and JavaScript. Similarity in the implementations will allow for comparing the performances of the applications without having to give second thoughts to them having differences in the implementations. Hence in this section, the differences will be highlighted and explained.

Both the Vue and the Vugu application have a separate file for the functions that create the Sudoku board. In Vugu it is a .go file with a Sudoku struct and functions belonging to that struct. It also has helper functions for Vugu wiring. In Vue, a similar approach approach is used with a .js file, but this file is containing a class called Board instead of a Sudoku struct. This is because Go does not have classes compared to JavaScript. The JavaScript class has a constructor that has the data fields Board and Count. The Board data field is an array of arrays. The Count data field is an integer with a start value of 0. In the Go implementation, only the data field Board of type slice of slices is used. A slice in Go is the same as an array in JavaScript. The Count in Go is a global variable instead of a data field. This is because in Vue only the Board class is imported from the .js file. In Vugu the whole file exists in the same package and it is not necessary to import.

From this point all the functions used for creating a Sudoku board are implemented in a similar form, only syntax differences occur. All the important functions and how they work are explained in Section 4.2. In Vugu the Sudoku struct is initialized in the setup.go file instead of in the Board component for JavaScript. This is because of Vugu wiring. This is also the case for setting the difficulty using the struct ChooseDifficulty.

In Vue this could be achieved by using Vuex, but it is not implemented in this application since it does not affect the performance of the application which was the main goal. It was used in Vugu to see if Vugu had a similar feature as Vuex. Using Vugu wiring only changes where the Sudoku struct is initialized and does not affect the overall application, it only allows for a shared object and allowing all components to use this struct.

The only major differences are the uses of classes in JavaScript and structs in Go which does not affect the usability of the application. The other differences are where the Sudoku struct is initialized in Vugu compared to Vue. Since the functions used to create and solve a Sudoku board are implemented almost equally, the performance should not be affected either. This leads to the conclusion that the differences in these implementations highlighted in this section does not affect the use of the application or the performance.

4.4 Implementing the app in Vugu and Vue

To create the Sudoku applications, a backend in Go and JavaScript had to be created. The two backends hold the same logic and functions, but with some differences due to the differences between the two programming languages. For the Vugu application, its backend is located in its file named Sudoku.go, and the Sudoku struct and functions from within this file are reachable for every .vugu file as long as they all exist within the main package. Upon rendering the application page, Vugu converts all the .vugu and .go files within the main package to generate a main_wasm.go file. For the Vue application, it's backend is located in its own file named Board.js. The Board class and functions from this backend, however, are exported from the Board.js file and then imported from within the various .vue components that use functionality from this backend.

The two applications are made with the same components, they both have a main component that holds the main logic for the application, a component that holds the matrice for the Sudoku board itself, and a rules component briefly explaining the rules of the game. All of the components are placed within a components folder and are made reachable through routing. The Vue application utilizes the Vue framework's createRouter and createWebHistory modules from the Vue-router library to handle the routing. The applications' Home and Rules view are routed with router-links by, and the components are attached to their corresponding intuitive view.

Routing is also accomplished in the Vugu application. The Vugu application has as mentioned above the same components, but also an additional Root component, which is the top-level component that houses everything else. This component has a Root struct type, which defines a Builder and a Navigator. The Navigator allows users to navigate between the different pages using routing, and the Builder sets the appropriate HTML-code, when the Navigator passes the selected page, via "vg-comp" into the Body data field, which then renders the correct page.

4.5 Summary

Both Vue and Vugu utilizes the concept of organizing code into reusable components. These are fairly simple to use and do not require a broad understanding of the two frameworks to get started. Therefore the conversion from Vugu to Vue was somewhat intuitive. Both the frameworks use routing to navigate between the application's pages/views. The routing is also done very quickly as both of the frameworks can interact with a virtual DOM without having to touch the real DOM or go through the DOM API. One could argue that the setup for routers in Vue is somewhat more clear because the Vugu router requires the additional Root component with its struct containing the Builder and Navigator. However, after processing some of the Vugu routing documentation and examples it proved not to be as complicated as anticipated. For the respective programming languages, there are also some minor differences, these being the usage of arrays in JavaScript and slices for Go, untyped syntax in JavaScript and typed in Go, and the general buildup for functions in Go as the language frequently uses pointers, and JavaScript does not.

Chapter 5

Testing and Comparison of the Results

In this chapter, some in-depth comparison between Vugu and Vue is discussed. The performance comparison between the backend code in Go and JavaScript will be inspected. The comparison between Vue and Vugu is presented by performance testing the two applications with tools available in the browser. Furthermore, support, documentation, and community available for the frameworks are discussed. In the final part of the chapter, the overall test environment for Vugu and Vue are examined and compared with a short conclusion of the chapter. A complete Sudoku board has 9x9 cells, all of them assigned a value ranging from 1 through 9, for the performance testing the degrees of difficulty are defined as follows:

- Easy: Removes 40 values from the board.
- Medium: Removes 48 values from the board.
- Hard: Removes 52 values from the board.
- Extra Hard*: Removes 55 values from the board.
- Limit*: Removes 56 values from the board.

* Means that these difficulties were only used for the tests in Section 5.1.

5.1 Transitioning from Go and JavaScript to Vugu and Vue

To investigate if there are any major differences in performance when exporting the backend code in Go and JavaScript to the browser, some results gathered from local test runs were assembled. These tests were executed accordingly to the difficulty specifications for "Easy", "Medium", "Hard", "Extra Hard", and "Limit" mentioned above. Node.js, introduced in Section 2.1.1, was utilized for executing the JavaScript code in the terminal, while the Go code simply could be executed by the command go run <scriptname>. The results from these tests are seen in Table 5.1 and 5.2, and in Figure 5.1 below.

Result No.	1	2	3	4	5	6	7	8	9	10	Average (ms)
Easy	4.0	3.1	3.5	4.0	3.3	2.4	4.9	3.5	4.8	3.0	3.7
Medium	19	35	19	14	8	6	15	15	8	23	16.2
Hard	82	148	60	48	254	124	173	211	190	144	143.4
Extra Hard	236	63	1465	377	564	50	84	305	1420	876	544
Limit	378	26602	4262	2838	819	2565	175	5332	1588	937	4549.6

Table 5.1: Results for creating a Sudoku board in Go.

Result No.	1	2	3	4	5	6	7	8	9	10	Average (ms)
Easy	62	98	73	70	58	65	63	86	87	58	72
Medium	112	1024	194	143	126	143	475	575	99	230	312.1
Hard	171	515	2996	5143	963	1277	5367	328	1570	237	1856.7
Extra Hard	8020	22093	15350	9535	871	9739	7486	601	10700	36094	12048.9
Limit	41906	12812	149161	2199	24555	552	11771	10150	16556	6651	27631.3

Table 5.2: Results for creating a Sudoku board in JavaScript using Node.js.

