Universitetet
i Stavanger

Faculty of Science and Technology

BACHELOR'S THESIS

Study program/Specialization:	Spring 6th semester, 2019
Computer Science	Open
Writer(s):	Signature
Christoffer Holmesland	.....
Jørgen Melstveit	.....
Gaute Haugen	.....
Faculty supervisor: Erlend Tøssebro	
Thesis title: Web application + mobile application for use in interactive teaching - DAT200 and DAT110	
Number of credits: 20	
Key words: Interactive teaching Web application Algorithms & data structures Drawing tool	Pages: 74 + enclosure: zip archive Stavanger, 15/05-2019

Frontpage for bachelor thesis
Faculty of Science and Technology

Table of Contents

1 Introduction

1.1 Task

The task for the bachelor thesis consisted of developing a game-based application that was to be primarily used in the courses DAT110 and DAT200. The application allows students to play games consisting of questions relevant to the topics introduced in the course.

The application was to take inspiration from Kahoot's interactive game system[1], where the questions are displayed on a projector and students can answer the questions using their mobile devices. One key difference between the application and Kahoot's is that the application needs to support a variety of question types. One of the struggles when doing a quiz for computer science subjects is that there are not any question types available to visually show how algorithms work. The application should give the students the ability to interact with data structures. This includes drawing trees and graphs. It should also be possible to work with arrays. The former required a basic drawing tool to also be implemented for the application.

The results of the game session were to be stored for the instructor. The instructor can then use the data in order to clear up common mistakes and use it to improve upon future lectures.

The primary target for the application was mobile devices, however, it was decided that the application would be more suited as a web application supporting use of both computer and mobile devices.

1.2 Goals

The following is a list of the goals, where the main goals were part of the bachelor thesis description. The sub-goals are some of the goals we added to deliver a more complete application.

1.2.1 Main Goals

- Allow students to answer questions about a data structure using a drawing tool with automated solution checking.
- Allow users to participate in the sessions using a mobile phone or a computer.
- Allow lecturer to see incorrect answers after the session is over.

1.2.2 Sub Goals

- Allow students to store their data using their Feide user as an identifier.
- Include automated solution generator.
- Allow students to compare their answer against the solution after the session is over.
- Make the application scalable for new courses and question types.

- Support language localization.
- Support modern browsers.

1.3 Motivation

The motivation for this application is to give students the opportunity to not only learn about algorithms and data structures by watching the instructor manually implement all the structures. Doing exercises will allow the students some time to think about what they learned during the lecture. This also allows the student to test themselves, and see whether or not they've actually learned anything from the lecture. The result after each session is given to the instructor, which in turn can be used for the instructor to amend future lectures. In a sense, the application gives both the student and the instructor reliable feedback involving individual and class performance respectively.

Other game-based learning applications like Kahoot have seen a lot of use in schools and universities, and for good reason, since it helps students interact more during the lecture. However, Kahoot has some flaws that were addressed early on during the planning of our project. This application is designed in a way to improve upon or prevent these flaws. For instance, Kahoot has too much of a focus on being a competition rather than self evaluation. This means a lot of students can get discouraged from participating or in general be afraid of making mistakes. Kahoot also encourages the players to answer quickly, this can lead to a lot of students answering the first thing that comes to mind, instead of taking their time to think about each question.

1.4 Workflow

When developing a web app, one crucial step is to make sure that the users will be able to use your application. To make sure that the tools we used where supported, we used caniuse.com[2] with some of the HTML elements to make sure that a high enough percentage of browsers would be able to use this application. This website allows you to search up a feature, and see statistics about which version of a browser supports it. The website also gives information about known issues in the different browser versions, and also the same information about subfeatures. When developing this web application we had a goal to support at least 90% of installed browsers. This goal was accomplished, and the features that are not supported are for browser versions that are outdated, and where the user should update to the newest version. All our features should work if the latest version of the largest browsers is used, e.g., Google Chrome, Firefox, Edge, Opera.

With previous experience using git and working with a combination of kanban and scrum, the choice was natural when we decided what version control management system we wanted to use. We started a project on GitHub using git as the version control manager. On GitHub, we split the project into sub-components as far as it was possible to make sure that we knew what had to be done at any moment. In combination with kanban on GitHub, we also wrote down sprints each week were we delegated work between each other. We were not very strict on

finishing each sprint, but the task was usually the preferred amount of work to get done for a week. We also set up a long term plan for when larger blocks of the project should be finished.

When choosing languages for writing the application, we had a few languages to choose from. With previous experience with a full-stack application in JavaScript with Node.js the previous semester this ended up being our choice for the server side language. When choosing the language, we also made sure that a Node.js server would be able to serve all the features required for our application. For the client, we knew that we needed HTML, JS, and CSS, but we wanted to use frameworks to make the development easier. We choose Vue as a framework for making HTML and JS. For CSS we used Bootstrap in combination with standard CSS.

2 Background

2.1 Canvas

In web development, there are three ways to display graphics to the user. HTML elements can be added and removed to the page using JavaScript, and elements can be styled using CSS. This method is very slow because manipulating the DOM (Document Object Model) means that the whole page has to be rendered again. Another option is to use SVG (Scaleable Vector Graphics)[3], which is done by adding elements to the SVG object. When elements are added to the SVG object, they are also added as elements to the webpage. Frequent DOM updates are very slow because the position of every element has to be recalculated. When visualizing graphs and other data structures both SVG and Canvas[4] are good options. However, since this project requires the user to interact with the data structures in real-time, the canvas element was selected. The canvas element exposes a drawing API (Application Programming Interface), which lets the developer display things on it. Text, lines and other simple shapes can be drawn. The disadvantage of using canvas over SVG is scaling graphics. The user should be able to zoom in and out while modifying a graph. Compared to how this can be achieved with the canvas, the SVG solution is simple.

2.2 Node.js

JavaScript has traditionally been a language which could only be used in web browsers. This means that client code had to be in JavaScript and server code had to be in another language. When working on the same project, it is often easier to only use one language. Node.js[5] is a JavaScript runtime which makes it possible to run JavaScript code on both client and server side. Because the code does not run in a browser, it is possible to access the file system and the operating system of the machine running the application. Node.js comes with a command line tool called npm[6] (node package manager) which can be used to install packages in your project. Packages are code which other developers have written and shared. This feature makes it easier to not "reinvent the wheel" when creating your application. There is only one viable alternative to Node.js, which is Deno[7]. Deno is made by the same person as Node.js but is supposed to be more secure. One of the ways of achieving this is by removing the npm tool, which is the reason why Node.js was chosen over Deno.

2.3 Express

A simple web server which only returns a HTML file is fine if the application only contains static files, but if it is more complex, a web framework should be used. Express[8] is a web framework for Node.js. By default, it does not come with many features. It is possible to define handlers for different URLs, and it can start a HTTP server, but its strength is middleware functions. Middleware functions are functions which does something with either the request or response objects before the actual URL route handler function sees them. This design means that Express is very modular and it is only needed to include what will be used. This is why

Express was chosen over the simple HTTP server which is available in Node.js. Instead of Express, Meteor[9] could have been used. Meteor was not chosen because it works best with websites where a lot of the content is static, which is not true for this application. It also lacks good support for relational databases which is something this application uses.

2.4 Compression

When a user connects to the webpage, the browser requests the HTML file from the server. If the request is valid, it will send back the HTML file. When it is rendered, it will request JS files and CSS files when it is needed. This is no problem when the files are small or if the user has a good connection, but if the file size gets larger or the user is not connected on a fast connection. The user will have a bad experience. This can sometimes be solved by using compression, and in this case, all files sent to the client is compressed with the compression package[10] using GZIP compression.[11] As long as the application is divided up and only data that is required is sent, the compression package helps with the file size on the files that the user request, and in our case, it reduced the size of files up to 80%.

2.5 Connect History Api Fallback

The website in this project is built as a single page application, which means that there is only one HTML file, and extra content is loaded in using JavaScript when needed. Using the HTML 5 history object[12] the user can be given the illusion that they are visiting other pages on the site. E.g. the URL will show www.site.com/contact instead of www.site.com/index.html. If a user enters the URL www.site.com/contact in their browser, the web server will try to return a file to them called "contact". This file does not exist, and the user will get a 404 error. Connect-history-api-fallback[13] is an Express middleware function which redirects the user request through the HTML file which results in the user seeing the page they expect.

2.6 Cookie Parser

HTTP does not store any state about who the user sending a request is. A problem resulting from this is that it is not possible to check whether the user is logged in or not. To solve this, cookies can be included in the header field of the HTTP request. In Express the cookies can be accessed by parsing the header field of the request. To make this more straightforward for the developer, a package called cookie-parser[14] can be used instead. cookie-parser is a middleware function which retrieves the cookies from the request and places them in the req object of the request handler.

2.7 Mocha

When writing software, bugs are often introduced without the developer noticing. As a program grows, it gets very time consuming to test every possible execution path manually. To solve this, tests are made. There are three different types of tests, unit testing, integration testing, and system testing. Unit tests are used to test smaller pieces of code which do not rely on

other parts of the application. Integration tests are made to test if different subsystems of the application work together in an expected way. System tests are made to test if the system as a whole works as intended. Mocha[15] is a testing framework which provides excellent support for testing asynchronous code. Because web development is often asynchronous, Mocha was chosen for writing unit and integration tests. System tests cannot be written in Mocha because there is no way to open a browser and simulate user interaction. Other options which were considered are QUnit[16] and Jasmine[17]. Both of them do not provide the same support for testing asynchronous code as Mocha and were therefore not selected.

2.8 Dotenv

When working on an application which talks to other systems, sensitive information about how to access the other systems are often needed. This can be database login information, or API access keys. This is information that should not be shared with other people, and it should not be included in source control. The sensitive information is often stored in environment variables on the system running the application. In Node.js, environment variables can be accessed from the process.env global object. The variables are passed into this object when the program starts. This is good because then the values are not stored in the source code itself. However, this also means that before running the application the variables have to be set every time. Instead, the package Dotenv[18] makes it possible to store these variables in a file. Dotenv will parse the file, and load all the variables into the process.env object.

2.9 ESLint

When working with a language like JavaScript which does not check for errors before running the code, it can be useful to have a tool to detect errors before the program runs. ESLint[19] is a linter for JavaScript. A linter is a tool which analyzes code and warns the programmer about mistakes. When working in a team with several people the code can often get harder to read over time if the developers use different coding styles. To prevent this, it is possible to configure ESLint to also check the coding style. This project is configured to use the Prettier[20] option which means that coding style is enforced.

2.10 PassportJS

PassportJS[21] provides a set of middleware functions for authenticating users using OAuth[22]. It uses strategies to authenticate users correctly. It's used together with the passport-openid-conncet[23] package to let users use their Feide[24] account on the Interaktiv Undervisning site.

2.10.1 OAuth 2.0

OAuth 2.0[25] is the most used authorization protocol today. The OAuth protocol was originally developed for systems to easily and securely get access to user information stored on other systems over the internet.

OAuth 2.0 workflow consists of the client, that is the third party system that wants information about the user, first redirects the user to the authorization server together with wanted scope, in order to authorize the user. After the user has been authorized, the user will be asked whether or not he will consent to allow the client access to data requested. What kind of data and what the client can do with them is determined by the type of scope that is sent with the request. If the user consent, the user will be redirected back to the client page on a callback URL and an authorization code is given to the client. This code is later sent back to the authorization server in order to authorize the access to the user's resource owner. If the authorization code is valid, an access token is given to the client which the server can use to access the user's resource server and obtain the requested data.

Most of the webpage redirects happen on front channels while obtaining and using the authorization token happens on the back channel. This is done in order to make sure that no sensitive data is accidentally intercepted by someone with malicious intent. It also guarantees that even if someone managed to intercept the authorization code sent from the authorization server ahead of time, they still would not be able to use it on the user's resource server. This is because the resource server requires a highly secure tunnel on the server side.

2.10.2 OpenID Connect

The OAuth protocol was never designed for authentication. It was only designed for authorization and permissions over the web. OpenID Connect[26] is a layer that was placed on top of the OAuth 2.0 protocol in order for OAuth to also be able to handle authentication.

OpenID connect allows OAuth 2.0 to store an ID token, which can be used to get information about the user. With this extra user information OAuth 2.0 can be safely be used as an authentication protocol. In this project, a combination of OAuth 2.0 and OpenID Connect was used in order for the web application to be able to authenticate students through Uninett's dataporten[27].

2.11 Socket.IO

Socket.IO[28] is the most commonly used WebSocket[29] for Node.JS, which makes it the most supported and with the most active community of the alternatives. Socket.IO is built using Engine.IO[30] and creates a connection between the server and the client in real time, with bidirectional event-based communication. In this application, multiple users will send updates to the server, and the server will send out various updates to numerous clients in real time. This would have been very complicated and time-consuming to get right with AJAX or regular HTTP requests. When a Socket.IO function is set up, it is easy for the functions only to send the information that is meant for that client. Socket.IO also has a feature called "rooms" where the server can add a connection to a virtual room. Then when a quiz is running, and the server needs only to update clients connected to that quiz. It will broadcast its messages to that room, and then only clients connected to that room will get that message. This makes Socket.IO ideal

for making the server handling more than one quiz at the time. Also, all the socket.IO functions are asynchronous which makes the wait time for a response a lot shorter.

2.12 SQLite3

SQLite[31] is a SQL database engine which does not require a separate server process to operate. Instead, SQLite reads and stores all the data directly to a single stored file. This is one of the reasons why SQLite databases are used in a lot of small- and medium-sized web pages. This is primarily the reason why SQLite was used in this project over other database engines.

By default, if a connection is made to a SQLite database which does not exist. It will automatically create a new database file for the requested database. It does support all SQL features, which makes it both easy and effective to use.

2.13 Vue

Vue[32] is a framework for building single page applications (SPA) which means the output is a single HTML file and a main JavaScript file. Each component and view have their own JavaScript and CSS file. When the user clicks around on the page, the main JavaScript loads in components or views by adding the content to the HTML file. Then the DOM gets updated with the new components. These updates make navigation and interaction on a webpage feel more smooth and responsive. When programming the client side of the application, the HTML, CSS, and JavaScript get their own container inside the Vue file. When compiling a Vue project, the Vue CLI has to be used to make compatible HTML, JavaScript, and CSS files.

2.13.1 Vue CLI

The Vue CLI[33] is used as a developer tool to start, build, manage and deploy a Vue project. When setting up a Vue project, there are some options to how the project is initialized and which plugins to include. Some of the plugins available help with linting, templating the "pages" and setting up Vue features. One of the plugins that was used in this project is the Vue Router.

2.13.2 Vue Router

Vue Router[34] tries to mimic the multipage application, by using routes and views. A route is a mapping from an URL path to a view. Views are Vue components that should be used as a template for a page. This leaves the user with the impression that the single page application acts like a multipage application, but the website feels more smooth and responsive as the webpage does not load in another HTML file. It replaces elements in the DOM using JavaScript.

2.13.3 Vuex

One problem that is common to run into when creating a medium to large size application in Vue, is how to handle variables used by more than one component. This is solved by a plugin called Vuex[35], which makes it possible to store variables in one location and let all components

read and mutate variables. When creating a store, it is possible to split it up into smaller stores and still have all the components be able to access the entire store. A store will mostly consist of a state, mutations, actions, and getters. The state is where all the variables are stored. To retrieve the state getters are used. Getters cannot mutate the state; only retrieve the state. In order to mutate a variable, mutations must be used. This is accomplished in two steps. The first step is to call an action function. This contains the business logic where the function has read access to the state. The action function is usually used for format validation and to call one or multiple mutation functions with the desired state change. The mutation function is the second step and has write access to the state. The mutation functions are where the state change happens. The getters and actions are accessible from all components in the Vue project. Vuex allows the variables to be mutated in a predictable way, making sure that the state is not changed unexpectedly.

2.14 Babel

When making a web application with JavaScript it is important to remember that different browsers support a different set of JavaScript features. Many new JavaScript features are not fully supported by all of the major browsers, e.g. Google Chrome did not have full ECMAScript 6 support for almost two years after the standard was defined. Some browsers also implement specific features which are only available in that browser. These features are often more performant and should preferably be used by the developer. To make it easier for the developer a tool called Babel[36] can be used. Babel parses JavaScript code and makes sure it can run on the specified browsers. If a given browser does not support the code, the Babel tool will replace it with compatible code. In this project, the Babel tool is part of the Vue build process. There are not any alternatives if the goal is to make the code browser compatible. Languages like CoffeeScript and Typescript will compile to JavaScript, but they do not have the same support in Vue as JavaScript does. Since the Vue build process by default includes Babel, it was chosen as the best option for this project.

2.15 Bootstrap

This web application is designed to work on both mobile and desktop devices. The different HTML elements needed to be placed depending on what kind of device the user has. In web development, this is called responsive design. Bootstrap[37] is a web framework for creating responsive websites. It provides CSS classes which can be used to define the layout of the page and behavior of elements in a responsive environment. CSS for styling elements is also provided. This makes it easier for developers with little design experience to create a consistent style across the entire application. The project BootstrapVue[38] has been used because it contains predefined Vue components which are designed in a way such that they interact well with the Vue reactive DOM. There are many alternatives to Bootstrap, two of them are Materialize.css and pure CSS. Materialize.css is very similar to Bootstrap, but it was not selected because it does not provide any clear advantages over Bootstrap, which the developers were already

familiar with. Pure CSS was not picked because a lot of the functionality of Bootstrap would have been recreated, which did not seem like a good use of development time.