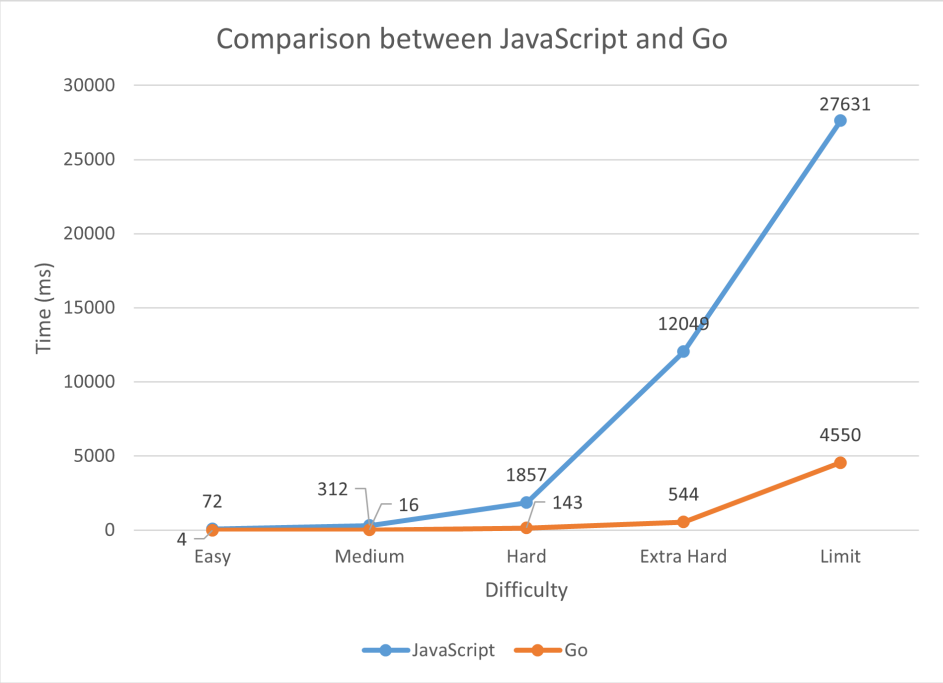


Figure 5.1: Comparison between JavaScript and Go when creating a Sudoku board.

Looking at the results from the backend JavaScript and Go code after performing 10 test runs gave the following results seen in Table 5.1 and 5.2. Some extra degrees of difficulties above the ones used in the application were also tested as an attempt to limit test and find the implementations outer borders. In the tables above, these extra difficulties are "Extra Hard" and "Limit", meaning that for these difficulties 55 and 56 values were removed from the Sudoku board.

In Figure 5.1 the average time duration for creating a Sudoku board is compared against each other. From this graph it is clear that Go has a big advantage over JavaScript which is expected since Go is a faster programming language. This is mostly due to the fact that Go compiles to machine code, and does not have to be interpreted.

As mentioned earlier the minimum given amount of values for a Sudoku board to maintain its unique solution is 17, meaning that the input parameter k from the function CreateUniqueBoard(k) mentioned in Section 4.2.2 is not allowed to be any higher than 64. As the input parameter k is allowed to grow towards this limit of 64, more randomness will occur in the implementation and this is reflected in the high fluctuations that can be observed in the results. This randomness in the implementation grows coherently with the number of cells removed because this leaves for more unsigned cells that the Backtracking Recursive algorithm needs to iterate through.

5.2 Performance test of the Vugu- and Vue application

The performance test in this sub-chapter is performed by utilizing the performance tool located in every browser. To gather results from a test run of the application, it is possible to start a recording that will present a summary of the various actions the web browser had to perform to be able to present the web application to the user. Within the browser performance tool, there are also four sub-tools available for users to take a closer inspection of the application. One of the four tools is called "Waterfall", which is a tool that shows a summary of all the operations that the browser had to execute in order to display the application. In Figure 5.2 is an example of how this sub-tool works, with the results of the DOM event where a Sudoku board is created with the difficulty "Hard" in the Vugu application. These operations could amongst others be internal JavaScript calls and garbage collection performed by the browser.

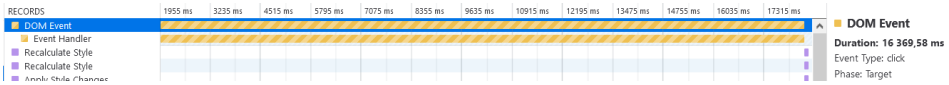
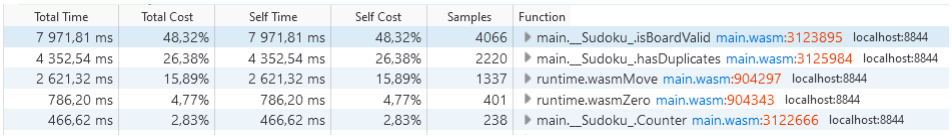


Figure 5.2: Waterfall example in the Performance tool.

Another sub-tool is the "Call-Tree". This sub-tool displays a summary of which of the internal functions were called upon the most and occupied the runtime of the specific test recording. In Figure 5.3 is an example of the sub-tool from a sample recording of the Vugu application. The figure displays a summary of the most time-consuming functions when creating a Sudoku board with the difficulty "Hard" in the application.



Total Time	Total Cost	Self Time	Self Cost	Samples	Function
7 971,81 ms	48,32%	7 971,81 ms	48,32%	4066	main.__Sudoku_.isBoardValid main.wasm:3123895 localhost:8844
4 352,54 ms	26,38%	4 352,54 ms	26,38%	2220	main.__Sudoku_.hasDuplicates main.wasm:3125984 localhost:8844
2 621,32 ms	15,89%	2 621,32 ms	15,89%	1337	runtime.wasmMove main.wasm:904297 localhost:8844
786,20 ms	4,77%	786,20 ms	4,77%	401	runtime.wasmZero main.wasm:904343 localhost:8844
466,62 ms	2,83%	466,62 ms	2,83%	238	main.__Sudoku_.Counter main.wasm:3122666 localhost:8844

Figure 5.3: Example of the Call-Tree sub-tool showing the most time consuming functions.

The third sub-tool “Allocations”, gives a summary of the heap allocations made by the developed code over the recording time interval. The last sub-tool “Flame Chart”, gives a summary of the browser's JavaScript call stack over the recording time interval. Those two sub-tools will not be mentioned further on in the the performance testing subsections as these did not generate any significant nor interesting results.

For every test run of the two applications, a recording was made that started when triggered by a user clicking one of the difficulty buttons and ends when the application displayed the queried board matching the difficulty. This would create a DOM event, as the one shown with the "Waterfall" sub-tool in Figure 5.2 above. From that recording, it is possible to inspect the results to look for differences in performance. One way to compare the Vue and the Vugu application against each other is to look at the time difference when requesting boards with the greatest degree of difficulty, "Hard", giving results similar to what can be seen in Figure 5.3.

5.2 Performance test of the Vugu- and Vue application

When performing these test recordings there are different variables that can affect the results, especially the tests revolving around using the "Hard" difficulty, which generates the most open and unfilled board, and also subsequently introducing more randomness. One of the reasons why this may affect the results is due to the Backtracking Recursive algorithm that is used as the implementation logic in the application. This implementation increases exponentially in time, and could in theory alongside randomness cause the application to run for a longer time. The randomness is hard to remove because of the way that the Sudoku application is implemented.