2.16 Cypress

The application required some end-to-end tests in order to check the functionality of the website. In this project, Cypress was chosen as the testing tool for these kinds of tests.

Cypress[39], [40] is an open source web testing framework that is primarily used for end-to-end testing. Cypress is designed in a way that allows test-driven development to be used more efficiently in web development. Cypress allows the developer to efficiently write test suites to their website during development in concurrently to coding the application. This makes Cypress scale a lot better compared to most other end-to-end open-source test frameworks.

An alternative to Cypress was using a Selenium based framework. Selenium[41], [42] is an open-source framework used for automation testing on web applications. Selenium was first released in 2004 and is still currently the most used tool for automation testing for web pages. Selenium is not a single tool; rather the framework is built up by numerous Selenium tools that are all merged. Some of these tools include Selenium IDE, Selenium WebDriver, Selenium RC (Remote Control). Selenium has the advantage over Cypress when it comes to supporting multiple programming languages like Java, Python, and C#, supporting more web browsers, and having the option to record the user actions and converting it to test scripts. Unfortunately, this comes at the price of having to install and setup each of the Selenium tools.

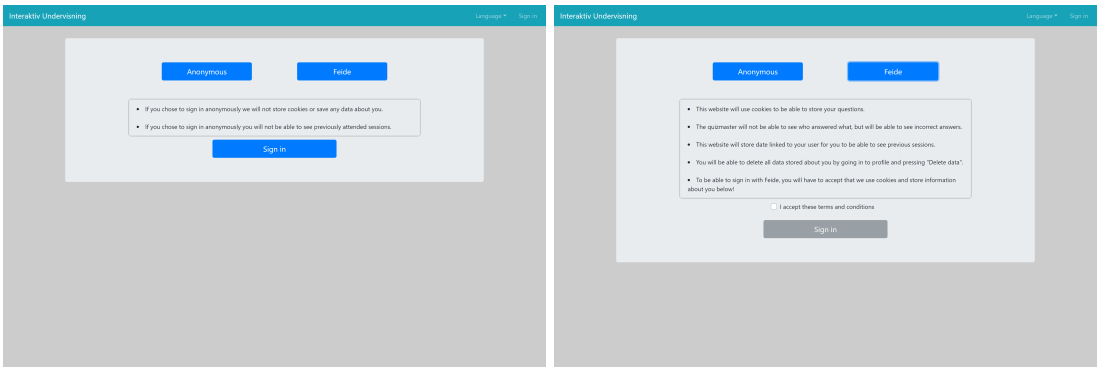
Despite one of the group members having some previous experience using Selenium, Cypress was instead chosen. There were multiple reasons for why Cypress was preferred over Selenium. The first reason was that Cypress only required tests to be written in JavaScript similarly to the Mocha unit test framework, whereas Selenium usually required the use of drivers and conversations between different languages. Since the application was primary developed using JavaScript, the group felt there was not any reason for using Selenium over the fact that it supported multiple languages. The second reason was that Cypress was built to handle modern JavaScript frameworks; this includes supporting Vue, which Selenium seemed to have several well-known issues with. Selenium requires many tools needed to be used together in order to write functional end-to-end tests. Cypress is a tool that has all these tools built into it. This makes Cypress relatively easy to install and get started with. Selenium, on the other hand, required the installation of multiple Selenium licenses tools, libraries, and other testing frameworks to get started. All in all, the advantages of Cypress strongly outweighed the advantages of Selenium for this application.[43]

3 Construction

3.1 Login

When a user visits the website, they need to decide between logging in using their Feide account or continue as an anonymous user. At this stage, their user-rights level is set to 0, and they can only access the login page. Anonymous users have a user-rights level of 1 and can participate in sessions. If the user authenticates with their Feide account, they are assigned a user-rights level depending on their admin status. A regular user with no special rights will get their level set to 2. Level 3 is used for student assistants. Admins (lecturers, professors, ...) are assigned level 4. Users with a user-rights level above 1 can see statistics about sessions they have participated in. Student assistants and admins can create, edit and delete questions and sessions. They can also host a session. Admins can create courses. They are also able to add and remove admins and student assistants to a course. A user with admin or student assistant privileges can apply to get a role in another course. Only admins for that course can accept or decline the request.

3.1.1 Implementation



(a) This is how the login page looks when anonymously is selected.

(b) This is how the login page looks when feide is selected.

Figure 3.1: Login Page

If a user wants to be anonymous, their user-rights level is first set to 1. They are then redirected to the client page. Since it was decided that an anonymous user should not be tracked or have any information stored about them. They are limited to only clicking buttons on the page to navigate. Any attempts to reload the page or going directly to an URL results in the user getting redirected to the 401 error page indicating that the user is not authenticated to view this part of the web app. All answers that are sent in by an anonymous user is stored in the database for the lecturer to view. There is no link between the user and the answer given. Therefore, all answers from anonymous users link to the same user in the database.

If the user clicks on the Feide login button and accepts the terms, a HTTP POST request is sent to /login/feide. When the server receives the request, it is passed on to PassportJS's authenticate function. The authenticate function redirects the user to an external site for

authentication, before redirecting them back to the specified callback URL on the application site. The authenticate function takes an argument telling it where and how to redirect the user; this is called a strategy. The /login/feide route uses the "passport-openid-connect" strategy to connect to Uninett's Dataporten authentication servers. If the user successfully logs in with their Feide account, they are redirected back to /login/callback/feide. This route uses the PassportJS authenticate function to exchange the access code with an access token which is then passed on to the route handler. The handler reads the HTTP request to get information about their Feide account. This information is used to check the database and create an in-memory user object which the server uses to decide what the user is allowed to do on the server. The user is finally redirected to the /client route where they can join a session. The server also sends a cookie for Feide users so that they will stay authenticated for one day, and during this time there is no need for the user to re-authenticate with Feide. When a Feide user signs out, the cookie is deleted. Since the web application is designed to be used on campus at UiS, users are stored in-memory when they are signed in and active on the web application, but this will result in problems if the application is scaled up to more users.

The way the database and the paths are designed it should be easy to add more sign in options for a user, such as Google, Facebook, etc. The way the user right system is designed, only Feide users can be admins or student assistants. When the server first starts the site will have zero admins, but there is a variable in the environment file where admins can be added. This bypasses the normal way to add an admin via the admin site.

3.2 Project Structure

3.2.1 Vue

When creating this project we knew that a framework was needed for both the server and client. For the client, a few frameworks were looked into, such as Vue, Angular[44], React[45], etc. In the end, Vue was picked as it serves our needs better in both functionality and how easy it was to learn. When creating a Vue project, components are created in Vue files. Vue files can include many elements, but in general, it includes HTML, JavaScript code, and CSS. When building a project, the Vue CLI tool generates a single HTML file, JavaScript files and CSS files to a configured build directory. This directory is put inside the public folder on the server. This made it so that when changes are made on the client files it would rebuild and restart the server taking advantage of hot updating the web site while developing. The Vue file is divided into different parts by using tags to let Vue know the difference between HTML, JavaScript, and CSS.

One of the benefits of using Vue is that the application can be divided into smaller files. If code is reused it is possible to create a single file and import it where it is needed. Another benefit of using Vue is that it is possible to watch a variable and re-render parts of the page if it changes. Dividing up the code into smaller files also makes it easier to debug since the code that needs to be looked at is smaller.

Every Vue component uses different features from the framework. The most used features for each component are prop, data, methods, computed and watch. Some of the more rare features are created, mounted and beforeDestroy. Each of these is part of making the web page responsive and reactive when variables changes. Created is a function that runs when the component is first loaded into the DOM, mounted runs just before the HTML in the component is rendered, and beforeDestroy runs before the component is removed from the DOM. The variables that are used in a component can be stored in two places, either in props or data, but they have two different use cases. If the component needs to get a variable from their parent component it should be stored as a props variable. A props variable cannot be changed from the component. If the parent component changes the value, the child component will get the new value. If the value is used in the HTML, the component is rendered again with the updated value. If the value needs to change inside the component the value needs to be stored inside a data property. This variable also triggers updates to the renderer if it is changed, but it has no link with the parent component. Methods are normal functions that can be called by other functions or by HTML elements. Computed methods run once the component is created, and every time a variable inside the function is changed. Watch methods watch a variable or a computed method and trigger a method every time it changes, where the new and old value is passed as arguments to the method.

3.2.2 Vuex

The application's components had some shared variables such as localization and user information. To make sure that all the components accessed the same information, it was stored in a Vuex store. The store contains the following information:

1. `user`. The `user` object stores the user's `username`, `feideId` and `userRights` level.
2. `adminSubjects`, which is a list of all the subjects the current user has rights for.
3. `loggedIn` is a boolean which stores whether or not the user is logged in.
4. `locale` is an object which stores information about the selected language in the `locale` property, and the available languages in the `localeList` property.
5. `courseList` stores all the courses the user has rights for.
6. `selectedCourse` stores the course id of the course the user is currently viewing.
7. `questionTypes` stores every available question type.

3.2.3 Vue Router

Since Vue is a single page application, it does not scale well since it would have to load in all resources even if it did not use them. A solution to this is Vue routing where it is still a single page application, but it will simulate the experience of a multi page application where only the resources needed for the shown content is loaded. When the user goes to a new route

the JavaScript and CSS files needed are loaded. In combination with the Vue CLI tool it is straightforward to set up, and easy to use. If Vue routing is selected when setting up the project, the tool creates a views folder and the config file for the Vue Router. When a new route is desired the only thing that is required is that the developer creates a new Vue file and uses it as a normal component. A new link needs to be created in the Vue Router config file pointing to the Vue component. The standard is to place all routes and views inside the views folder. All the components used by the views should be placed in a separate folder. Even though they are both Vue files and it is not possible to tell the difference by looking at the content inside the files. For every route, a separate JavaScript file is created which is only loaded when the user visits that route.

3.2.4 Server Structure

When setting up the server side, the project was divided into different blocks. Each block has its own responsibility as far as possible. If done correctly there are many benefits by splitting code into different blocks. The standard way of splitting code is to follow SOLID (S: Single responsibility principle, O: Open–closed principle, L: Liskov substitution principle, I: Interface segregation principle, D: Dependency inversion principle). If this is followed correctly it gives the advantages of MURDER (M: Maintainability, U: Understandability, R: Reusability, D: Debugability, E: Extensibility, R: Regression). Examples of this can be seen in how the socket functions are split up into categories. Client functions, admin functions, and Feide functions are split into their own file and then imported where they are needed. The same can be seen in the database functions, where get, insert, update and delete functions are split into their own files to make development easier. When it comes to extensibility, the server is written to be able to scale up with more question types in a straightforward manner. The way the server handles generating a solution, validating a question and checking an answer is written into their own files based on the question type. This makes it so that if another question type is added, the developer only need to add a couple of lines in each master file to make it work. The developer also needs to implement their algorithms in separate files.

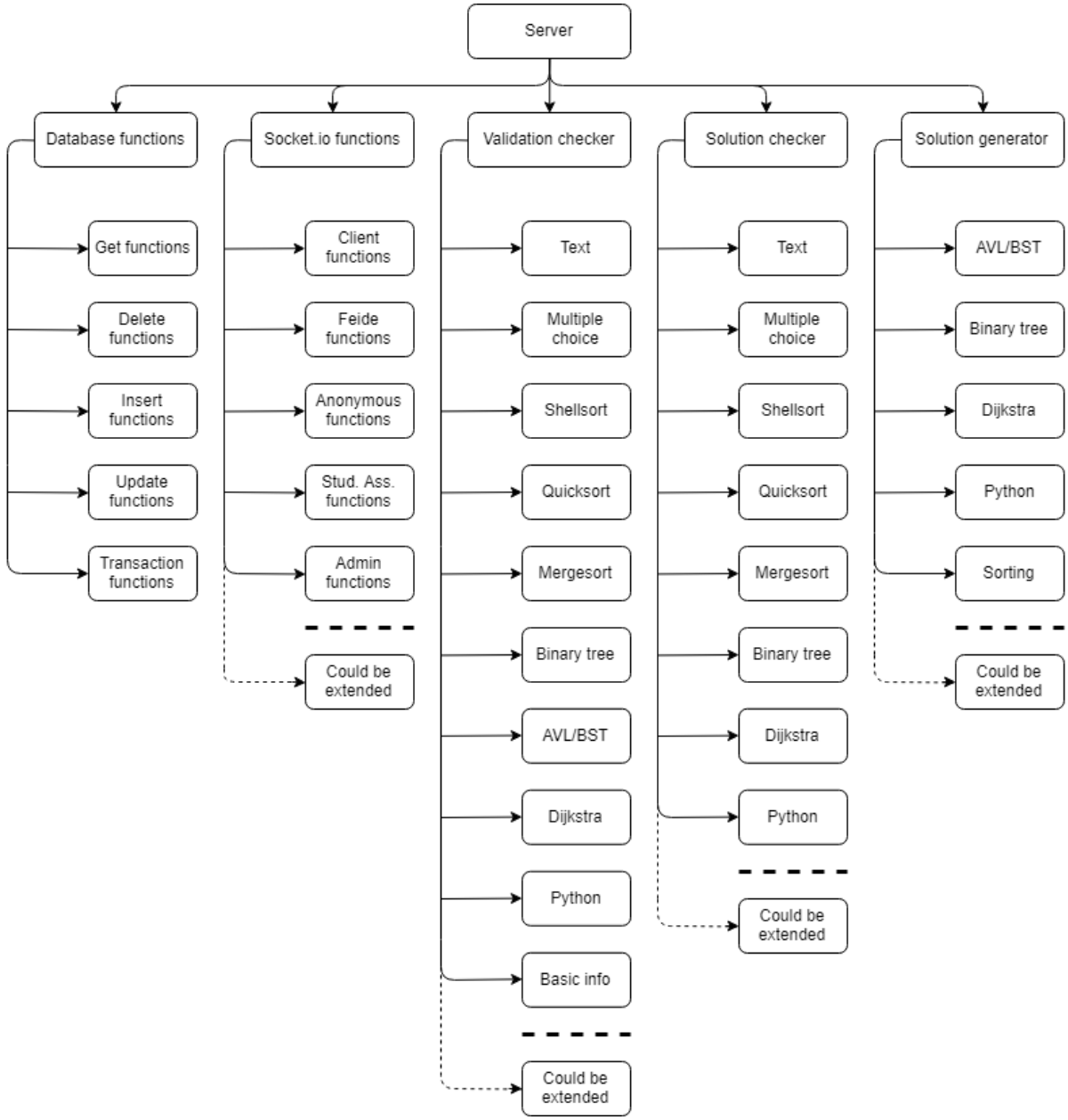


Figure 3.2: This is a diagram displaying the server structure and how the code is divided up into different subfiles

3.3 Sorting

Sorting algorithms are used to reorder elements in a data structure, often arrays or lists. The elements are ordered by a rule depending on what type of element the data structure contains. Numbers are often ordered from the lowest number to the largest. Characters are often ordered by transforming the character into a number, where ”a” is often defined as smaller than ”b.”

To sort the elements an algorithm must be used. They have different performance and memory usage, so it is important to be able to pick the best fitting algorithm depending on the situation. This project contains implementations of insertion sort, shell sort, merge sort and quick sort which are common algorithms. They are written to work on arrays containing any element

which can be compared using the >, < and == operators in JavaScript. Instead of returning the sorted version of an array, they return functions which can be called to navigate between the steps of the algorithm. When checking if a student's answer is correct, it is pointless to check if they managed to sort the array, because they might not have sorted the array according to the rules of the algorithm. The steps are therefore returned so that together with the GraphDrawer, the students can graphically perform the sort. It is then possible to compare each step with what the student did and see if they know how the algorithm works.

3.3.1 Merge Sort

While performing the sort, every stage of the algorithm is saved in a list of steps. There are three stages which are stored as a step.

- Initial. The initial step is stored before the sorting starts. It contains a copy of the unsorted array and a reference to the limit.
- Split. The split step is added when an array is split into two arrays. It contains a copy of the array being split, and copies of the new arrays.
- Merge. The merge step is added after two arrays have been merged to a new array containing the sorted version of all the elements. It contains a copy of the two arrays being merged and a copy of the resulting array.

When implementing the algorithm, performance and memory usage were not considered to be important. The algorithm runs only once per question to generate the steps so that each student's answer can be compared to the right way of performing a merge sort. For this reason, the intermediate arrays are allocated dynamically instead of being reused in later steps. This gives the algorithm worse performance, but its average case performance is still $O(n * logn)$. Memory usage has not been considered because the algorithm has to store more information than normal to store the steps.

3.3.2 Quick Sort

While performing the sort the algorithm, it also stores the state in steps. This is done for the solution checker to be able to check if the student has drawn the correct answer when simulating the quicksort algorithm.

- Initial: The initial step is stored before the sorting starts. It contains a copy of the unsorted array.
- Split: The split step is stored each time the algorithm splits a list; it stores the unsorted list, the pivot point, left and right list.
- Merge: The merge step is stored each time the algorithm merges the sorted lists and pivot point. It stores information about the sorted left and right list, the pivot point and the sorted list after the merge.