When the application is solving the Sudoku board both the Vugu- and Vue applications are using a brute force technique. This will try all possible values until it finds the correct solution, which there can only be one of since the board has only one correct unique solution. This will in theory be faster for a software program since this brute force technique does not depend on randomness and the unlimited number of attempts such as the Backtracking Recursive algorithm used when creating a Sudoku board. There exists only a limited number of possibilities when solving a Sudoku board and the number of possibilities depends on how many clues are left on the Sudoku board and how those are arranged. The brute force approach will be finished fairly quickly because the only thing the function needs to do is to loop over all the possible solutions in each cell until it is solved.

5.3 Performance test when creating a Sudoku board

In this section, an overview of the results after using the performance tool in Mozilla Firefox on the Vugu and Vue application can be found. The results from switching between the different degrees of difficulty in the application are represented below. The results found in Table 5.3 and 5.4 are from using the "Waterfall" sub-tool and measuring the time duration used to create a Sudoku board. An example of the "Waterfall" sub-tool is shown in Figure 5.2. The results are from measuring the application 10 times for each difficulty. The average is used in the further comparison.

Result No.	1	2	3	4	5	6	7	8	9	10	Average (ms)
Easy	160	231	220	127	253	150	149	116	130	107	164
Medium	394	145	235	491	414	276	270	158	650	169	320
Hard	3946	14801	4177	2473	9353	7548	6526	3522	5158	25659	8316

Table 5.3: Results for creating a Sudoku board in Vugu with Performance tool using Mozilla Firefox.

Results No.	1	2	3	4	5	6	7	8	9	10	Average (ms)
Easy	1700	1316	2300	1355	1849	2539	2582	2414	1133	4749	2194
Medium	6714	5214	30675	22596	13966	18657	32980	9482	44807	5015	19011
Hard	142250	29117	70060	94950	18718	21287	41077	70116	16068	40061	54370

Table 5.4: Results for creating a Sudoku board in Vue with Performance tool using Mozilla Firefox.

From the results in the tables above it is clear that based on our implementation that Vugu performs much greater than Vue. With the difficulty set to "Easy", the duration for the DOM event in the Vugu application averages at 0,16 seconds while the DOM event for the Vue application averages at 2,19 seconds. Furthermore when the difficulty is set to "Medium" the Vugu application averages at 0,32 seconds while the Vue application averages at 19 seconds. Lastly, when the difficulty is set to "Hard" then the Vugu application averages at 8,3 seconds while the Vue application needs an average of 54,37 seconds to complete.

5.3 Performance test when creating a Sudoku board

Plotting these results for creating a Sudoku board into a line graph as shown in Figure 5.4, shows that the Backtracking Recursive algorithm used in the function that creates the Sudoku boards is exponential in time. As a result of this, creating an even harder Sudoku board will result in a massive increase in time waiting for a Sudoku board to be ready. The results also show that the implementation used in the Vue and Vugu applications seems to favor Vugu as the Vue application grows much faster in the exponential time. When creating a Sudoku board with the difficulty "Easy" the Vugu application outperforms the Vue application by 991%, with the difficulty "Medium" the Vugu application outperforms the Vue application by 5841%, and lastly, when the difficulty is set to "Hard" the Vugu application outperforms the Vue application by 554%.

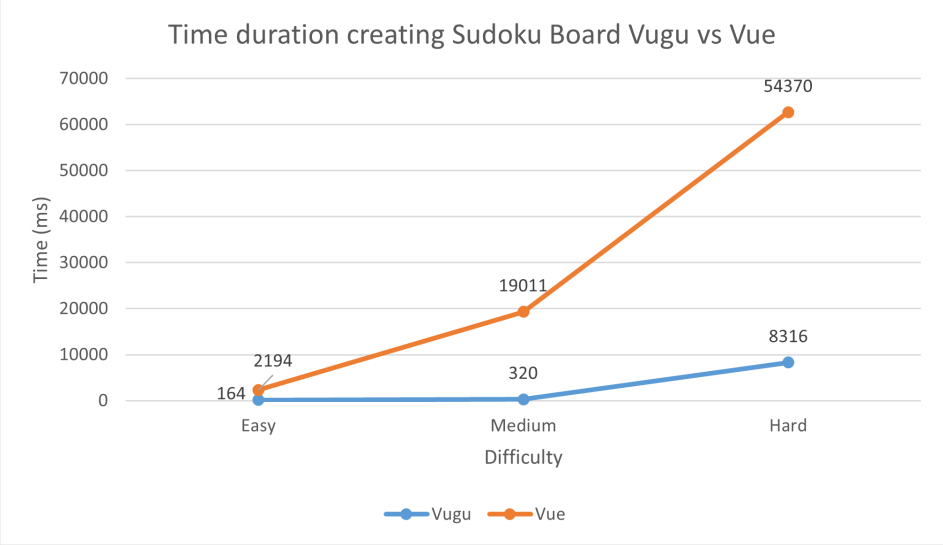


Figure 5.4: Difference in time between Vugu and Vue.

5.3.1 Analyzing the Call-Tree

The Figure 5.3 above from Section 5.2 shows an example of how a result from the Call-Tree sub-tool could look like for a test run. The fields "Samples," "Self Time," and "Self Cost" from the Figure 5.3 are not relevant for the measuring done here. "Samples" is the number of samples that were taken during the execution of that function. "Self Time" shows the time spent in that exact function without its children, which means other functions inside that exact function. "Self Cost" is the percentage of the total number of samples. The last two fields, "Total Time" and "Total Cost", are the ones that are relevant and will be further discussed below. "Total Time" is the time used to execute a given function. "Total Cost" is a percentage of the total number of samples in the recording. "Total Time" will be around the same number as "Samples" and hence "Samples" is not relevant for this measuring.

Reading the values from the Call-Tree when the degree of difficulty is set to "Easy" for creating a Sudoku board, gives little to no insight in performance since the time used for creating a Sudoku board is so low and the functions take less than 10ms. Gecko, which is Firefox own browser engine, alongside some built-in functions in WebAssembly is the most time-consuming function when difficulty is "Easy" for the Vugu application. Therefore in tests concerning the Call-Tree sub-tool, "Hard" will be used as the difficulty to generate results for further comparison. In Table 5.5 and 5.6 are the relevant fields from the Call-Tree listed.

Total Time	Total Cost	Function
7972	48.32%	isBoardValid
4353	26.38%	hasDuplicates
2621	15.85%	runtime
786	4.77%	runtime
467	2.83%	Counter

Table 5.5: Overview over relevant data from Call-Tree sub-tool; Vugu application.

From Table 5.5, reading the values generated in the Call-Tree by the previous recordings from Table 5.3 generated the following Call-Tree for the Vugu application, shown in Table 5.5 above. The function isBoardValid is the most time-consuming for the Vugu application. This is not unexpected since this function must verify changes done to the board when trying to create a Sudoku board. The function hasDuplicates is a helper function for isBoardValid so it makes sense that it follows as the second most significant function. Then there are some runtime functions belonging to the WebAssembly file. The last function worth mentioning that uses a significant amount of time is the Counter function, which is the function mentioned in Section 4.2.2 responsible for making sure that the board maintains its unique solution.

Total Time	Total Cost	Function
36469	28.05%	isBoardValid
27730	21.33%	chunk-vendors
27302	21.00%	garbage collection
18473	14.21%	hasDuplicates
2779	2.14%	Counter

Table 5.6: Overview over relevant data from Call-Tree sub-tool; Vue application.

Looking at the Call-Tree generated from the table holding the Vue results in Table 5.6 above, there are some distinct differences to the Vugu Call-Tree. Similar to the Vugu application, we see that the function isBoardValid is the one that occupies most of the recording. Then we see that the Call-Tree is populated by 3 distinct blocks that play a great part in the fact that the Vue application needs a lot of runtime before being able to display the queried Sudoku board. Chunk-Vendors is a bundle for all the modules that are not shipped from the Vue application, but from other parties, called third-party modules. In the Vue application, these are located in the /node_modules directory that holds amongst other things all the Vue dependencies. Some optimization can be made here, such as i.e. setting the webpack mode to production mode, which will reduce the data sent from node_modules to the browser, but this is not done here for comparison reasons since it is not needed for the Vugu application.

The next obvious difference in Table 5.6 apart from the Vugu Call-Stack in Table 5.5 is that the garbage collector is active for 21% of the recording in the Vue application. The garbage collector is a process from the browser that attempts to find and release memory that no longer is used by the application. The fact that the garbage collector is so insignificant for the Vugu application "Call-Tree" and significant for the Vue variant "Call-Tree" is an interesting fact because the implementation for the two applications are the same, but it interferes with the garbage collector differently. This means that JavaScript and Go code are handled differently by the browser. The implementation is giving the Vue variant very little justice, as the garbage collector is extending the rendering with an entire 27,3 seconds.

Lastly, the last two blocks in the Vue generated Call-Tree from Table 5.6 are occupied by the functions hasDuplicates and Counter which matches the Vugu generated Call-Tree from Table 5.5. The function hasDuplicates has a Total Cost of 14.21% which is approximately 50% less than the Total Cost for the isBoardValid function, which corresponds nicely with the results from the Vugu generated Call-Tree. The same goes for the Counter function, which for the Vue generated Call-Stack occupies 2.14% of the recording, whereas occupying 2.83% in the Vugu generated Call-Tree.

5.4 Performance testing of the application in different browsers

There are different preferences for which web browser to use amongst users and web developers. Therefore it could be interesting to compare the applications performance in different browsers. For this performance test the 3 browsers; Mozilla Firefox, Google Chrome and Microsoft Edge were selected, and the recordings were made with "Medium" as the degree of difficulty.

The earlier results in Table 5.3 and 5.4 is used as the base value for comparing the results in this section. The result from Mozilla Firefox were 0,32 seconds for the Vugu application, and 19 seconds for the Vue application.

Result No.	1	2	3	4	5	6	7	8	9	10	Average (ms)
Firefox	394	145	235	491	414	276	270	158	650	169	320
Chrome	230	411	324	650	473	206	390	216	303	269	347
Edge	291	598	516	339	306	208	237	174	1215	102	399

Table 5.7: Overview over results for creating a Sudoku board in Vugu with Performance tool in different web browsers.

Result No.	1	2	3	4	5	6	7	8	9	10	Average (ms)
Firefox	6714	5214	30675	22596	13966	18657	32980	9482	44807	5015	19011
Chrome	37282	7843	4302	58181	8157	32425	3617	11156	19071	4539	18839
Edge	16472	36549	13329	26911	14523	18699	10767	5524	18050	10734	17156

Table 5.8: Overview over results for creating a Sudoku board in Vue with Performance tool in different web browsers.

Based on these results there is to some degree a difference between each browsers performance when running the Vugu and Vue application. For the Vugu application, with its results located in Table 5.7, the Chrome web browser seems to be approximately 8% slower than Firefox, and the Edge browser seems to be approximately 24% slower than Firefox. For the Vue application, there is a different scenario. From the results for the Vue application seen in Table 5.8 it is clear that out of the 3 browsers, Mozilla Firefox was the one who performed worst, performing 1% slower than Google Chrome, and 11% slower than Microsoft Edge.

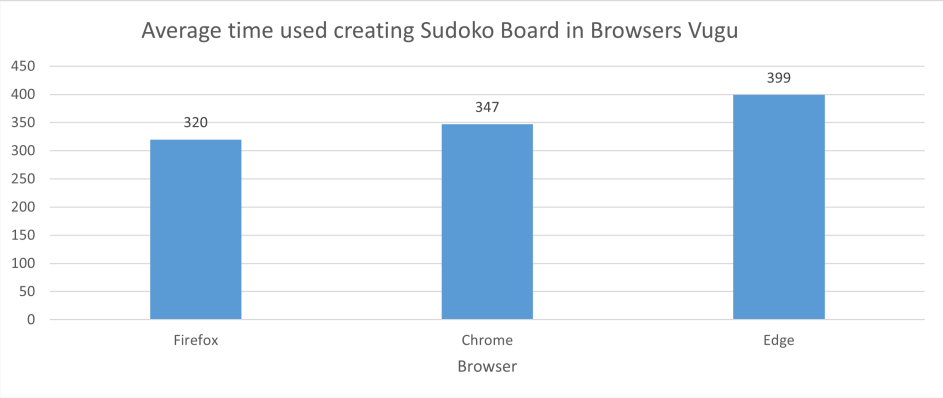


Figure 5.5: Average time duration creating a Sudoku board with difficulty "Medium" in different browsers; Vugu application.

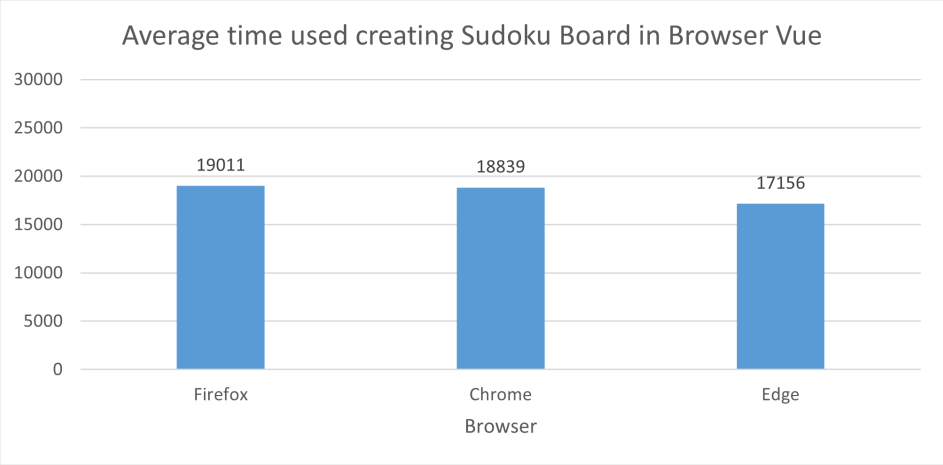


Figure 5.6: Average time duration creating a Sudoku board with difficulty "Medium" in different browsers; Vue application.