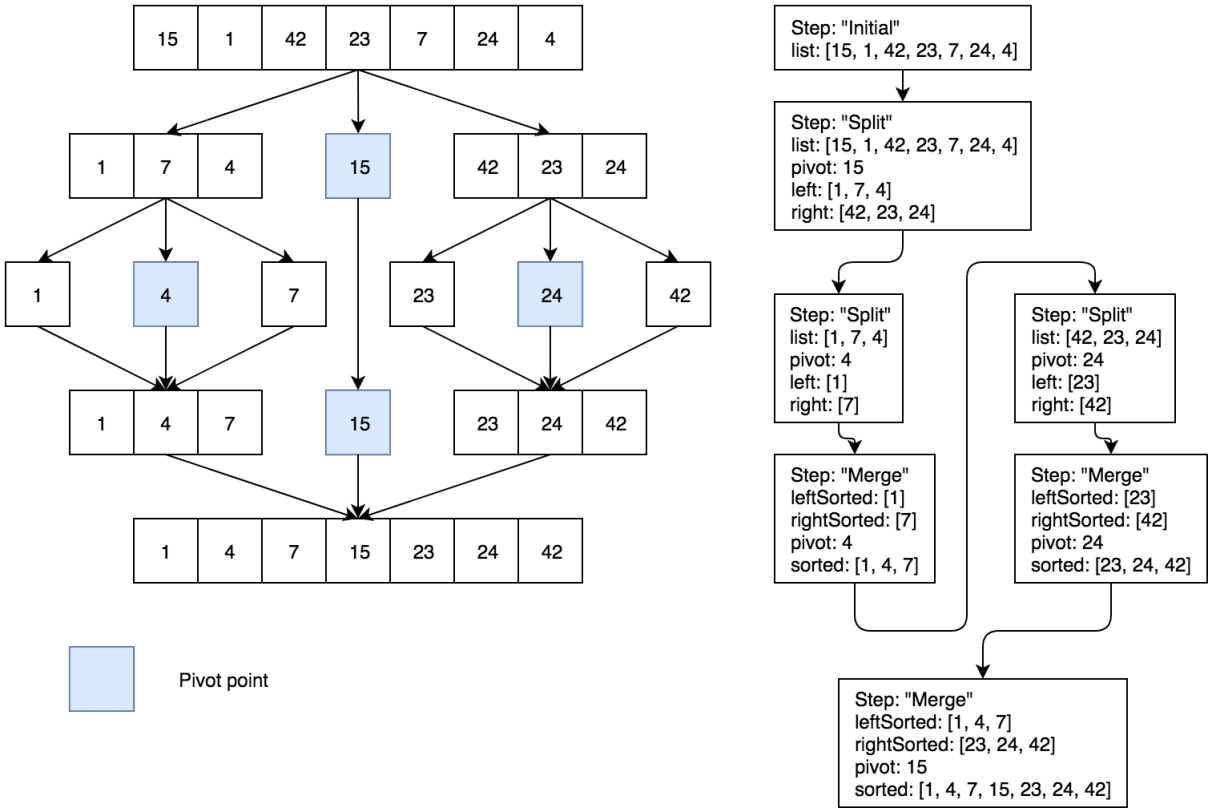


Figure 3.3: Quicksort

3.4 Data Structure

3.4.1 BinaryTree

Most of the student tasks involving binary trees usually involve drawing a resulting binary tree. In order to avoid giving the students help with drawing a correct binary tree structure, the GraphDrawer was designed in a way that lets them mix between drawing tree structures and graph structures. However, this meant that multiple extra criteria needed to be considered when checking the students work with the solution. It was necessary to implement a JavaScript class, `Tree`, for representing the binary trees drawn and `BinaryTreeNode`, representing every node in the tree. These classes were designed in a way such that they can be used in all the taught binary tree structures in the Algorithms and Data Structures course. These are the standard binary tree, binary search tree (BST), and AVL tree.

The `Tree` class consists of a root node referencing the node at the start of the tree, and an array of all the current nodes in the tree, this also includes the root node. The `BinaryTreeNode` class consists of the node's value and an array containing the node's child nodes. A notable design choice was that each binary node and all their children nodes are stored in arrays. The tree has an array of all current nodes in the tree, where index 0 is the root node. In addition, all nodes have an array containing references to its children nodes. Normally a tree structure would have only needed to have an independent variable referencing the two different child nodes. However, because students can draw any tree, it was a better option to store the children references in

arrays instead. The reason for this was because a student had the option to draw a tree with more than two children, which is not allowed in a binary tree structure. Arrays also do not cause any problems with distinguishing the different child nodes either since the nodes will be divided properly using different placement indexes. A left child node will always be located at index 0 and a right child node will always be located at index 1. If there are more than 2 children nodes in the array, then the binary tree criteria are not fulfilled, and therefore will not be accepted as a valid tree object. Multiple functions were implemented to not only transform the drawn tree to a Javascript tree object, but also to check whether or not the created Javascript object is qualified as the chosen binary tree structure. Additional inbuilt class functions for both the tree class and node class were needed for certain tree interactions. For instance, when storing the current tree as a step, a function for duplicating the tree was implemented.

Because the binary tree has very few restrictions, creating a solution tree object for the data structure proved challenging. Unlike the BST and AVL tree structures, the binary tree only required each node to have a single parent and have no more than two children. Therefore the order of the nodes in the tree does not matter. The solution checker for the binary tree was therefore set up so that it checked the nodes in the students binary tree and compared it to the list of nodes demanded in the question. If the student delivered a binary tree that either did not have the required nodes or broke any of the binary tree conditions, then the answer would be incorrect. In any other case, the student will get the answer correct. This means that the solution object stored in the database is an array of the nodes wanted in the binary tree.

3.4.2 Binary Search Tree

The solution checker is designed to check that the tree object created from the students drawing, matches the tree created in the solution. The solution checker recursively calls a function named `checkNode` that compares each node in the tree with the corresponding node in the solution. If the two nodes do not match, the solution checker returns false, and the student has answered incorrectly. If all the compared nodes match the solution, the solution checker returns true, meaning that the student has answered the question correctly. The solution checker traverses the tree inorder. It will first check the left node, then the parent, then the right node. This solution checker is also used for AVL trees.

A function `createBinarySearchTreeSolution` was implemented to create a BST based on given arguments and/or existing tree object. This function was implemented so that the admin did not need to manually create the solution trees for each question. The function takes an array with integers and possibly an existing tree as parameters. If the existing tree is not given, the function creates a new tree based on the values in the array. If an existing tree is given, the values in the array are added to this tree instead. It is important to note that the function will not work with duplicate values in the array or the given tree.

The `createBinarySearchTreeSolution` also needed to create resulting trees upon deleting an existing node in a tree. Because there are multiple ways to choose nodes for replacements when

deleting a node with two children. It was required for the `createBinarySearchTreeSolution` to create a list of the potential final resulting trees. It is required to specify whether the user wants to add nodes or remove nodes from the tree when creating the solution trees. The function cannot switch between the two during execution, and an existing tree is required in order to remove nodes in the tree.

The `createBinarySearchTreeSolution` returns an array containing the different states of the BST during its creation. Since there can be different resulting trees when deleting a parent node, all resulting trees are stored in an array. The last element in the array is the finished BST tree object that is compared with the drawn tree from the students. Originally the drawn tree from the student is a graph object and has to be transformed into a `Tree` class object using the general tree function `createTreeObjectFromCanvasObjectver1`.Once transformed into a BST object the student tree is compared with all the trees in the solution object in the solution checker.

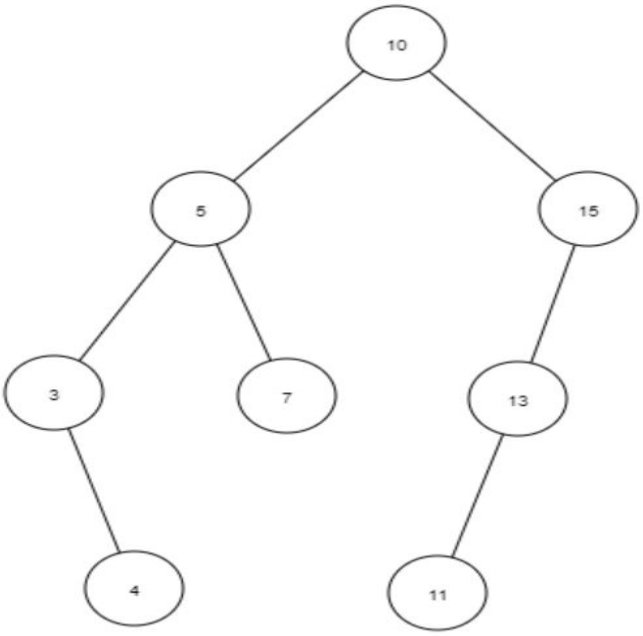


Figure 3.4: The figure shows an example of a Binary Search Tree.

3.4.3 AVL

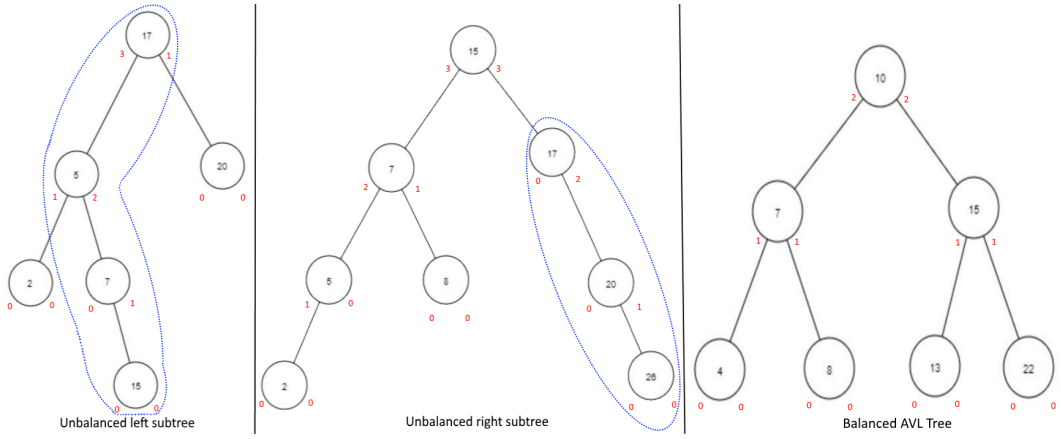


Figure 3.5: The figure displays 2 unbalanced binary search trees and a fully balanced AVL tree.

An example of balanced and unbalanced BSTs can be seen in figure 3.5. The left tree is a BST that has too many nodes in the left subtree. The center tree is a BST that has too many nodes in the right subtree. The tree to the right in the figure is a fully balanced BST, which makes it qualified as an AVL tree. The red letters represent the height of the node's children. The value is determined by how many nodes there are between the leaf nodes to the selected node. The blue dotted lines mark the subtree that breaks the AVL condition.

In this project, the name of the rotation direction is based on which subtree side has the most nodes. This means that in the previous example 3.5, the middle tree, a right rotation was needed that would have rotated the right subtree one step to the left. The opposite happens in a left rotation; too many nodes in the left subtree will make it rotate in the right direction. In some circumstances, a double rotation is needed. These, on the other hand, are named after the last direction the nodes need to rotate in order to balance the subtree. Double left rotation means that a rotation towards the right is first used on the right child of the node causing the imbalance. This switches the right child's spot with the imbalanced node. Then followed by a rotation on the new imbalanced node towards the left. The opposite will occur on a double right rotation.

The function `createAVLTreeSolution` creates an AVL tree used as the solution object for a question. The functionality is mostly the same as the `createBinarySearchSolution` function mentioned earlier. However, some extra functionality was needed in order to balance the tree. If an existing tree is given, the given tree needs to be fully balanced before any insertions or deletions can be done to the tree. Once the existing tree is balanced or no existing tree was given, the values in the added array are inserted in the tree. Just like in `createBinarySearchSolution`, removing nodes from a tree requires that an existing tree is given. After every insertion and deletion, the tree will once again have to be checked to see whether or not it is still balanced. If the tree is not balanced, the tree will have to rotate in order to rebalance itself. The rotation is determined by which subtree currently has the most nodes. Not to mention insertion and

deletion require a different set up when it comes to using normal rotation or double rotations. The returning array from the `createAVLTreeSolution` is used and has the same format as in the `createBinarySearchSolution` function. The only noticeable difference is that the array for the `createAVLTreeSolution` also need to store the tree state after any rotations have been performed.

Because the `Tree` class contained circular references between a parent node and child nodes, it proved difficult to store solution BSTs and AVL trees into the database. This was because it was not possible to convert the tree object to JSON, since JSON cannot handle circular references. This means that the circular references had to be removed before the solution could be stored in the database. To solve this an export function was implemented for the `Tree` class. The export function removes the circular references by turning the parent reference into the parent's node value. Once the parent reference is gone, the tree object was able to be converted to JSON format. Of course, since the solution checker requires BSTs and AVL trees to be in a certain format, an import function also needed to be used to convert the tree object back to its original form. The import function has two tasks. The first task is to replace all null values with an undefined value. The second job is to reinsert the proper parent reference to each node in the tree. This is simply done by traversing the tree and remembering the previously visited node as the current node's parent. Once this is done the solution object can be checked with the converted tree the students drew with the GraphDrawer and see if they match or not. If the question revolved around removing nodes, then the converted tree only needs to match one of the trees in the solution object.

3.5 GraphDrawer

To solve the problem of letting students visually answer questions, the GraphDrawer Vue component was created. GraphDrawer can be included on any page where data structures should be displayed. It is built as a layer on top of the HTML5 Canvas element.

The GraphDrawer includes a rendering engine which is responsible for converting a part of the world to an image displayed to the user. The world is where every node and edge is placed before they can be rendered. A rendering engine is a type of software which generates graphics from data. The world contains everything the user can interact with. The interaction is done through the HTML5 Canvas element. User interaction events are received from the canvas element in the form of "The user clicked at this (x, y) position," or "The user currently has two fingers on the screen at (x1, y1) and (x2, y2)". The GraphDrawer needs to convert the screen coordinates to world coordinates, and then decide if the given event at the given world position has any meaning for the state of the world. The coordinate conversion is done by the camera.

The world consists of two types of entities, nodes (vertices) and edges. Entities are stored in arrays on the GraphDrawer object. Nodes are drawn as either a circle or a square. A node needs to specify its position in the world, which is stored on the node object as the properties `x` and `y`. It also needs to store information about how large it is, which is stored in the `w` and

`h` properties. If the node is a circle, then `w` and `h` should have the same value, and the value should be the circle radius. If it is an ellipse, then the `w` is the radius along the x-axis, and the `h` is the radius along the y-axis. A node needs a value, stored in the `v` property. This can be text, a number, or a combination of text and numbers. GraphDrawer will only render nodes if they have the `x`, `y`, `w`, `h` and `v` properties. There are also several optional properties a node can have. `selected` is a boolean used by a controller to indicate whether a node has been selected (`true`) or not (`false`). `culled` is a boolean used by the camera to indicate whether a node should be drawn (`false`) or not (`true`). `fillColor` decides what color is used to render the inner part of the node. If it is undefined, white is used. `strokeColor` decides what color will be used to render the outline of a node. If it is undefined, black is used.

Edges are used to link nodes together. An edge needs to have two properties, `n1` and `n2`, which are references to the nodes being linked by the edge. Edges are drawn as either a line or an arrow, between the two nodes. If the edge has the property `directed` with the value `true`, an arrow is drawn pointing from `n1` at `n2`. If `directed` is `false` or `undefined`, a line is drawn. The `strokeColor` property decides which color will be used to draw the line/arrow. If it is undefined, black is used. An edge can also have the `v` property. `v` represents the cost of going from `n1` to `n2` (and from `n2` to `n1` if `directed` is `undefined` or `false`). `v` should be a number, and its value is drawn next to the line/arrow.

To render the world, three objects are needed: the world, a camera, and a HTML5 canvas. The world has information about the entities. The camera is used to decide which entities need to be drawn to the canvas. Two additional canvases are created, such that drawing the world is easier. The drawing buffer `drawBuffer` and the static buffer `staticBuffer`. The drawing buffer size is five times greater than the canvas, and the static buffer is the same size as the canvas. These objects are needed because the world might be larger than what is possible to display on a single webpage. The canvas decides how much of the webpage is used to render the world. The camera can then be moved around to display different parts of the world. When the GraphDrawer starts, an interval is started which lets the GraphDrawer update 60 times every second. Each time the GraphDrawer has the opportunity to update is called a frame. Because the application is designed to be used by both phones and computers, computationally expensive and time-consuming operations should be avoided if possible. For this reason, the GraphDrawer will often discard frames. Discarding frames means that no operation is performed, even though the opportunity to do something was there. The update function is called once per frame. If the `dirty` property is `false`, the frame is discarded. The GraphDrawer will only set the `dirty` property to `true` once, at startup, so the initial state is rendered. Afterward, a controller has to tell the GraphDrawer to not discard the next frame by setting `dirty` to `true`. The `dirty` property makes it possible to have a responsive user interface, and save CPU time at the same time. When a frame is not discarded, the world is redrawn. This is done in several steps:

1. The canvas is cleared. This is done to prevent entities from appearing at two positions at the same time.

2. Nodes and edges are drawn to a drawing buffer (`drawBuffer`). This is done by the GraphDrawer.

3. The user interface is drawn to a separate buffer (`staticBuffer`). This is done by the GraphDrawer if the `operatingMode` is set to `Presentation`, or by a controller if the `operatingMode` is set to `Interactive`. `operatingMode` is used by the GraphDrawer and by controllers to decide what kind of user interaction is allowed.

4. `drawBuffer` and `staticBuffer` are drawn to the canvas.

Some users of the site could be using old hardware or somehow restrict the performance of their browser. The GraphDrawer is designed to update 60 times per second. This means that every frame is allocated: $1000/60 = 16.666...$ milliseconds to do the work. If the frame uses more than the allocated time, the user will experience this by having the program not respond for some time. If this happens, then the next frame is discarded, to give the computer time to catch up again. 60 frames per second were picked to balance how smooth the graphics move, and the program performance. If more than ten frames need more than the allocated time, the number of frames per second is lowered by 25%. By lowering the number of frames per second, the user will still have a responsive program. The movement of the graphics looks normal unless the time per frame is increased significantly.

The drawing buffer is used to prevent screen tearing. Screen tearing happens when the world is updated in the middle of drawing the world to the canvas. The result is that some parts of the canvas show the old state of the world, while other parts show the new state. By using a separate buffer for drawing, the user will be shown a complete image of the world at every frame, instead of one which is in the process of being drawn. The user interface uses another buffer because it makes it easier to draw the interface on top of the world, by simply placing the draw statement after the world draw statement. The `staticBuffer` uses canvas coordinates to draw, which needs fewer operations to consistently place the interface inside the camera view.

To let the user interact with the world, two event listeners are attached to the canvas. `mousedown` is used to respond to mouse clicks, and `touchstart` is used by touch screens. These listeners use the same callback function. The callback function passes the event to the right handler, depending on handler priority. A handler is a function which takes the event as an argument and returns whether or not it consumed the event. If the event was consumed, it should not be given to any other handler. The priority of the user interface handler should generally be higher than the world handler, but the order is not forced. The GraphDrawer first checks if the user interacted with one of the buttons. If not, then a controller is given the chance to handle the event. Finally, the event is given to the gesture handlers. Gesture handlers are responsible for detecting patterns in user interaction and check if they match a given gesture. The GraphDrawer implements two gestures: pan and zoom. Pan is used to move the camera around the world. A pan gesture happens when the user clicks a mouse button and moves the pointer without releasing the button. This can also happen if the user drags their finger across the canvas. The movement must be larger than a given threshold before the camera starts

moving. A zoom gesture is defined as the user dragging two fingers in opposite directions. If the fingers move towards each other, the zoom decreases. If the fingers move away from each other, the zoom increases. The average point between the fingers will remain at the same position relative to the canvas edge. The position of events is stored in different properties of the event object, depending on what type of event it is. To make it simpler to write code that works with all types of events, a helper function `setEventOffset` was created to convert event positions to a common format. This function should be used right after receiving an event object, to be sure the position of the event is available in the expected properties.

The GraphDrawer constructor takes four arguments, `canvas`, `locale`, `config`, and `window`. `canvas` should be a reference to the HTML Canvas which will be used. `locale` should be an object containing references to all of the displayed strings in a given language. `config` should be an object containing configuration options. `window` should be a reference to the browser window object. The following is a list of all the configuration options available:

1. `nodeShape`, determines the shape which is used to render the nodes. Possible values are `"Circle"` and `"Rectangle"`. If no value is given, `nodeShape` is set to `"Rectangle"`.

2. `controlType`, determines which controller is used by the GraphDrawer. Possible values are the names of the controllers as a string. If no value is given, `controlType` is set to `"Sort"`.

3. `operatingMode`, determines if the user can manipulate the world, or view a list of states. Possible values are `"Interactive"` and `"Presentation"`. If no value is given, `operatingMode` is set to `"Interactive"`.

4. `displayEdgeValues`, enables the `v` (value) property of edges. The value will be drawn next to the edge and can be modified if the `operatingMode` is set to `"Interactive"`. If no value is given, `displayEdgeValues` is set to `true`.

5. `directedEdges`, enables drawing of edges as arrows instead of lines. If it is set to `true` then all edges which does not have the `directed` property set to `false` are drawn as an arrow instead. If no value is given, `directedEdges` is set to `false`.

To determine the position and size of a canvas, it is not enough to define the size in the CSS. If the size defined by the CSS is different from the width and height properties of the canvas, the image will be stretched to fit the size defined by the CSS. Because the application is going to be used by screens of any size, the canvas size needs to scale with the screen size. To achieve this, the update function will check if the width of the canvas matches the `clientWidth` property. The `clientWidth` property is how much space the canvas has been given on the screen. If they do not match, then the size properties of the canvas need to be changed. The width is set to the same value as `clientWidth`. The height is determined by the screen aspect ratio and the canvas width. By using the aspect ratio, the canvas has a greater height than width on mobile devices, and a smaller height on computers.

When the size of the canvas is changed, the size of the `drawBuffer` and `staticBuffer` also need to change. Because they are also canvases, this is simply done by updating the width and height properties. After the width or height property of a canvas is changed, any data drawn to it is cleared. The `drawBuffer` can be fixed by setting the `dirty` value to `true`, because this will make the GraphDrawer redraw the world. The `staticBuffer` is usually not redrawn every frame, e.g., the buttons to navigate between algorithm steps, and is not part of the draw function. To redraw the `staticBuffer`, every object on it needs to get a new position and be redrawn right after changing the canvas size. The camera keeps a reference to the size of the canvas in its `viewportHeight` and `viewportWidth` properties. If these are not changed, the resulting image is stretched.

After resizing the canvas, the `drawBuffer` can be too large, if the user has a screen with a very large resolution, e.g., above 4K. Browsers have different limits, but based on numbers from Mozilla [46] and Chromium [47], modern browsers have a width and height limit of 32767 pixels for the canvas element. If this limit is exceeded then depending on the browser, the canvas can stop working, the site can stop working, or the browser can crash. The actual limit seems to be much lower than 32767 pixels. The canvas slowed down significantly when the width was larger than 7000 pixels, and completely stopped working when the width was larger than 8500 pixels. The website was viewed on a modern version (v73.0.3683.103) of the Chrome browser. To be sure that the size of the canvas won't prevent the GraphDrawer from working, a limit of 7000 pixels has been implemented.

The position of an arrow from one node to another node depends on the shape of the node it is pointing at. If the node is a circle, then the arrow is drawn at the circumference. This is done by creating a line between the two nodes and checking where it intersects the circle. Rectangle nodes have four defined positions for the arrow, one at the center of every edge. To figure out which point to use, the angle (in degrees) between the nodes is used. The angle is calculated from the x-axis and increases counterclockwise. The edge is picked based on the following criteria:

1. If the angle is between 45 and 135 degrees, the point at the top edge is picked.
2. If the angle is between 135 and 225 degrees, the point at the left edge is picked.
3. If the angle is between 225 and 315 degrees, the point at the bottom edge is picked.
4. If no edge has been picked, the point at the right edge is picked.

Developers can add or remove features from the GraphDrawer by creating controller objects. Controller objects are used to modify the user interaction and the world presentation. They can interrupt the GraphDrawer at pre-determined states. It is also possible to create custom interrupts by creating timers (`setTimeout`, `setInterval`) or by adding event handlers to the canvas object. The controller interface is simple by design and only requires two methods. There are also some optional methods which can be implemented to add more functionality.

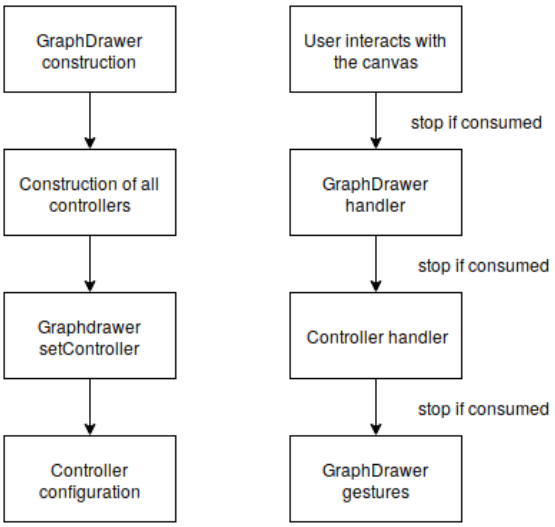


Figure 3.6: Graphdrawer - controller interaction

1. Required: The controller constructor. The constructor should take two arguments. The first being a reference to the GraphDrawer, and the second being an optional configuration object.

2. Required: `export()`, should return an object. The `export` method is called when the system wants to retrieve an answer or solution from the GraphDrawer. The returned object should be a representation of the world, which can be understood by one of the algorithm implementations, or by a solution checker function.

3. Required: `mouseDownHandler(e)`, should return `true` or `false`. The `mouseDownHandler()` is called when the user clicks or touches the canvas. It is called with the event as the first and only argument. `mouseDownHandler()` should return `true` if the event was handled and should be consumed, or `false` if it was ignored.

4. Optional: `configure()`. If the controller was given a configuration object at instantiation, then the `configure()` method should be implemented. It is called when the GraphDrawer changes controller.

5. Optional: `dirtyUpdate()`. `dirtyUpdate()` is called right before the world is rendered and drawn to the screen.

6. Optional: `parseSteps()` and the `steps` property. If the `operationMode` of the GraphDrawer is set to `"Presentation"`, then the controller is expected to have a list of steps in the `steps` property of the controller. The user can navigate through the list. `parseSteps()` is called every time the user changes the viewed step. The index of the viewed step is stored in the GraphDrawer property `currentStep`.

7. Optional: `onCanvasResize()`. Anything drawn by a controller to the `staticBuffer` is cleared when the canvas is resized. If the controller needs to redraw when the canvas

size changes, the `onCanvasResize()` function can be implemented. It is called when the canvas has a new and valid size.

Even though the controllers allows for configuration, not all of the configuration can be changed when the GraphDrawer component is included. E.g., Dijkstra start and end colors are defined inside the component, and not as a property on the component. This is done to enforce a consistent style in the application. If it were possible to change the colors every time the GraphDrawer was included on a page, it would be confusing for the users. The colors can be changed in the component definition, but they have to be the same for the entire application.

3.5.1 Camera

The camera object is responsible for letting the user view only a section of the world at a time. It is implemented in a way that only works for two dimensions. It is also not possible to rotate the camera. This is done to prevent unnecessary complex mathematical expressions from being used. The camera object has a position given by the `centerX` and `centerY` properties. Depending on the canvas size and the `zoomLevel` a section of the world will be rendered. To determine if a node or edge should be drawn, the `cull(object, isNode)` method is used. It returns `false` if the object is inside the camera view, or `true` if the object is outside. A culled object should not be drawn. A node is culled if it is outside the camera view. An edge is culled if both nodes are outside the camera view.

The camera uses the `project(x, y)` function to convert screen coordinates to world coordinates. The camera has a position in the world and also has access to the size of the canvas where user interaction happens. The conversion can, therefore, be done by checking what percentage of the screen width the interaction happened at and then adding the same percentage of the camera size along that axis to the position of the camera. This works for both axes. If the camera is positioned with its left side at $x = 50$, and the user clicks in the middle (50%) of the canvas, then the world position would be $50 + \texttt{camera\_width} * 0.5$. To convert from world to screen coordinates, the `unproject(x, y)` function should be used. This uses the same logic for converting, but in the opposite direction.

3.5.2 Graph0

Graph0 is the controller which is responsible for making it possible to work with graphs and trees. It can be used for both data structures because a tree is a graph with some restrictions. Graph0 can also be used to create a Dijkstra task by drawing a graph and marking the start and end nodes.

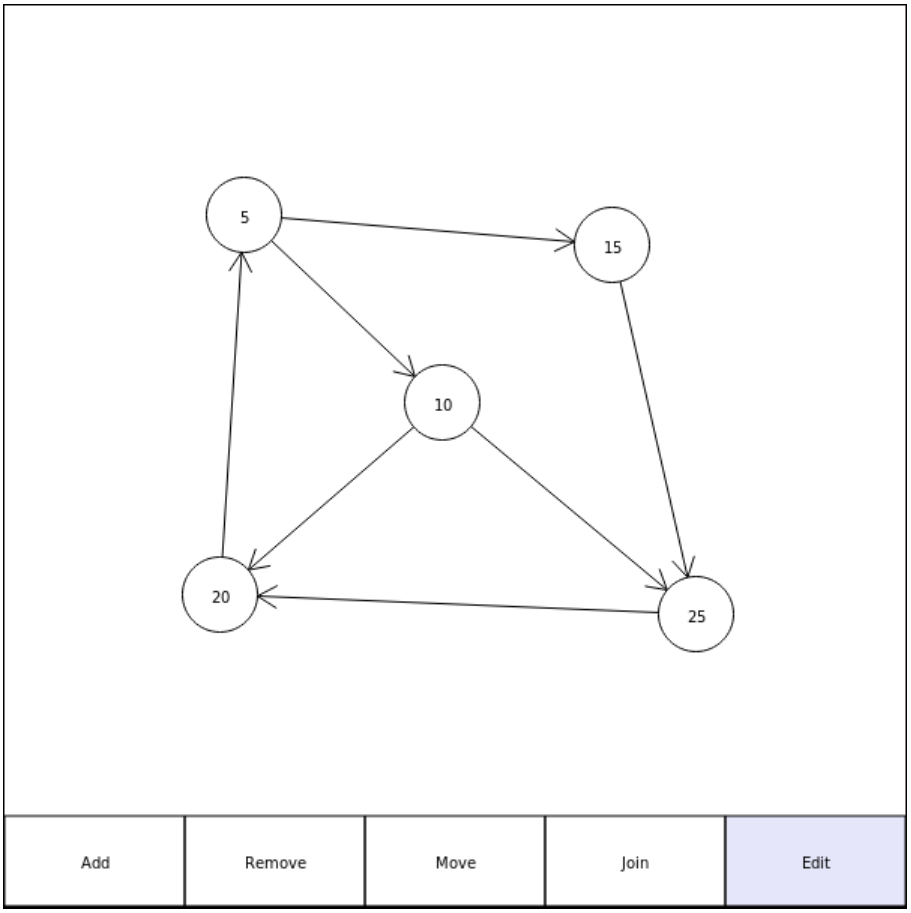


Figure 3.7: Graph0 - user interface

The Graph0 constructor takes two arguments. The first should be a reference to the GraphDrawer object. The second is an optional configuration object. The following properties can be set from values found in the object.

1. `exportType`, determines if the drawn graph should be exported as a tree or as a graph. Possible values are `"Graph"` and `"Tree"`. If no value is given, `exportType` is set to `"Graph"`.

2. `steps`, contains information about a graph or tree which Graph0 should load. This is typically the solution of a task, which should be displayed step by step.

3. `importType`, determines what kind of object `steps` contains. It can either be `"Graph"` or `"Tree"`.

4. `importSource`, if the `importType` is `"Tree"` then the `importSource` property needs to have information about where the tree originated from. If it came from one of the algoritms, it should have the value `"Algorithm"`. If a user drew the tree, then it needs to be `"Student"` instead.

5. `subType`, determines what kind of algorithm to use. Possible values are `"Dijkstra"`. If no value is given, `subType` is set to `undefined` and no algorithm is used.

6. `startNodeColor`, determines which fill color the start node is marked with. If no value is given, `startNodeColor` is set to a light green color.

7. `endNodeColor`, determines which fill color the end node is marked with. If no value is given, `endNodeColor` is set to a light red color.

The Graph0 controller defines five different states, Add, Remove, Move, Join and Edit. Each of them has their own event handler defined by a method in the class. There is also a Mark state which is only available to the user if the `subType` is set to `"Dijkstra"`.

1. Add. The Add state lets the user add a new node to the graph. The new node is placed at the position of the interaction event. The value of the node is set to a letter if `subType` is set to `"Dijkstra"`, or 0 if `subType` is undefined. The Add state always consumes the event.

2. Remove. The Remove state lets the user remove a node or an edge from the graph. If the interaction event happened inside of a node or close to an edge, it is removed, and the event is consumed. If nothing was removed, the event is not consumed. If a node is removed, then any edge connecting the node to another node is also removed.

3. Move. The Move state lets the user move a node. This is done using a drag and drop motion. When the interaction starts, a reference to the node under the interaction event is stored. When new events are received, the position of the node is updated to match the position of the event. The event is consumed only if the first event happened inside a node.

4. Join. The Join state lets the user create an edge between two nodes. This is done using a drag and drop motion. When the interaction starts, a reference to the node under the interaction event is stored. When the interaction ends, the event handler checks if the is a different node under the event. If another node is found, an edge between this node and the first node is created. The event is consumed only if the first event happened inside a node.

5. Edit. The Edit state lets the user edit the value of a node or an edge. The event handler starts by checking if there is a node under the interaction event. If a node is found, the user can edit the value. If no node is found, the handler checks if the event is close to an edge. If it is, the value of the edge can be edited. The event is consumed if the user was given the option to edit a value.

6. Mark. The Mark state lets the user mark nodes as either a start or end node. If a node is found under the interaction event, its marked value is updated based on its current value. An `unmarked` node is marked as the `start` node. A `start` node is marked as the `end` node. An `end` node is marked as `unmarked`. The event is consumed if a node changed its marked value.