5.4 Performance testing of the application in different browsers

Figure 5.5 and 5.6 shows the average time duration each browser used creating a Sudoku board with difficulty "Medium" for the Vugu- and Vue application respectively. From looking at the graphs and percentage of difference it may look like there is a difference, but when it comes to usability the time difference is almost not noticeable for most of the cases. The results can also be little misleading. This is most noticeable for the Vugu application where a 24% difference between Firefox and Edge seems like a big difference, but in fact it is only 79 milliseconds difference, which is not noticeable. For the Vue application there is also the case where Firefox is 11% slower than Edge which may not seem like that big of a difference, but in milliseconds this corresponds to 1855, which is approximately 1,86 seconds and that is a difference that could be noticeable. Because of the limited number of tests and small differences in the results, it is not possible to say that this is a factual result. To get a decisive decisive from this test case, more tests should be completed and evaluated.

5.5 Perfomance test when solving a Sudoku board

The implementation also has some logic in place for solving the board after it has been created, described in Section 4.2.3. To test how this logic performs and compare the applications to each other, the same test approach is also utilized here. By recording the application using the built-in performance tool with Mozilla Firefox, the results were as displayed in Table 5.9 and 5.10 for the Vugu- and Vue application.

Result No.	1	2	3	4	5	6	7	8	9	10	Average (ms)
Easy	14	12	11	10	16	11	10	11	9	12	12
Medium	14	16	18	21	11	15	11	15	15	26	16
Hard	54	49	13	87	51	60	35	138	60	12	56

Table 5.9: Overview over results for solving a Sudoku board in Vugu with Performance tool in Firefox.

Result No.	1	2	3	4	5	6	7	8	9	10	Average (ms)
Easy	212	168	79	226	169	174	144	198	176	170	172
Medium	190	229	201	340	287	167	215	221	255	205	231
Hard	901	740	804	720	787	802	880	867	796	779	808

Table 5.10: Overview over results for solving a Sudoku board in Vue with Performance tool in Firefox.

Based on the results displayed above, it is clear that solving a Sudoku board in both applications is much faster than creating a Sudoku board. Another thing to notice is the continuation of the trend that the Vugu application continues to outperform the Vue application. In this case the difference in performance is also quite significant.

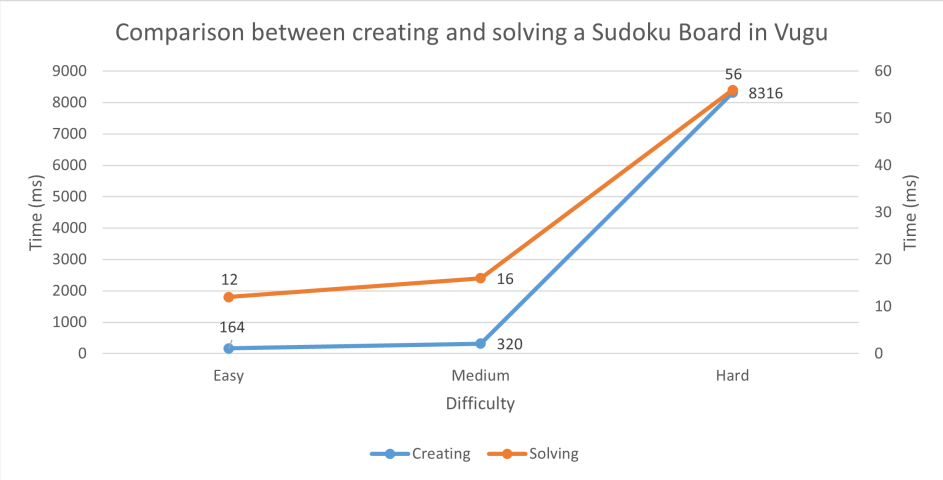


Figure 5.7: The time duration for creating and solving a Sudoku board in the Vugu application.

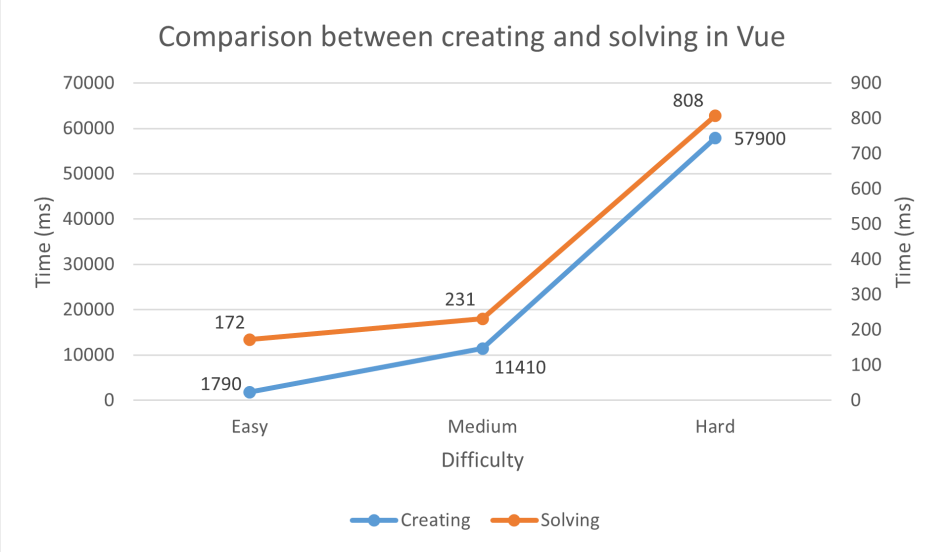


Figure 5.8: The time duration for creating and solving a Sudoku board in the Vue application.

By comparing the average time for solving a Sudoku board with the average time of creating a Sudoku board, as illustrated for both applications in Figure 5.7 and Figure 5.8, it is clear that these continue to grow proportional to each other in time. This is because of that the runtime of the brute force approach involved in solving the Sudoku board will grow exponential along with the total number of attempts to solve the board. This also is the case with the Backtracking Recursive algorithm used in creating the Sudoku boards, since the runtime will grow exponential with the number of cells removed.

5.6 Summary

Based on the results derived from the sections above, it is clear that the Go and Vugu application outperforms the JavaScript and Vue variant for all of the performance cases tested in this thesis. Therefore it is safe to say that for this type of project, Go and Vugu is the optimal choice over JavaScript and Vue if performance is the only measurement. This could be observed already from the first test case in Section 5.1, where the terminal executed Go scripts performed significantly better than the JavaScript scripts executed in the terminal with Node.js. This passed on and became a trend onto the following test cases where the Vugu and Vue applications were executed in the browser for Section 5.3, 5.4 and 5.5.

Chapter 6

Test Capabilities and Support in Vugu and Vue

At any given point in the software development life cycle, one might easily implement bad code or introduce bugs to an application. Therefore it is important for any project that the application is tested thoroughly throughout the development cycle to identify defects, reduce flaws and increase the overall quality of the application. Due to Vue already being a fairly established framework with lots of support around the project, there already exists test frameworks that can be used to maintain the quality of the application. Vugu on the other hand has no support for extensive testing to verify that the Vugu files and components work as intended. Hence the main purpose of this chapter is to highlight how testing might be done for a Vugu application today, and how this was achieved in this thesis.

6.1 Testing capabilities in Go and JavaScript

Testing in Go is simple and intuitive using package testing[14]. This package is a part of the Go standard library and using this package when creating unit tests is fast and simple. As long as the functions are in a Go file it is easy to write test cases for each function. JavaScript has many dedicated test frameworks, i.e. the Jest framework[15]. This was used during the development of both the Vue application and the JavaScript backend. Jest is a JavaScript testing framework that was built by Cristoph Nakazawa[16], and is now maintained by Facebook. Jest is a testing framework that is simple to set up, and can easily be added to any project using the Vue CLI mentioned in Section 3.2.1. Below in Figure 6.1 and Figure 6.2 are unit test examples written for a function used in both the Go and the JavaScript backend.

```go
func TestRandomNumber(t *testing.T) {
    num := RandomNumber()
    if num < 1 || num > 9 {
        t.Errorf("Number must be between 1 and 9, it was %d", num)
    }
}
```

Figure 6.1: Simple test in Go.

```javascript
test('Generate a random number between 1 and 9', () => {
  let num = RandomnNumber();
  expect(num >= 1 && num <= 9).toBe(true);
});
```

Figure 6.2: Simple test in JavaScript using Jest.

Comparing the two tests seen in Figure 6.1 and 6.2 it is clear that they work and is setup in a similar way. Both of the tests are testing the RandomNumber() function which is supposed to return a number between 1 and 9. If it does not return the expected result, the test will fail.

6.2 Test capabilities in Vugu

In Vugu there is no framework, at this point in time, dedicated for testing the Go code and functions in a Vugu file. There is a way around using the Go package testing that was achieved in this thesis. This is done by using the functions created in the vgen.go file that is generated when running the devserver.go file. This way it became possible to test the output of the Vugu components. This approach will now be reviewed.

The hard part about testing in Vugu is to test if the components have the desired output. An important test to verify this is to test that the Vugu functions generates the desired HTML code output. Since there is no testing framework in Vugu for testing this, a solution is to use the Build function from the vgen.go file. This is a function that is created for every component and is the function that builds the component that is rendered in the browser. However, this is not intuitive and makes the code less readable.

Creating a test using this approach is done by creating a test page in the Vugu application as seen in Figure 6.3. On this page, there exists two lists, one ordered list and one unordered list, both with three elements. The ordered list is written in straightforward HTML code and the unordered one is written using the Vugu function "vg-for". This function is a for-loop that creates the number of intended list elements.

Figure 6.3: Test component as seen in the browser (left) and the code file that generates it (right).

Then to test the output of the "vg-for" function, a test case as the one that can be seen in Figure 6.4 and 6.5 is created using package testing as if the test was for an ordinary Go function. This test must then check if using the "vg-for" gives the same result as HTML code. This is to be able to verify that the Vugu function works as intended when used. This will also make it possible to write a test that can verify that Vugu files and components creates the intended HTML code and output.

A test to verify generated HTML output can be made by first creating an struct of the component that is going to be tested. For this case, it would be an struct of Test. This enables the use of the Build function of the Test component from the vgen.go file. The Build function has a parameter which is an object of BuildIn and hence this must be created using vugu.BuildIn. When putting the BuildIn object as a parameter in the Build function the output will be an object of BuildOut. The BuildOut object has four data fields, which are; Out, Components, CSS, and JS. It is the Out which is an object of VGNode that will be used in the test. The Out is a tree over the HTML code created in the component, which in this case is the Test component.

By applying this approach it may look like that it is possible to check if the elements inside both of the lists are equal by traversing the node tree, but when writing the list elements in HTML code the <li> elements do not get appended to the node tree. The solution to finding the number of <li> elements that have the <ol> element as a parent is to find the <ol> element and then use the data field innerHTML that all VGNode objects have. This data field then shows what the <ol> element has as children.

One solution to this problem is to check the number of <li> elements that are created inside the unordered list as seen in Figure 6.4. One drawback with this is that the number of <li> elements must be known, else there is no point in checking the number of list elements. This method allowed it to be possible to create a unit test that checks if the "vg-for" did create the desired number of <li> elements. To confirm that everything is correct, it is also possible to visually verify the number of <li> elements by checking the Vugu application in the web browser. This was done and it proves that it is possible to write tests that can verify that the Vugu functions give the desired output.

```
125  func TestBuild(t *testing.T) {
126      test := Test{}
127      buildIn := &vugu.BuildIn{}
128      vgout := test.Build(buildIn)
129
130      tree := vgout.Out[0]
131      liElements := []string{}
132
133      ul := tree.LastChild.PrevSibling.LastChild.PrevSibling
134      ul_child := ul.FirstChild
135
136      for {
137          if ul_child.NextSibling.NextSibling == nil {
138              break
139          } else {
140              liElements = append(liElements, ul_child.NextSibling.Data)
141              ul_child = ul_child.NextSibling
142          }
143      }
144      if len(liElements) != 3 {
145          t.Errorf("Length must be 3, it was %d", len(liElements))
146      }
147  }
```

Figure 6.4: Test counting if the number of <li> elements is correct; Vugu variant

The other approach of verifying the output of "vg-for" is to first find all <li> elements that are appended to the tree and then use the innerHTML data field from the <ol> element to find the <li> elements that are coded in HTML. Then compare the <li> element from the tree with the <li> elements from the innerHTML data field. This approach is illustrated below in Figure 6.5. If both are the same length then it is verified that "vg-for" works as intended. This could then later be used when testing that a Vugu application gives the desired output.

```go
func TestBuildHTML(t *testing.T) {
    test := Test{}
    buildIn := &vugu.BuildIn{}
    vgout := test.Build(buildIn)

    tree := vgout.Out[0]
    liElements := 0

    html := *tree.LastChild.PrevSibling.FirstChild.NextSibling.InnerHTML
    html = strings.Replace(html, " ", "", -1)

    for i, _ := range html {
        if html[i] == 10 {
            if i+3 > len(html) {
                break
            }
            line := string(html[i+1]) + string(html[i+2]) + string(html[i+3])
            if line == "<li" {
                liElements++
            }
        }
    }

    ul := tree.LastChild.PrevSibling.LastChild.PrevSibling
    ul_child := ul.FirstChild
    liElementsList := []string{}

    for {
        if ul_child.NextSibling.NextSibling == nil {
            break
        } else {
            liElementsList = append(liElementsList, ul_child.NextSibling.Data)
            ul_child = ul_child.NextSibling
        }
    }

    if liElements != len(liElementsList) {
        t.Errorf("Failed test! Both list elements must be equal. OL: %v, OL: %v", liElements, len(liElementsList))
    }
}
```

Figure 6.5: Test comparing innerHTML with elements from the tree.

From both Figure 6.4 and Figure 6.5 it is clear that these are tests with low readability and they are time consuming to create. But since there exists no real alternative to this approach when creating tests for Vugu, this is the approach to use at this point in the development of Vugu.