Before any of the import functions can be used, the data which should be imported needs to be placed in the controller property `steps`. Graph0 can import a list of graphs with the `_parseGraphSteps` function. `steps` should contain an array of graph objects. If the graph object has the correct format, it is read and placed into the world. The graph object should have the following properties.

1. `type`, with its value set to `"Complete"`.
2. `nodes`, which should be an array of node objects. The node objects in the array are copied to the world.
3. `edges`, which should be an array of edge objects. The edge objects in the array are copied to the world.

A tree structure can also be imported with the `_parseTreeSteps` function. Importing a tree is a more complicated process because the nodes and edges are not available for copying, so they need to be constructed from the tree information instead. To read a tree object, it needs to have the `treeInfo`. `treeInfo` should contain an array of tree roots. If there is more than one root, only one of them is added to the world and buttons which let the user navigate between different trees are added.

3.5.3 Sort

The Sort controller lets users perform the Quicksort and Mergesort algorithms on arrays.

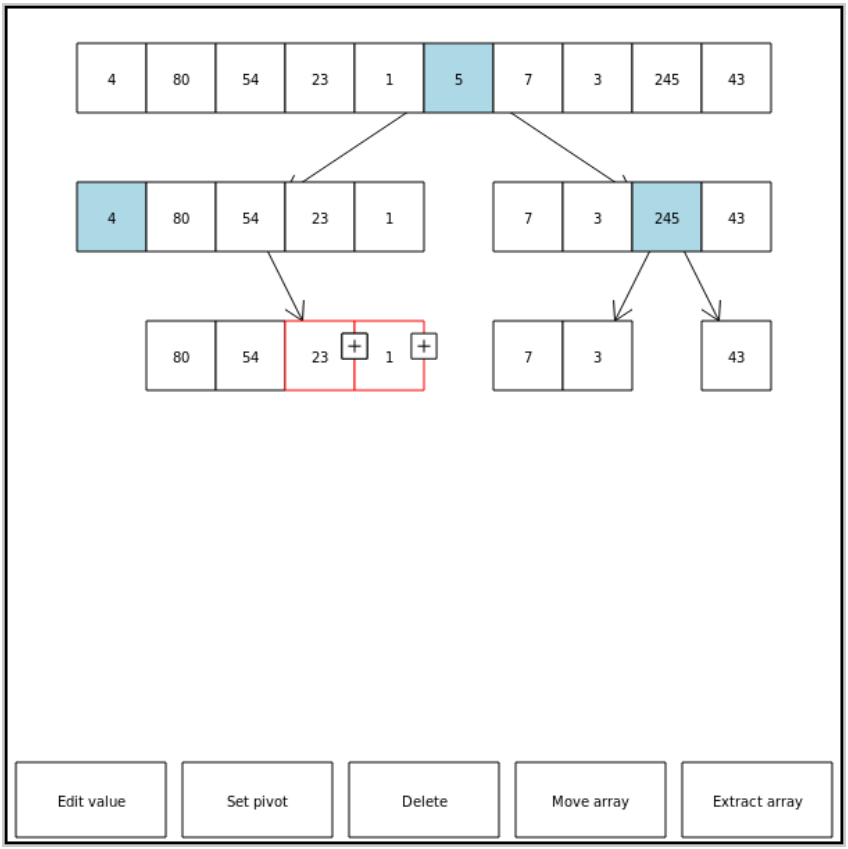


Figure 3.8: Sort - user interface

The following properties can be determined by the optional configuration object:

1. `sortType`, determines which algorithm to use. Can either be `"Mergesort"` or `"Quicksort"`. The default value is `"Quicksort"`.
2. `bsf`, button-size-factor, determines how large the "+" buttons between nodes are. The default value is 3.
3. `pivotColor`, determines which fill color nodes which are marked as a pivot have.
4. `selectedColor`, determines which stroke color nodes which are selected will have.
5. `extractType`, determines node position relative to other nodes when extracting nodes from an array. Possible values are `"vSorter"` which means the nodes are positioned based on their value, from low to high, and `"xSorter"` which positions them based on their x-position in the world.
6. `joinType`, determines node position relative to other nodes when joining the nodes from different arrays to a new array. Possible values are `"vSorter"` and `"xSorter"`.
7. `steps`, contains information about the starting array which the student needs to sort. It can also contain information about all of the actions one of the algorithms performed to sort the array.

The `mouseDownHandler` function has four parts. It first checks if the world is empty. If it is, the first click creates the first node and array. If it is not empty, then it checks if any of the buttons were clicked. The buttons can only be clicked if they are visible, and they are made visible when the user selects one or more nodes. Only one node can be marked as `selected` (marked by + buttons 3.8), but several can be in the selection list (marked by a red border 3.8). When the user selects a node, the "Edit value", "Set pivot" and "Delete" buttons are shown. If the selection list contains nodes from only one array, the "Move array" and "Extract array" buttons are also shown. If the `sortType` is `"Mergesort"` and the selection list contains nodes from at least one array, then the "Join" button is also shown. If no button is clicked, and the user is trying to move an array, the array will be moved to the position of the interaction event. Finally, if nothing else happened, the selected node and the selection list is updated based on which nodes the user interacts with. When the user clicks and holds down their mouse button, the controller will start tracking which nodes are under the cursor. When the button is released, any node which was under the cursor will be in the selection list, and the last node from the list will be the selected node.

Arrays do not exist in the GraphDrawer's world. They are therefore only a part of the Sort controller. The controller has a reference to an array of array objects. The array objects contain a reference to every node in the array, the position of the array, and a list of other arrays which the array has a connection to. When an array is created, its position is set to the position of the first node. This is done so that the array will not move when more nodes are added to it. When a node is added to the array, all of the other nodes in the array will have an invalid

position. To fix this, the node is first placed at the correct index, and then every node in the array is positioned according to their index in the reference array. When a node is removed, the same steps are performed. Because arrays do not exist in the GraphDrawer, edges between them also do not exist. Any link between two arrays is an edge between the nodes closest to the center of the arrays. This is achieved by updating the link when a node is added or removed from the array.

Even though all the controllers have a property with the name `steps`, the format does not have to be the same. Graph0 imports and exports the whole state of the graph after every operation. The steps in Sort contain information about which operation was performed, and on which elements. E.g. `type: "Split", list: [], left: [], right: [], ...`. The sort controller can export and parse the following step types: "Initial", "Split", "Merge", "Complete". The initial step is used to describe the starting array. The split and merge steps are used to show what operation was performed, and its result. Because the exported steps also follow this format, it is possible that the student did something which is not considered a valid operation, e.g., split one array to four new arrays. To handle this, the complete step is used. If a student performs an invalid operation, the complete graph (similar to Graph0) is exported instead of an operation. The following is a list of all the required properties for every step type:

1. `"Initial"`
    - `list`, contains all the nodes in the array which should be sorted. The list should only contain the node values, because the node objects are created by the import function.
2. `"Split"`
    - `list`, contains all the nodes in the array which should be split. The list should only contain node values.
    - `left` and `right`, contains the nodes which are either placed in the left or the right array. At least one of them is required. One of them could be undefined, because splitting an array on a pivot, might only give one new array if a non-optimal pivot was picked.
    - `pivot`, determines which node is the pivot node. If `sortType` is `"Mergesort"` this should be `undefined`. The student is able to mark more than one node as the pivot, however, if that happens, the operation is invalid, and the step type will be `"Complete"` instead.
3. `"Merge"`
    - `merged`, contains the result of the merge operation.
    - `list1`, contains the node values from one of the lists which is being merged.
    - `list2`, contains the node values from the other list which is being merged.

- `pivot`, if the `sortType` is `"Quicksort"` this is the node which is used as the pivot for the merge.

4. `"Complete"`

   - `arrays`, should be an array containing all of the arrays. This does not enforce any restrictions on the arrays, e.g., one array can link to more than two other arrays.

3.5.4 Dijkstra

The Dijkstra controller lets users perform Dijkstra's Shortest Path First algorithm on a given graph.

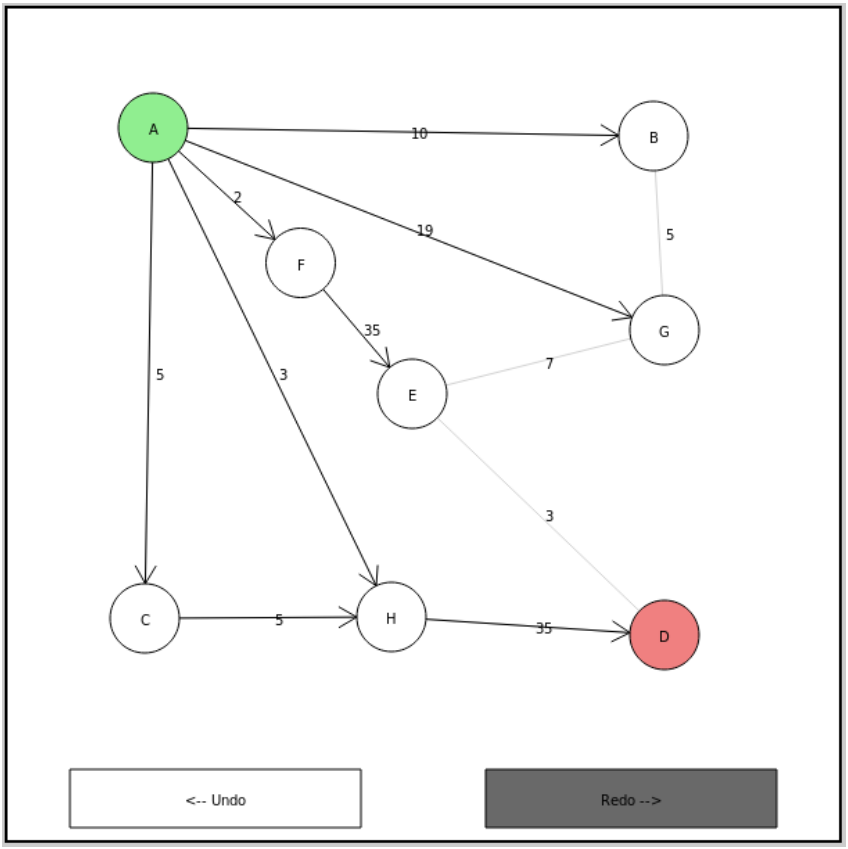


Figure 3.9: Dijkstra - user interface

If the Dijkstra controller is not given a configuration object, then it will only display a white screen without allowing any user interaction. This happens because this controller is made for finding the shortest path, or for showing how the algorithm finds the shortest path. It cannot be used to create a graph. The configuration object can determine the following properties:

1. `steps`, should be an array of the steps the algorithm uses to find the shortest path. This can be undefined if the intention is for the student to use the algorithm to find the path. If `steps` is defined, then the `graph` configuration is not needed.

2. `startColor`, the fill color used for the start node.

3. `endColor`, the fill color used for the end node.
4. `edgeColor`, the stroke color used for lines which show the path between nodes.
5. `graph`, contains information about the graph which the student should use the algorithm on. The following list is all of the possible properties on the `graph` object.
    - `graph`, contains the graph.
    - `from`, should be the starting node.
    - `to`, should be the end node.
    - `nodes`, should be an array of node objects which the graph consists of.

The Dijkstra controller uses the steps array to store operations. There are three step types:

1. `"Initial"`, is used to store information about the graph. It has the same properties as the `graph` configuration object.
2. `"Distance"`, is used to show that the algorithm is checking the cost between to nodes. A distance step has two properties, `current` and `node`, which are references to node objects. The distance step means that the algorithm is checking the cost of traveling from `current` to `node`.
3. `"Path"`, is used to show the shortest path between the start and end node. The path step should only have one property, `path`, which is a list of the node values along the path.

When the user is answering a Dijkstra question, their solution is stored as GraphDrawer edges. The student edges and guide (gray) lines are separated based on the `directed` property. The undo button works by checking if there are any directed edges in the edge array. If there is, then the student has performed an action which can be reverted. When an action is reverted, the edge is stored in the `redolog` array. If the `redolog` has entries in it, then the student can redo the action they reverted. When a new edge is created by the student, all entries in the `redolog` are removed. The reason for using undo/redo actions instead of letting the student remove individual edges is because when their answer is exported, their actions are based on the order of the directed edges in the GraphDrawer. If they were able to remove individual edges, the exported edges would not match what the student meant when they created them.

3.5.5 Python

The Python controller lets users answer questions about the relationship between variables and objects in a given Python script.

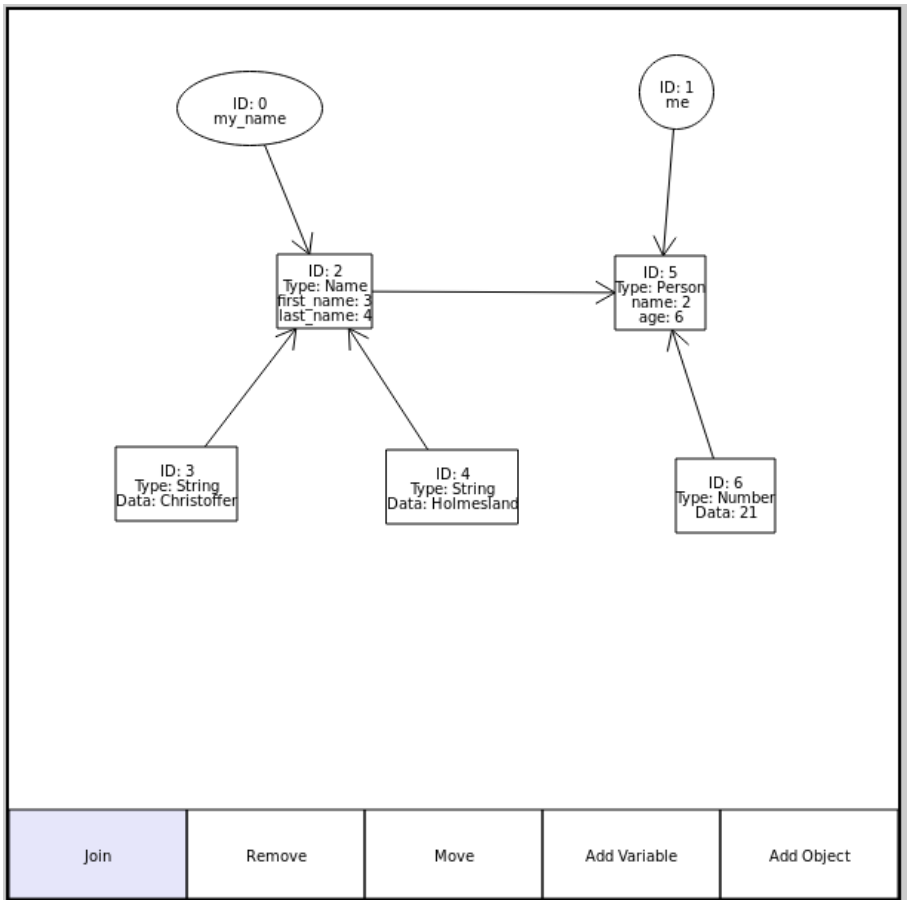


Figure 3.10: Python - user interface

The Python controller can work without any configuration, but it will only allow objects of the base types "String", "Number" and "Boolean". A configuration should, therefore, be supplied to allow for more engaging questions. The only configuration available is the `steps` property. The steps property should be an array with at least one element. When the Python interpreter is used, the last element in the array contains extra information about the classes and functions defined in the script. This step is examined for class information. Every defined class is added as an object type, and the code is parsed again to find the possible class properties.

Like the Graph0 controller, the Python controller implements different states and state handlers. The state can be changed by the user by clicking on one of the buttons "Join", "Remove", "Move", "Add Variable" or "Add Object". The join state lets the user define a relationship between a variable and object, or a relationship between two objects. The remove state lets the user remove a relationship or a node from the graph. The move state lets the user change the position of nodes. The Add Variable state asks the user for a variable name and lets them add it to the graph. The Add Object state asks the user for an object type, and optionally an object value if the object type is a value type, and lets them add it to the graph.

The variable and object concepts do not exist in the GraphDrawer; therefore they are defined in the controller. They are stored in the arrays `variables` and `objects`. A variable needs three properties, `name`, `links`, and `id`. `name` is a string with the name of the variable. `links` is

an array where the elements are the objects which the variable references. `links` should never have more than one element, because a variable is only able to reference one object. It needs to be an array because the student is allowed to make mistakes. The `id` property is the identifier of the GraphDrawer node which is used to display the variable in the graph. The position of the node does not change the behavior of a variable, and it, therefore, does not need to store a complete copy of the node. This also makes converting a variable to JSON simpler, because there is only one node object.

An object needs several properties:

1. `type`, which is a string determining the type of the object. E.g. `"String"`.
2. `baseType`, is a boolean which is `true` if the `type` is either `"String"`, `"Number"` or `"Boolean"`.
3. `value`, is used if the `baseType` is `true`. It contains the value of the object, e.g., `True`.
4. `fields` is an array of field objects if the `baseType` is `false`. Field objects have two properties, `name` and `value`. The name is the name of the property, e.g., `"first_name"`. The `value` is the id of the object containing the actual value of the property. This is done because if the object with the value is not a base type, then the value is another object. E.g., the `name` in the figure references an object of the Name class.
5. `id`, which is the id of the GraphDrawer node used to display the object.