6.2.1 Test capabilities in Vue

The same HTML code as shown in Figure 6.3 can also be tested using the Jest testing framework for components rendered with Vue, shown below in Figure 6.6. The describe block is used to group the test cases belonging to a single component. Within this describe block, a wrapper object is used for mounting components in addition to allowing for usage of predefined methods from the Vue Test Utils. In this case, the TestPageComponent containing the HTML code shown in Figure 6.3 is mounted, and from here assertions are used to first find all the <li> elements from the component and check if it corresponds with the expected amount.

```
describe('TestPageComponent', () => {
    test('Get the li elements from the test page component', () => {
        const wrapper = mount(TestPageComponent);
        const liElements = wrapper.findAll('li');
        expect(liElements.length).toBe(3);
    })
});
```

Figure 6.6: Test counting if the number of <li> elements is correct; Vue variant

Comparing the code snippets from Figure 6.4 and Figure 6.6 to each other, the clear advantage that Vue has over Vugu in this area shines through. The fact that Vue is an established framework has opened the door to the Jest testing framework, and due to the many methods from the Vue Test Utils, the test that needed 30-40 lines of code in Go, only require 4-5 lines of code in JavaScript when using Jest. The code snippet from the JavaScript and Vue variant is also highly readable, especially compared to the Go and Vugu variant of the test.

6.3 Support around Vugu and Vue

The support and community existing amongst the vast diversity of projects and programming languages plays an important part in the world of programming. If it is hard for developers to find documentation and help when stuck, the project may take longer to finish. Also, new and inexperienced developers will have a harder time getting familiarized and learn how to use the different technologies.

Vue is a fully developed framework for JavaScript, and hence there is a lot of information and resources available on the internet besides the official pages and documentation for Vue[17]. This will make it easier for developers new to Vue to learn how to develop web applications using the framework. The community around Vue is a big advantage since it will be much easier for developers to find help when there are more people experienced in the framework and thus leading to problems being solved quicker.

When comparing the support and community of Vugu to that of Vue, it is clear that Vugu still has a way to go. Apart from the Vugu web page[6], documentation[18], and GitHub[19], there are limited resources available. This will make it harder for developers to learn how to use Vugu. This especially concerns developers that are inexperienced with web development and frameworks such as Vue. Hence Vugu comes up short when comparing the limited resources available to the diversity of resources available for Vue.

6.4 Summary

When summarizing this chapter it is clear that Vue has an advantage over Vugu in regards of both the test capabilities and the support around the framework. Vugu does not have a library or framework for testing except the Go package testing. Whereas Vue has plenty of available resources for this purpose, i.e. the Jest testing framework.

As demonstrated above in Section 6.2 testing components and code structure in Vugu can be done, but at this point in time, this is very complicated and also very time inefficient as compared to Vue. For Vue, where there already exist sufficient and reliable testing frameworks, functionality in Vue components, and applications in general, can easily be verified without this becoming complicated and time-consuming tasks.

The support and community around Vue is much bigger than that of Vugu and hence the resources available are larger and easier to find. Meaning that if issues occur while using Vue, solutions are most likely to be found compared to Vugu. This could be an important note to regard when choosing a framework between Vugu and Vue, especially if the developers are inexperienced in web development.

Chapter 7

Final Conclusion and Further work

For the final chapter of this thesis, an overall conclusion will be presented. Thoughts, results, and experiences during the development process will also be discussed. Finally, ideas for further work to either enhance the application of this thesis or to improve the development for future projects will be proposed.

7.1 Conclusion

The question of whether it would be possible to implement a similar Sudoku application in both Vue and Vugu proved itself to be possible. Similar features and functionality were achieved during the development process which made a solid base for the comparison part of the thesis. The applications were implemented with as few differences as possible, with the goal of narrowing down eventual differences to be related to the nature of the programming language of the framework.

7.1 Conclusion

When looking at the overall performance of the applications, and comparing the results, they showed that the Vugu application overall performed better than the Vue application. This was not unexpected given the Backtracking Recursive algorithm involved in computing the Sudoku boards, where it was expected that Go would outperform JavaScript. It is also important to note that Vue and JavaScript were not given much justice in terms of optimal performance, as the application was developed with features and functionality where WebAssembly and hence, Go and Vugu, should thrive.

With Vue having the Vue CLI, mentioned in Section 3.2.1, it is clear that getting started with a Vue project is a much more comfortable and time-efficient process than getting started with a new project in Vugu. When starting up a project in Vue, simply select the features needed for the project either its Vuex, Vue Router, or adding a unit testing framework, and the CLI will make sure that all dependencies are installed correctly. To achieve the same by using Vugu, given the fact that Vugu does not offer a CLI, own research was needed to ensure the starting point required for the particular project. For the application of this thesis however, the fact that Vugu does not have an own CLI did not create any particular hassle, but for greater projects dependent on several developers working as a team, Vue has a beneficial advantage over Vugu in terms of getting started with a new project.

Since Go is a very lightweight programming language and Vugu is utilizing WebAssembly to compile and render applications, it was expected that it would perform better than the JavaScript and Vue application. From looking at the results from creating a Sudoku board in Go and JavaScript in Section 5.1 it became clear that Go most likely would outperform JavaScript in most of the aspects of the application. The Backtracking Recursive algorithm, used for most of the computational logic, grows exponentially in time, which is why there is such a big difference in the measured performance. Based on the results from Chapter 5 it becomes clear that choosing Vugu and Go would be optimal over selecting JavaScript and Vue for the types of applications that require heavier computational operations to some extent.

The Vue variant could most likely be optimized to achieve results closer to what the Vugu variant achieves, but this is not opted for in this thesis, as this would have lead to changes in the application and introduced more differences to the implementation compared to the Vugu implementation. Since the objective was partly to see if Vugu and WebAssembly could perform better for the types of applications where it should shine compared to JavaScript and Vue, similar implementation was prioritized at the expense of optimizing the respective applications.

7.1.1 Testing Frameworks

As of the status quo, Vugu does not have any dedicated testing framework for testing the output of Vugu files. This means that it is harder to ensure that the written code works as intended as applications grow larger and more complex. Vue has an obvious advantage over Vugu at this point since there exist several testing frameworks compatible with Vue for this purpose, i.e. the Jest testing framework. Therefore, as the components and written code in Vue are testable, it is easier to keep the Vue code bug-free as the application grows larger and more complex. There are possibilities to test the Vugu application as demonstrated and proven in Section 6.3, but this is neither intuitive nor a time-efficient process.

7.1.2 Features

Both Vue and Vugu have features to help developers create applications. Some of these features are similar for both frameworks and provides the same functionalities. This are i.e. Vugu Wiring for Vugu and Vuex for Vue which represents its respective frameworks state management pattern. Routing is also a feature both frameworks provide, and makes no differences from a user's point of view when used in applications. Vugu may still be in an experimental state, but while developing the Sudoku application regardless of which variant, there were no features one had and the other did not have.

7.1.3 Support and Community

As mentioned in Section 6.3, Vue has a lot more resources and documentation available on the web compared to Vugu. This is mostly because Vugu still is in an experimental and early phase, with its first commit to the official Vugu Github on the 20th of March 2019[19]. There are some places to find resources and help while creating a Vugu application, but those are limited to the official Vugu page[6] and Github. Vue however is a fully developed framework for JavaScript and is used in real applications everywhere, i.e. the popular Chess.com[20] platform. This leads to the fact that it is much easier to find information, documentation, and examples everywhere when creating a Vue application. Because of this, Vue would probably be the better choice for inexperienced developers when choosing which framework to use for their project.

7.1.4 Final thoughts

When summarizing everything it becomes clear that Vugu should be the obvious choice based on the results derived in this thesis. Thus, confirming that Vugu and WebAssembly is the best alternative to developing demanding web applications, acting as a "helping hand" to JavaScript in the areas where JavaScript comes up short performance wise. However, because of the state of development that Vugu still is in, Vue would be a better choice for a real application that is to be deployed in the near future, and does not depend on heavy computational operations. An example of this might be a Tic-Tac-Toe game. The reasoning behind this is because Vue is a fully developed framework with a big community. This will make it easier to develop and maintain in comparison to Vugu, which is continuously getting updates that could affect an application. Something like this was experienced late in the development stages of the Vugu application, where additional support features to the Vugu root file was deployed to the Vugu documentation[6] by the Vugu team, which could have introduced changes to the Vugu application in this thesis.

7.2 Further work

In this section possible tasks and implementations to the application or frameworks, especially Vugu, will be discussed. Because Vugu is still under development and in an experimental state, Vugu has the most suggestions for further work. The Sudoku application overall also has a lot of possibilities regarding improving the application as a whole.

7.2.1 Test framework for Vugu

Vugu does not have a dedicated testing framework that allows users of Vugu to create simple and intuitive tests to verify that the application does what it is intended to do. With a test framework, the potential test capabilities and the quality of work would increase. This would again lead to faster, easier, and better development of Vugu applications. Large projects in Vugu would be easier to test and distribute with a test framework. Because of the lack of a test framework for Vugu, a future assignment could i.e. be to create a simple example of one. One feature of this framework could be to print out a tree structure over the HTML nodes. This would allow users to check that Vugu commands such as "vg-for" works correctly. It will also help to keep an order of how the different Vugu components are built up.

Another useful feature to the test framework could be to test the different features of the application. An example of a test such as this could be that when an action like clicking a button to show a division is triggered, then after the button click, a test could be implemented to verify that the button is clicked and that the correct division is being shown. This would be way easier than rather having to start up the application and visually verify whether or not the button works as intended, for every newly added segment.

7.2.2 Command Line Interface (CLI) for Vugu

As of the status quo, Vugu does not have its own CLI, or anything that comes close to the Vue CLI. By creating an own CLI for Vugu, it could help and guide new and inexperienced developers to find Vugu as an attractive framework for learning web development. By having a CLI with a UI similar to Vue's own CLI, giving developers the alternative to choose for themselves if they want Vugu wiring or Vugu routing pre-installed in their projects, ready to be used, could lead to an increase in popularity as this would take away the need for a very technical understanding of the various concepts and how to tie them together. Therefore this could also be a potential meaningful future assignment.

7.2.3 Improvements to the application

Because the goal of the thesis was to compare Vue and Vugu against each other, some implementations that were irrelevant for the thesis were left out. Other implementations were done in such a way that the differences between Vue and Vugu would be shown. Removing and adding implementations that overall will improve the Sudoku application could then be an assignment for the future. There are several ways to improve the overall application. Some of those improvements will be discussed below.

Improvements could be more features that would make the application more usable and more interesting for new users. Example of different additional features that could be implemented to achieve this could be:

- Print out the created Sudoku board, and afterwards scan/upload image of your proposed solution to check if its correct.
- System that stores Elo ranking, as described in the following link `https://www.chess.com/terms/elo-rating-chess`, for players, feeding them suitable boards corresponding with each players respective Elo.

By adding one or both of these features to the application, the use of the application would be more than just to compare Vue and Vugu.

Another possible improvement is to implement better and faster functions that will improve the overall response time of the application. The functions that create the Sudoku board are an obvious candidate to improve as seen in the results in Table 5.5 and Table 5.6. One way that could be done is by removing the randomness. This randomness could be removed by a deeper knowledge of Sudoku and how it works. This could then result in a way to remove cells and create a Sudoku board without the factor of randomness.

Bibliography

[1] Ben Popper. The 2020 developer survey results are here! https://stackoverflow.blog/2020/05/27/2020-stack-overflow-developer-survey-results/, 2020. [Accessed: 02.04.2021].

[2] TIOBE the software quality company. Tiobe index for 2021. https://www.tiobe.com/tiobe-index/, 2021. [Accessed: 10.04.2021].

[3] Ryan Dahl. Github ryan dahl. https://github.com/ry, 2021. [Accessed: 21.04.2021].

[4] TIOBE. Tiobe index. https://www.tiobe.com/tiobe-index/, 2021. [Accessed: 25.03.2021].

[5] Evan You. Github evan you. https://github.com/yyx990803, 2021. [Accessed: 14.04.2021].

[6] Vugu. Vugu: A modern ui library for go+webassembly. https://www.vugu.org/. [Accessed: 24.03.2021].

[7] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[8] Manuel Rubio-Sánchez. *Introduction to Recursive Programming*. CRC Press LLC, Boca Raton, Florida, 2018.

[9] Narasimha Karumanchi. *Data Structures And Algorithms Made Easy*. CareerMonk Publications, IIT Bombay, Mumbai, India, 2017.

[10] Paul C. Jorgenson. *Testing: A Craftsman's Approach, Second Edition*. CRC Press LLC, Boca Raton, Florida, 2002.

[11] Mozilla. Performance. `https://developer.mozilla.org/en-US/docs/Tools/Performance`, 2021. [Accessed: 26.03.2021].

[12] Corey Scott. *Hands-On: Dependency Injection in Go*. Packt Publishing Ltd., Livery Place, 35 Livery Street, Birmingham, 2018. So, how do I define DI?, Page 8.

[13] Gilles Civario Gary McGuire, Bastian Tugemann. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem via hitting set enumeration. Technical report, School of Mathematical Sciences, University College Dublin, Ireland, 2013. Abstract, Page 1.

[14] Go. Package testing. `https://golang.org/pkg/testing/`, 2021. [Accessed: 12.05.2021].

[15] Jest core team. Jestjsio. `https://jestjs.io/`, 2021. [Accessed: 11.04.2021].

[16] Christoph Nakazawa. Christoph nakazawa github. `https://github.com/cpojer`. [Accessed: 23.04.2021].

[17] Vue team. Vue documentation. `https://vuejs.org/v2/guide/`, 2021. [Accessed: 11.04.2021].

[18] Go. Vugu. `https://pkg.go.dev/github.com/vugu/vugu`, 2021. [Accessed: 07.04.2021].

[19] Vugu. vugu. `https://github.com/vugu/vugu`, 2021. [Accessed: 07.04.2021].

[20] Chess.com team. Official chess.com platform. `https://www.chess.com/`. [Accessed: 23.04.2021].

List of Figures

LIST OF FIGURES

List of Tables