To display meaningful information about variables and objects in the GraphDrawer nodes, the `generateNodeText` function should be used when a variable or object property changes. The text of GraphDrawer nodes needs to be a string in the `v` property. The function converts the properties to a string and adds it to the linked GraphDrawer node. Every node starts with text displaying the id of the node. This is done so the user can easily differentiate between two objects with the same values. The second line of a variable is the name of the variable. Object nodes also show the name of the properties defined in the `fields` array, and the id of the linked object. If the object does not have any fields, the object value is displayed instead. The resulting node text can be seen in the figure.

When the student answer is exported from the controller, it includes all the variables and objects created by the user. A copy of the complete GraphDrawer graph (nodes and edges) is also exported in case the student solution is not valid. When the solution is checked against the correct solution, the information about the variables and objects is used. When the solution is imported back into the GraphDrawer, the graph is used.

The process of importing the correct solution of a Python question is more complicated. The solution only contains information about the Python variables and objects. The controller objects and variables need to be created based on the given information. Only one step is read at a time, and this step is called ”the solution”. This process is however made simpler by the

fact that the solution will always have the correct relationship defined between objects. The solution is imported by first creating controller variables from every variable in the solution. Every object in the solution will either be linked to by another object, or by a variable. It is, therefore, possible to import the solution objects, by starting at the list of objects linked to by the variables, and then recursively looking at any objects linked by the given objects. A combination of the `_uniqueId` defined on GraphDrawer nodes and the id reference on the solution objects are used to create the edges between the variables and objects.

3.6 Python

Python is the programming language used in the DAT110 course at UiS. Students are often struggling to understand how to work with objects because some objects are mutable and some are not. To make it easier for them to learn, a Python interpreter was implemented which parses a subset of the language. The interpreter returns an array of state objects, which contain information about the relationship between variables and objects after every statement. Together with the GraphDrawer, the interpreter can be used to ask questions about Python code. The student is able to see which object a variable points to, and they can see when the value or reference changes. Before the custom interpreter was implemented, using the official Python interpreter CPython [48] was considered. Because of the size of CPython, and the limited time available for this project, it was decided that implementing a custom interpreter was the better option.

The following features of Python are supported:

1. Variables of the following types: Number, String, Boolean, and Object. The Object type is any user-defined type.
2. Functions can be defined using the `def` keyword. They can either belong to the global scope or a class. Functions can return something using the `return` keyword.
3. Classes can be defined using the `class` keyword. If a function with the `__init__` name is defined inside the class, it will be called when a new instance of the class is instantiated.
4. IF statements can be created with the `if` keyword. ELIF and ELSE statements are also supported.
5. The following mathematical operators are supported: +, -, /, *.
6. The following comparison operators are supported: and, or, !, ==, !=, ¡, ¿, ¿=, ¡=.
7. Expressions can be grouped and separated using parentheses.
8. Lines starting with a # are treated as comments and will be skipped by the interpreter.

Significant Python features missing from this interpreter:

1. The standard library.

2. Lists and dictionaries.
3. Inner classes and functions.
4. Shared class variables.
5. Class inheritance.
6. Loops.

Every operator has the same priority, which means that expressions are always evaluated left to right. This makes some expression behave in an unexpected way. The following statement results in an error: `if 1 + 1 == 2:`, because it is evaluated as `1 + (1 == 2)`. To prevent this from happening, parentheses should be used to separate the expressions. The correct statement would be: `if (1 + 1) == 2:`.

3.6.1 Interpreter

There are three main functions in the interpreter. `parseLine` tries to figure out what the meaning of a code line is. `parseLine` is also responsible for deciding which line to parse next. A line can either be a variable assignment which is handled by the `parseLine` function, an expression which is handled by the `evaluateExpression` function, or a statement which is handled by the `handleKeyword` function. A statement is anything starting with one of the keywords. An expression is something which can be evaluated to a value.

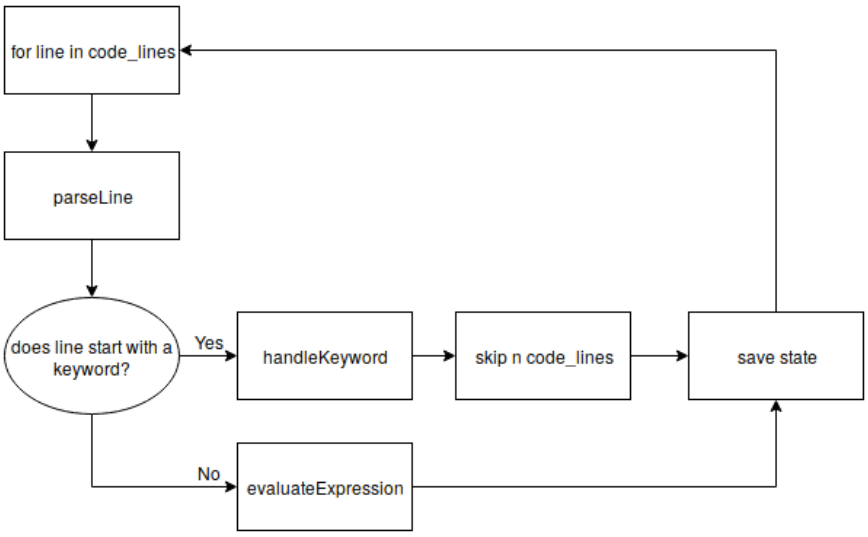


Figure 3.11: Interpreter loop

An important feature of the interpreter is the scope objects. A scope is an object with information about the variables, classes, functions, and data inside the scope. Functions are scopes because they can contain private variables. Classes are also scopes because they can contain functions. Before code can be parsed, a global scope is created. Anything not belonging to a specific scope is placed in the global scope. Scope data is an array where the index is the

address of the stored data and the value is the stored data. Because the interpreter is implemented in JavaScript, there is no need to separate value and references types in this array, because JavaScript and Python behave the same way. Scope variables are a mapping from variable names to data addresses. Scope functions/classes are mappings from function/class names to function/class objects. Because the interpreter should save the state of the program as steps, the `parseLine` function returns an object containing information about the state. If the state was anything other than a function or class definition, the current state is saved as a step.

A function is an object with a `name`, a list of arguments `args`, a list of the code lines belonging to the function `code`. A function should also be a scope. The name is used to identify the function. The arguments are used to make sure that when the function is called, the right amount of arguments are passed. The code is stored so the function can be evaluated when it is called. After a function has been defined, the `handleKeyword` function returns a state of type `"SkipLines"` which tells the caller which lines have already been handled. The interpreter uses the `callFunc` function to call Python functions. When a function is called, three things happen in the following order:

1. The function arguments are added as local variables. Local variables mean they belong to the function scope. The arguments need to be in the same order as they were defined because named arguments are not implemented.

2. Every line found in the functions `code` list is parsed using `parseLine`. If a `return` statement is found before reaching the end, the function will stop parsing, before reaching the end.

3. The scope of the function is restored to an empty state and data is returned if a return statement was found.

A class is an object with a `name` and a list of the code lines belonging to the class `code`. A class should also be a scope. When a class is defined, the code is also parsed using the `parseLine` function. This is done so that any functions within the class is defined in the scope of the class. When the interpreter is instantiating an instance of the class, the `instantiateClass` function is called. `instantiateClass` first creates a new object containing all the functions from the class. If the class has a constructor it is called. Finally, the object is returned to the caller of `instantiateClass`.

When a code line is not a statement, it is passed to `evaluateExpression` so that it can be parsed. The following list contains information about how the parser determines what kind of expression the line is, and in what order, it happens.

1. The function starts by checking if the line is a base type value. Base types are defined as String, Number, and Boolean. If the expression is determined to be a base type value, an object containing the `type` and `value` is returned.

2. If the expression is not a base type value, then it could be a variable. The current scope

is checked to see if it contains a variable with the same name as the line. If it does, the data referenced by the variable is returned.

3. If the line contains a "." it could be a property access or a function call. The possible object and property names are extracted. If an object with the given property name exists, the referenced data is returned. If it does not exist, the expression can either be an undefined property, or an operation, e.g. `self.x + 1` will first evaluate to the property `x + 1` of the `self` object. If the property is undefined, an error will be thrown. If the property name contains an operation, it can be resolved by evaluating the `propertyName` as an expression relative to the object scope.

4. The expression is now checked if it is an operation, function call or class instantiation. This is done by looking at the parentheses.

5. If the expression is a function call or a class instantiation, the arguments are extracted and evaluated, before the function is called. This is done because a function call can contain expressions inside it, and any argument which is not an expression will return itself. Example: This `my_console.print("Hello, " + "World! From: " + my_name)` function call contains one argument call referencing variables belonging to the global scope. When the function is called, the code is evaluated relative to the `my_console` scope, which does not contain a reference to the `my_name` variable. Python does not allow classes and functions to have the same name, meaning that there is no need to differentiate between function calls and class instantiation, because class instantiation is a function call to the `__init__` function.

6. Any expression which has not already been evaluated should be an operation. An operation is an expression containing at least one operator and at least one expression. An operation expression is always evaluated left to right unless parentheses are used to group and separate operations. E.g. `1 + 2 + 3` is split into expression1 = 1, operator = + and expression2 = "2 + 3".

Python code is always written on the client side by a user and then sent to the server so it can be interpreted. Code is simply text, so transferring it works flawlessly. After the code has been parsed, it is either stored in the database when a question is created or sent back to the user if it came from the sandbox. The result of the interpretation is an array of state objects. Normally one of the state objects could be inspected, and it would be possible to see that both `a` and `b` reference the same object in the following code snippet: `a = "Dummy-text" b = a` . Before transferring a JavaScript object to a database or over a network, the object needs to be converted to a JSON object. When this conversion happens, the references are lost. They are also not available when the JSON object is converted back to a JavaScript object. When inspecting the code, it would like `a` and `b` reference two different objects with the same value. For this reason, a system for manually tracking object references was implemented. Every scope, variable and object is assigned a unique id. When the SolutionChecker or the GraphDrawer wants to check if two objects are the same, the usual JavaScript comparison (`a`

`==` `b`) cannot be used because it would always return false. Instead, the id needs to be checked, `a.id == b.id`. The id stays the same between different interpretation steps, and this means that the GraphDrawer can keep track of which variable links to what object simply by creating a mapping from the unique id to the GraphDrawer node.

3.7 Database

Since the web application requires a database to store information about users, questions, sessions, courses and more, a set of files were created to get, insert and update the information. The database is designed around features needed for the application. It was also created for scalability in mind with for example the possibility to add other OAuth/OpenID connect possibilities in the future. There are five scripts associated with the database, where get, insert and update functions have been separated into their own scripts. One script is connecting the get, insert and update scripts making it easier to import and use the database functions in other scripts. There is also a script which will run every time the server starts. This script makes sure that every table is created and have the correct column names. It is also the script that inserts default values such as question types and the anonymous user.

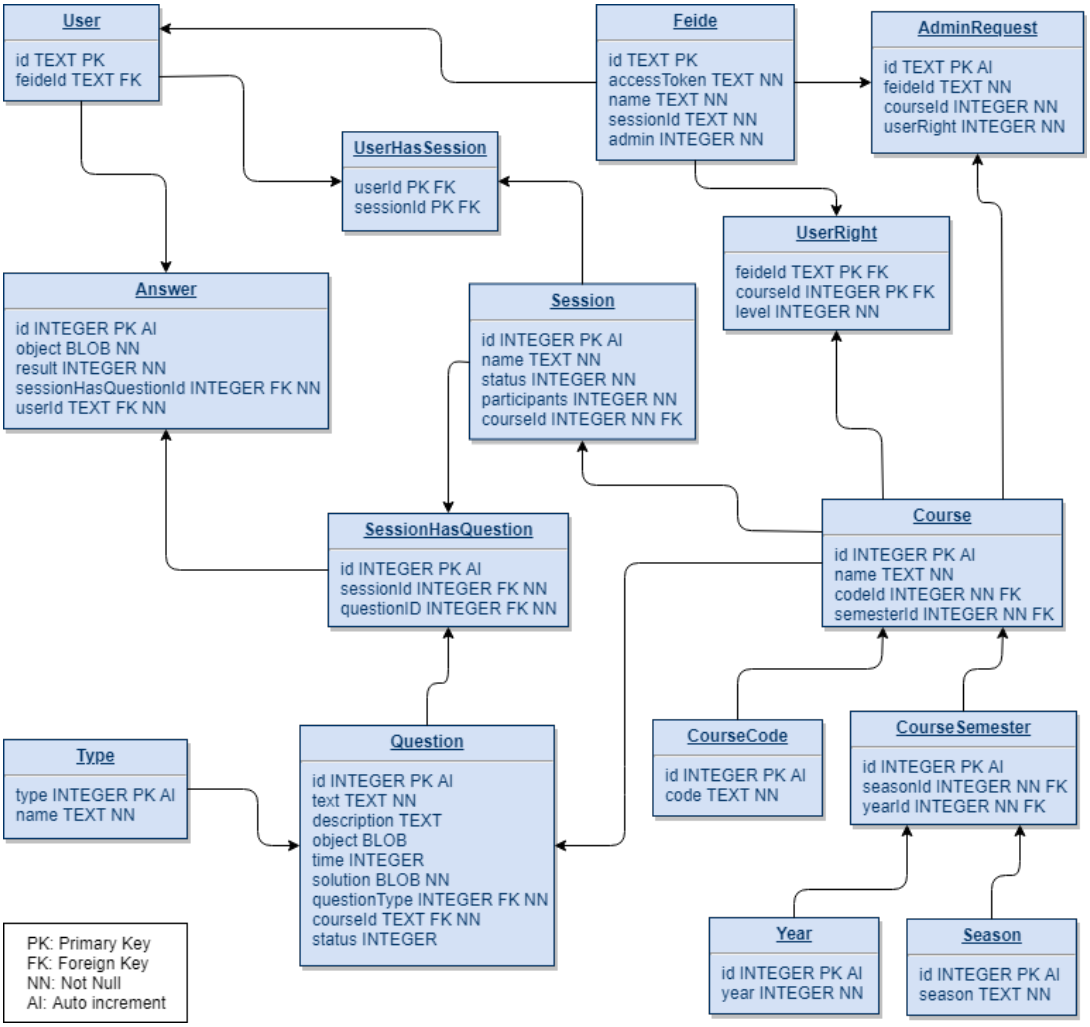


Figure 3.12: This is the database schema used for the application.

The table "User" has the anonymous user with user id 1. When an anonymous user answers a question, the answer is linked to that user, no matter who answered. Feide information is in its own table, this makes it easy to add more types of login services, e.g., Facebook, Google, and more. When an admin adds a question to a session, it is linked in its own table called "QuestionHasSession". This is to ensure that a question can be linked to more than one session or multiple times in the same session. The answers are also linked to the "QuestionHasSession" table, making sure that the answers shown to admins are only for that question in that session and not the total answers statistics for the question in general. Sessions are linked to a course. A course contains information about the course code and semester, enabling the option to separate sessions from each semester. Questions are linked to the course id, but admins do have the option to copy questions from one course to another in the question dashboard.

3.8 Testing

3.8.1 Unit & Integration Testing

All the unit and integration tests can be found in the folder named "tests". This folder contains the following three folders:

- algorithms
- cypress
- solutionChecker

The algorithms folder has the unit test for the algorithms used in the application. It is primarily used to test whether the algorithms do their intended job. The "test_sorting" JavaScript file contains unit tests for all the sorting algorithms. The "test_trees" JavaScript file contains the tests related to the `Tree` and `BinaryTreeNode` and the functions related to tree objects. This file has a mix of unit and integration tests. The "test_dijkstra" JavaScript file contains unit tests for the Dijkstra algorithm. The solutionChecker folder has a test for the quicksort solutionChecker. To run all the unit and integration tests input the command `npm run Test`. This command must be run in the App directory.

3.8.2 End-To-End Testing

The end-to-end tests in this project are stored in the "tests" folder, together with the unit-test, but in its own folder with the name "tests/cypress". Running the end-to-end test requires only clicking on any of the files, once the Cypress interface is open and visible. The different test files can be found within the folder named "specs". Every describe function in the test files are counted as their own test suite.

The time it takes for the web application to be loaded depends on the computer's processing power. Cypress executes commands faster than any normal human being. There is a possibility that some end-to-end test fail because the computer loads the content of the page slower than it takes Cypress to execute the wanted command. If this happens the test case fails, and this

most likely affects the following test suites.

It is intended that the end-to-end tests are run using the project's testing environment. In this environment, the server will run on port 8082 instead of 8081. It also requires the user to set up an environment file to test. To build the client in the testing environment, use the `npm run buildTest` command. To start cypress and server, use the command `npm run cypress:open`.

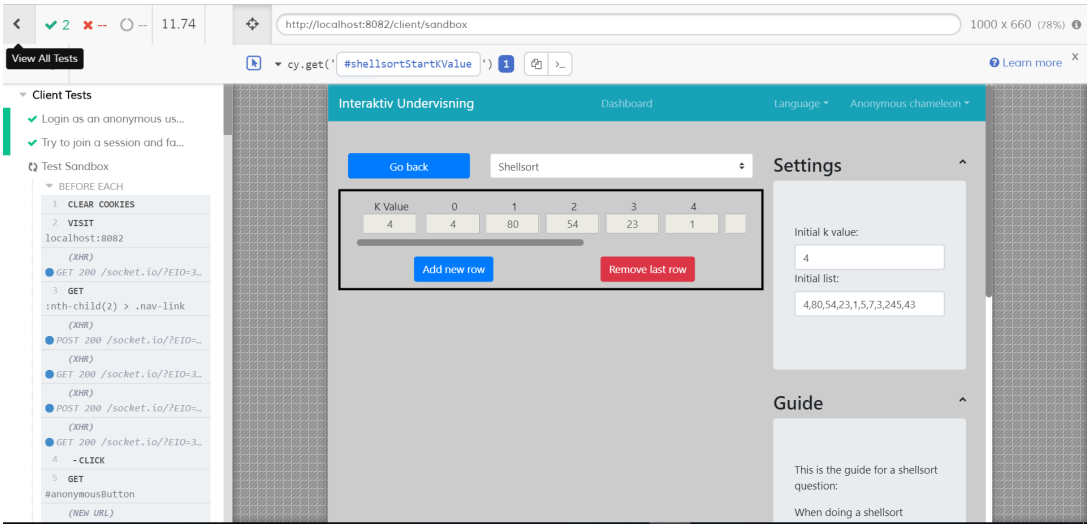


Figure 3.13: The figure displays the cypress testing tool.

4 Application

4.1 Creating a Course

The admins need to first create and select a course because most of the admin related actions are linked to a course. When creating a new course, a title, course code and a semester are needed. There is user input validation on both the client and the server. If they are not valid an appropriate error message is shown informing what went wrong. Both the course code and semester is only required to be created once. The course data is stored in the database. The course code is set up to be a string with three capital letters followed by three numbers. The semester is a combination of either spring or autumn and the year. The year and season is not something the user can edit, and each time the server starts the current year and the three following years are added to the database. Once a course is selected, an admin has the option to manage other admins and student assistants for the course.

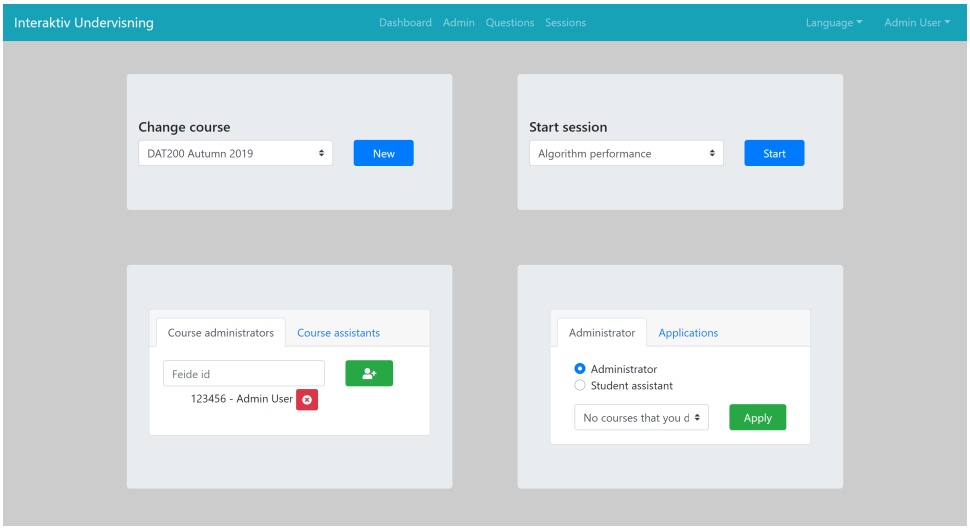


Figure 4.1: This figure displays the admin dashboard.

4.2 Creating a Question

Everything requiring a user right level above 2 uses the Admin component as the view. In order to start creating questions, the browser loads the Questions component into the Admin component. The NavBar component changes the URL route to load the Questions component. This makes the Vue Router change the displayed component. The URL can be changed using the `$route.push` function on the Vue context. The Questions component consist of other components such as EditQuestion and ShowQuestion. All components related to creating a question can be found in the "./App/client/src/components/admin/question" folder. On the "/admin/questions" page there is a select form to the left, which is in the SelectCourse component. Keep in mind that a course must be chosen for the user to able to create new questions. This is done because every question is linked to a course in the database. If the user attempts to create a question without having a course in the database, an error message will be displayed to the user.

The area in the middle of the Questions component is where the current questions for the selected course are going to be listed. The Questions component sends a request to the server whenever the component is loaded, the course changes, a question is edited, or a question is created. The request asks the server for all the current questions linked to the selected course. The server then calls the appropriate get function from the database script and returns the result. The result contains question titles, ids, and statuses. The result is then used by the Questions component to update the list of questions. In this way, the list should always be up to date with the contents in the database. A v-if statement is used to check whether the list containing questions is empty or not. If the question list is empty a list item containing a message is displayed. A v-if statement is a Vue feature allowing the use of a conditional statement on a HTML element. In this use case, the HTML element is only included on the page if the question list is empty.

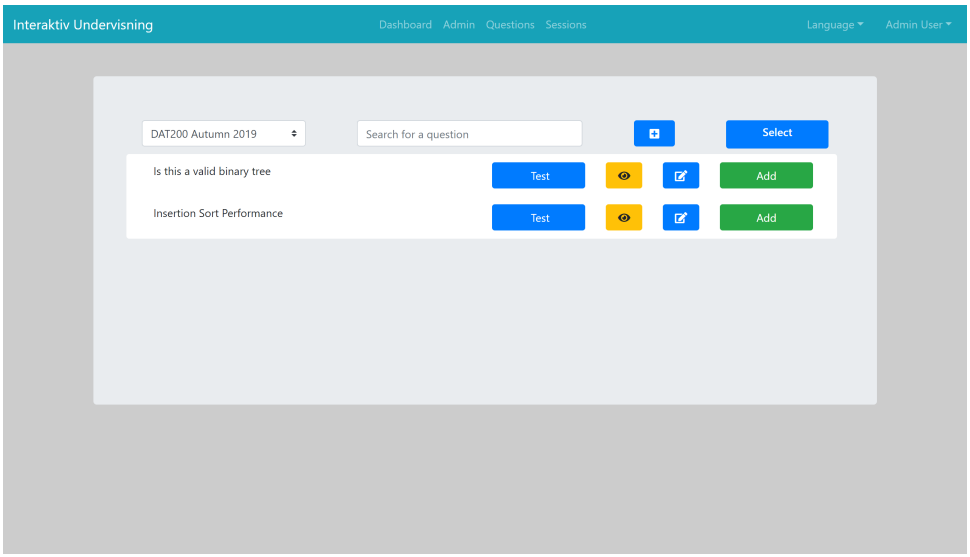


Figure 4.2: The figure displays an image of the Questions component.

Clicking the "+" button triggers the `showAddQuestionModal` method, allowing the EditQuestion component to be loaded in. This component contains a Bootstrap modal used for adding and editing question information. The main differences between the add and edit operations for the component are the following:

- The edit functionality needs to load the question data from the database and fill in the forms with the correct information.

- The request sent to the server contains the same question information, but the socket function that is triggered is different. The server functions are different primarily to distinguish between the insert and update SQL commands used on the database.

The modal in EditQuestion is divided into three parts, namely Basic Information, Media and Solution type. The Basic Information part contains an input field for assigning the question title, a text area for giving a question description, a time input field and a time slider for assigning time. The fields use v-model to mount to the object `newQuestion`. A v-model is a

two-way connection between a variable and a HTML element. The variable has to be a property on the `props` or the `data` object. This makes sure that if the variable value is changed either by functions or by user interaction, both will have access to the updated value. The values assigned from the different form inputs are stored in the `data` property of the Vue context. The time slider and time input use different value formats, because of this, the time slider is linked with the `time` property. The time input uses a computed method (`timeInput`) to convert the `time` property to a text value. If the user changes the time input the computed method will convert it back to an integer value. This means that these elements always change their value depending on the other, resulting in them having the same value. If the time is set to "00:00", then the question will not have a timer when used in a session. In the database, the value is stored as an integer, if the time value is 0, it is stored with a value of -1.

If the user uploads an image, the EditQuestion component the method `newFile` is triggered by an `onChange` event. The method validates the image because it is important to check that the file is an actual image file. If the validation passes, the image `File` object is converted into a buffer. This buffer, alongside the name, size and file type are pushed into a list as an object. When a question with an image is sent to the server, the image list is sent with the question information. On the server, it will first move the image buffer into a temporary list. Then the server creates file paths for each image and places them into the question object. After the question has been stored in the database, the image buffers are turned into image files and stored in the file system. Whenever a question is obtained from the database that has images, the server loads the images by using the file paths stored in the question object. When the image is going to be used on the client, a new buffer is created based on the image and stored in the question object. A question cannot have too many images assigned to it, as this would take up a lot of the needed data. The total amount of files attached to a question cannot exceed 1.5MB, and the user will get a warning once the amount exceeds 500KB. These checks will run in the validation once an image is added to the question. Images in a question are displayed as a preview with filename, file size, and a small image on the side.

Tables are added using 2-D lists where each element is a row. Each cell uses a HTML input field and has a v-model linking the value directly to the 2-D list. When displaying the 2-D list, each row and column is generated using a Vue feature called v-for. This feature allows HTML elements to be written once, and let a for loop generate the elements n times, based on the v-for input. The application also allows the addition of GraphDrawer drawings in an image format. When exporting a canvas element from the GraphDrawer as an image, a function `toDataURL` is called on the canvas element. The function returns a base64 data URL. Based on the information from this string an image object is created and added to the image list alongside images added by the user.

The solution type part focuses on determining the question type of the new question and for creating the question's solution. The type decides what format the question has for both the solution and for the client during a session. The solution to questions revolving data structures

and algorithms is created on the server in the solution generator. The user only needs to give the necessary information for the question so that it is possible for the student to solve it, and for the server to be able to create the solution. There are two reasons for having the server handle the solution generation. The first reason is to keep it away from clients participating in a session. The second reason is that forcing the user to write the entire solution for every question would be rather tedious. Not to mention reducing the chances of having the solution being written incorrectly.

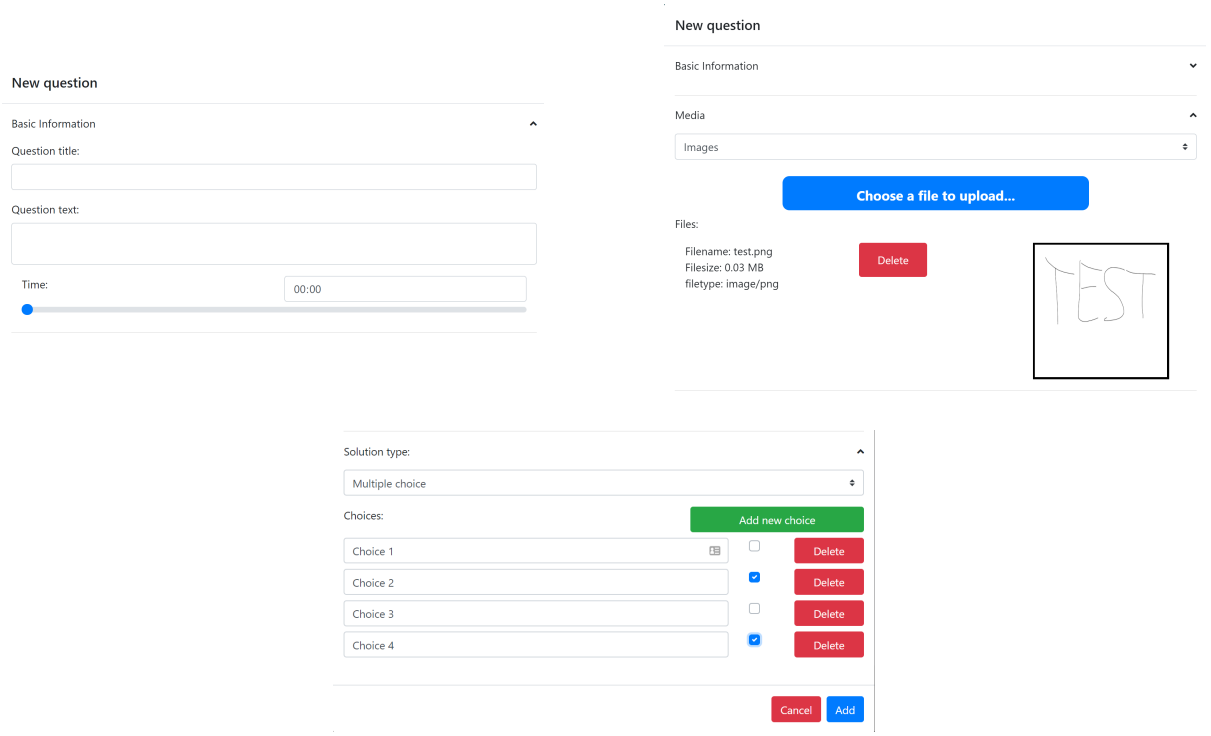


Figure 4.3: These figures display sections of the EditQuestion component.

When a user is finished creating a question, the question information is sent to the server through a socket request. The data sent to the server is the `newQuestion` object defined in the function `initializeState` in the component. Since all the properties in the `newQuestion` object are linked to the component, the server should have all the information it needs to generate the question solution. However, to avoid any problems regarding storing faulty questions on the server, the question information will first need to be validated by the server's validation checker. The user must at minimum fill in the information in a title, and the solution type with the solution properties for the wanted question to be valid. If the question information passes the validation checker, the question information and solution are inserted into the question table in the database. After the question is stored, socket messages are sent updating the question list on the client.

The question validation is divided into three script groups and linked together using a master script. The master script runs the validation for the first script group validating the basic information. The second script group runs the validation for any attached media. The third

script group is the largest and runs validation on the solution. All script groups run even if the first group fails. If a group fails an error is pushed into a list containing all errors. After the master script is complete, the result is sent back to the client. If a check fails the error is displayed in a Bootstrap alert box informing the user what went wrong.

If the validation succeeds, then the information is sent to the solution generator that generates the appropriate solution in accordance with the question's type. The solution generator has a structure similar to that of the validation checker. Each solution generator has its own JavaScript file. The main "SolutionGenerator.js" file is used to link the question to its proper solution generator function. If the question requires the GraphDrawer tool to visualize its solution, then the solution object created from the generator is going to follow a certain format. The solution object here is an array that contains all the steps necessary to solve the exercise marked with a type property assigned with the action taken at the step. The last step is going to indicate either "finish" or "done" and is always going to be the last entry in the array. It is this object that is going to be used in the solution checker during an active session. The reason why this structure is used for solutions using the GraphDrawer, is because the GraphDrawer can show the entire process of solving the chosen exercise revolving an algorithm or a data structure.

If the user copies a question from one course to another course, all of the information is inserted to the database as a new question. This is done to prevent a change happening in one course affecting the question in the other course. A question cannot be deleted or edited after it has been used in a session. The reason for this is that the question information is still needed for Feide users to look at their previous participated sessions. If the question was to be removed after having been used in a session, then the student would not have access to the question information.

The question structure for this application was specifically chosen such that it would be easier for developers to implement and add new questions types. Excluding the work that is needed for writing the actual data structure or algorithm and its solution format, all that is required in order to add a new question type to a session is the following:

1. Create a Vue component in both the questionResultScreenAnswer and questionResultScreenSolution directories. These are used for the DisplayQuestion component. The DisplayQuestion component is used for showing question results for each question.

2. Add a b-form-group element to the EditQuestion component where only the necessary form items or Graphdrawer properties are set so that the user can create the new question type.

3. If the question type requires certain unique information, then an extra property in the `newQuestion.objects` in the `initializeState` function should also be assigned and modeled appropriately to the HTML element in the form group.

4. Add a JavaScript file in the validation checker for stopping illegal actions to the new type.

5. Implement the solution generator for the question type which is going to create the format for the solution object stored in the database.

6. Add the new question type to question type array in the database.js file.

7. Finally, implement a solution checker that is going to verify that the student answered the question correctly according to the solution object.

8. Add text to the locale files if the new question type requires them.

4.3 Creating a Session

The structure of the Sessions component is similar to the Questions component. There is a list of sessions that is kept up to date when a session is added, edited or deleted. A session contains a list of questions that are stored in a separate database table linking the Question and Session tables. This ensures the possibility of adding a question to multiple sessions, and that an answer does not affect the statistic of another session.

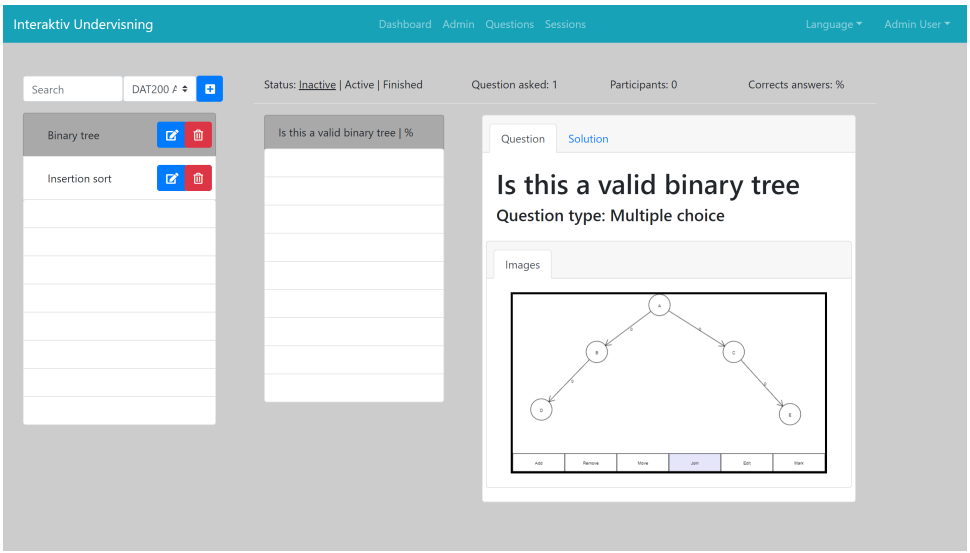


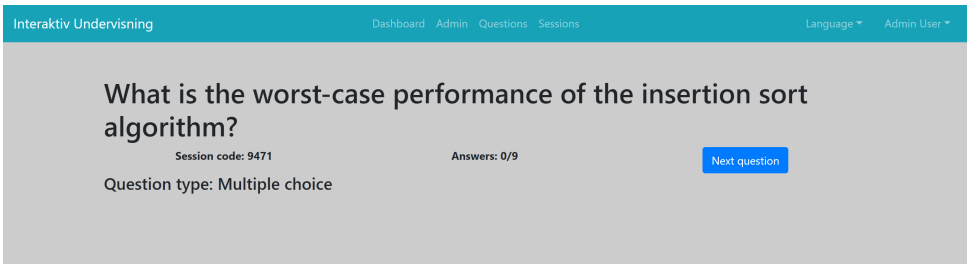
Figure 4.4: This figure displays the Session component.

4.4 Active Session

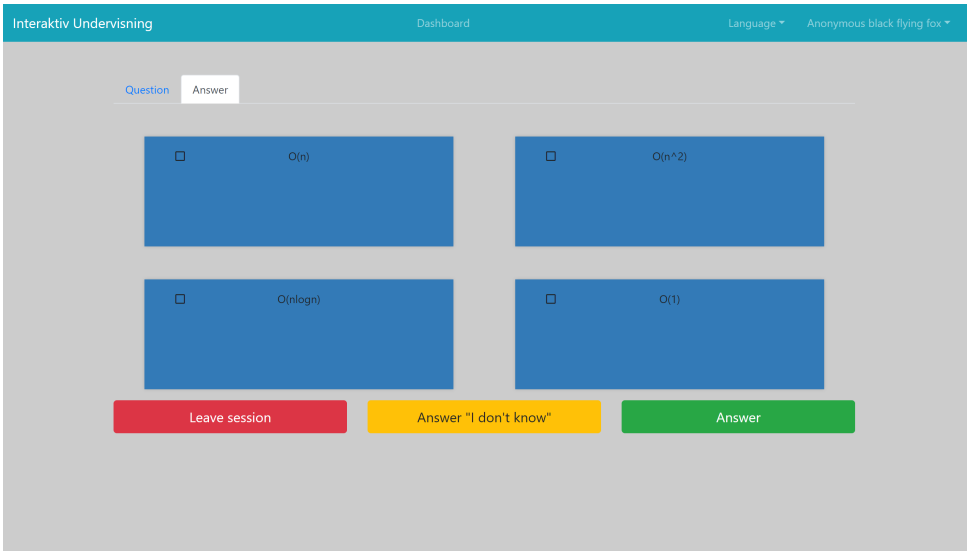
Admins or student assistants have a selector on the main admin dashboard for selecting a session. If there are no sessions, the user will be prompted to create one. After the user has selected a session the start button will be enabled. Pressing this button sends a request to the server requesting this session to be initialized. The server will then proceed to request information from the database of the session. Information of the session is stored as an object in a map. After the server has made the session ready, it will send a response back to the client and switch the view to a waiting room. The waiting room shows a code that students can use to connect and also display the current number of student connected.

On the main client dashboard, there is an option to join an active session. Once a client has

joined a session, the user also joins a room in Socket.IO. This is a feature that allows the server to send a message to all clients connected to that room and is used when the session is going from the waiting room to a question, or to the next question. When the admin sends a start signal, the server retrieves information for the next question and send a message to all clients connected to that room.



(a) This is the host page.



(b) This is the client page.

Figure 4.5: These figures display a question during an active session.

When a student receives a message with a question, the active session component switches to a component made to display a question. There is also a similar component for the admin, but they differ as they need other features. When the question component for a student is first shown, the answer tab is displayed, but the student can switch to see the question. This was done in an effort to give students the possibility to view each question if the display in front of the class is hard to see. When a student is on the answer tab, there is a component allowing the student to give their answer in a way that is specific for that question type.

When a user sends in their answer, it is sent to the server, where it goes through a solution checker. When using the solution checker, the main script needs the answer, solution, and question type. When getting to the main script, it looks at the question type and then sends the answer and solution to the correct checker. There is a solution checker for each question type, and it will go through the answer and compare it with a solution. The comparison will depend on the question type, but in general, it will loop through the solution and compare the

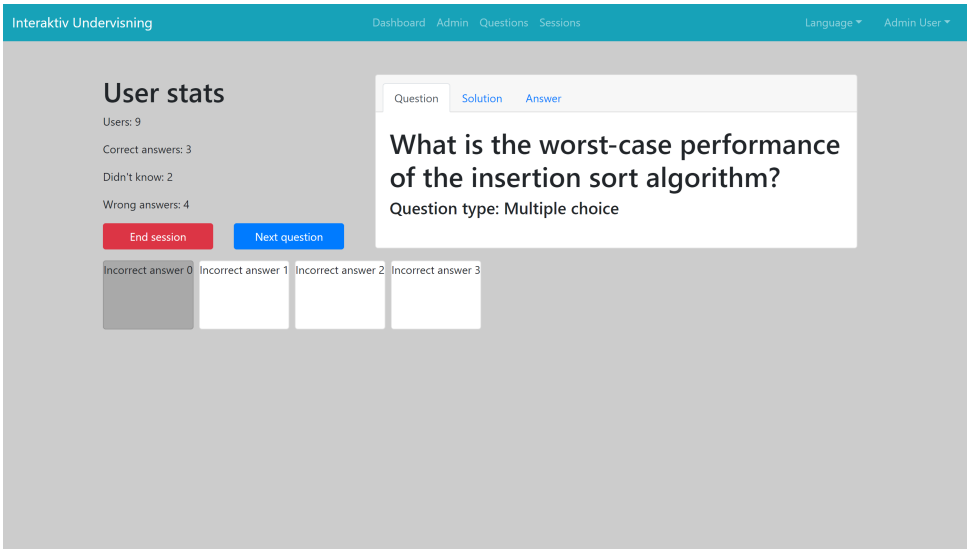


Figure 4.6: The figure displays the result screen of a question.

current loop step with what is in the answer. The current solution checker structure can be seen in figure 3.2, where the main script is called "Solution generator" and then all the sub scripts are for each question type. This not only makes it easy to expand with new question types but also makes the debug for solution checking easier.

When the server is finished with the solution checker, it sends a message back to the student moving them to a waiting screen with the result. The result will be shown as text and will be as discreet as possible. While the student is transferred to the waiting area the server sends a message to the host of the session informing how many users have answered the question. Once every student has answered the question, the server stores the answers in the database.
There are three ways for the session to move on from a question. If all the users connected to the session have answered the question it will move on, if the timer runs out or if the host presses the next button. All students are still in the waiting area, while the server sends statistics and all the wrong answers with the solution to the host. The host is shown basic statistics and a list of all wrong answers, where the host is able to compare the wrong answer with the solution. When the host is ready for the next question there is a button to press to move on. This sends a request to the server requesting the next question, and if there is a new question it is sent out to all students connected to the room and the host. When there is no question left, an end screen is shown for both the students and the host informing them that the session is now over and giving them a button to return to the dashboard.

If a user leaves or joins an active session, a message is sent to the server requesting the appropriate event. Leaving a session triggers the following events:

1. The server will first check if the user has answered the question or not.

2. The server uses different counters to keep track of users during a session. This is done to make sure the counter on the host screen shows the correct amount of answers and users.

It is also used to go to the result screen when all users have answered. When a user leaves the correct counters will update

Joining a session triggers the following events:

1. The server will first check if the user has answered the question or not, and send them to either the waiting room or to the question.
2. The counters are updated.

If the host loses connection, refreshes the page or leaves the page, an interval starts, giving the host five minutes to reconnect. No student is kicked out and can answer the current question. If the host does not reconnect, the session is removed as an active session, and all students are moved to the end screen. During these five minutes, a host can not start a new session and can rejoin the session or clear it. Clearing the session ends the session and sends all connected users to the end screen.

4.5 Sandbox

The application also includes a sandbox page, where students can practice the different question types. The content of the Sandbox component changes depending on the selected question type. The question type select element is mounted to the data property `questionType`. Depending on the value given to the `questionType` data property, the component content changes in order to accommodate the selected question type. This is done by using `v-if`. Question type Text is the default value for the select element. The only component used in the Sandbox view is the GraphDrawer.

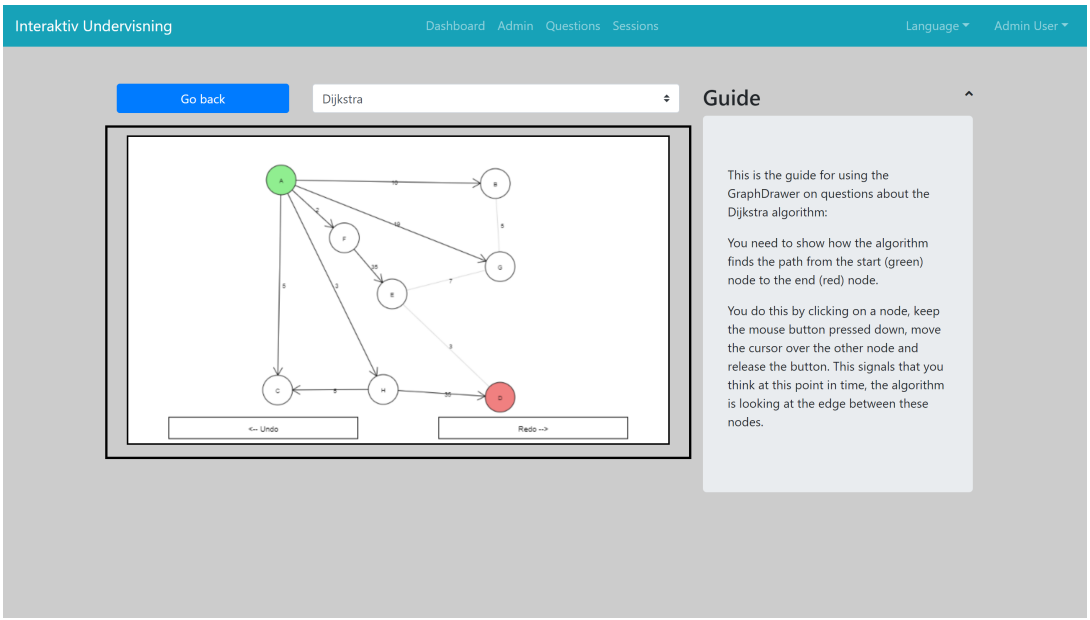


Figure 4.7: This figure displays the sandbox page.

The elements that are needed for a question type are always displayed in the middle of the page in its own container. The `getShowSettings` function is responsible for revealing the settings

section when it is required; otherwise, the function is responsible for removing the section altogether. The content in settings is determined by the question type and uses the same v-if system as the earlier segments. The guide and settings sections are also mounted to their own property called `showGuide` and `showSettings`. These properties determine whether the content in these sections are displayed or hidden to the user.

4.6 Localization

A feature that allows a larger group of people to use the application is allowing the user to change the language used throughout the application. As a base, the application lets the user select between English and Norwegian. The feature is structured in a way such that it allows easy addition of new locales. A locale contains all the text needed for the application for a given location. All locale files are loaded into RAM when the server starts. Locale files for this project are around 30KB. Since every user connecting to the server requests a locale file, it is an advantage to store them in RAM where the server has quick access. The file size allows the server to store them in RAM without having a significant performance impact even if there where a large number of locales. On the client the default locale is Norwegian. When the user changes the locale, a request is sent to the server, where the server will find the correct locale and send it back. When the client receives the response it stores the locale inside the Vuex store. This allows all Vue components to access the locale. If the user is authenticated with Feide, a cookie is stored when the locale is changed. This ensures that the user's preference is loaded when visiting the web site.

The structure of a locale file is a JSON object where usually a component will have its name as a property in the object. This allows each component to only get locale that is for itself. This setup also makes the addition of new locales straightforward. When adding a new locale a new file with the filename of the locale is created, and the content from another locale file is copied over and translated. When the server starts, it reads the content of the folder containing all the locales, so there is no need to add code when implementing new locales.

5 Evaluation

5.1 Vue

To make the interface between the code and the website simpler, we wanted to use a web framework. We needed a dynamic webpage where the different parts were constantly changing depending on the user interaction. Code separation was important for us because we wanted to be able to split it up as much as possible to minify the risk of one error affecting the entire application. We expected Vue to make this easier. Code reuse was also important. We knew from the beginning that the questions and solutions were going to be displayed on different pages, and we wanted only to write the code for this once. We had previously used templating engines, which makes it possible to use JavaScript variables directly in the HTML. This is something we expected Vue to do for us.

We had no experience in making single page applications, so we had a lot to learn. Vue met our expectations, and also had many features we did not know about. Examples are the Vue Router. We did not consider the fact that when there is only one page, the user cannot use the URL bar of the browser to navigate. We had some problems with storing application state between different views and components, but after some searching, we found Vuex which was made to solve that problem. Our experience is that Vue did not prevent us from doing anything we wanted to do, and it provided good solutions to common problems. Much time was spent in the beginning trying to learn Vue, but the time investment was worth it considering how much time we would have spent on recreating the same features found in Vue.

5.2 Express

From before, we had experience using Express and knew it supported the use of middleware functions. We assumed this would make the process of implementing OAuth2/OpenID Connect easier. Based on research, we expected Express to work well with Socket.IO, SQLite and Vue. Based on our experience during the development, this turned out to be true even though some configuration had to be done.

5.3 Socket.IO

In the web programming DAT310 course, we were taught AJAX requests. Even though this works well when the client needs information from the server, there is a limitation when the server needs to send a message to a client. During the software development DAT210 course, one of our group members worked with WebSockets. WebSockets allowed there to be a two-way connection between the server and clients. This seemed like a good solution for our project since the server needed to talk to clients when running a session. Socket.IO is the only well-developed WebSocket for a JavaScript-based server; therefore it was a natural choice. Despite being the only well-developed option, it fulfilled our needs and expectations.

5.4 SQLite

Our experience is that SQLite works as well as other database management systems. It was advantageous not to have to worry about a database server while developing the application. However, it surprised us to learn that foreign keys constraints were not enforced by default. Towards the end of development, we also discovered that SQLite only supports 60 transactions per second on an average HDD. If the limit is reached, the transactions build up in a queue and is ran once the database is free. After looking into the issue, we discovered our server requires following amount of transaction per action:

Action	Transactions
Login anonymous user	0
Login Feide user	1
Create/Edit question without images	1
Create/Edit question with images	2
Create session	2
Edit/Delete session	1
Add question to session	1
Delete user data	1
Starting/Ending session	1
End of question round	1

Table 5.1: Transactions per action

Looking at the table 5.1, one can see that the limit of 60 transactions can be reached quite quickly. If we had known this from the beginning, we would most likely have chosen a different database management system. Even after taking the last issue into consideration we still think that SQLite can handle the traffic that is produced at the scale it is going to be used at.

6 Conclusion

In conclusion, the current application is a web application that gives teachers the ability to create sessions which contain questions that are relevant to a lecture. The application implements all the features from the primary goals for the project. The application supports the use of WebSockets and can have multiple students participate in a session. The sessions have a layout that supports both mobile devices and computers. While the admin portion of the application is designed for pc users, it works on mobile. The application has a drawing tool, that is designed to be used both to answer questions and create questions about certain algorithms and data structures in the courses DAT110 and DAT200. The application stores the result of questions and sessions. This data can be used by the teacher to check if students understand the material in the course. Other features such as supporting OpenID Connect for student authorization, student feedback, and localization support were added during the development because these features worked well with the planned structure of the application.

During the development of the application, a lot of different tools had to be used. In general, the group had very little to no experience working with the tools used in the project. This includes developing single page web applications using Vue, using Socket.IO for the WebSocket functionality of the sessions, and OAuth in order to authenticate students using their Feide users. By the end of the project, the group members have gained experience working with these tools, and the group has, in general, learned a lot more about modern web tools and frameworks. The group has learned a lot about how to structure larger software projects, and working in teams.

One of the challenges with implementing solution checking and generating is that user input can create objects which do not fit with the actual data structure. For instance, when answering a question about binary trees, they can create nodes which have more than two children; they can create trees with several roots, or create cycles in the tree. This made it a lot harder to create correct algorithms because there were many edge cases for each question type.

6.1 Future Development

6.1.1 Server

Due to limited time and resources, a decision was made to make a single server and make it responsible for handling all traffic from the users. This will not be a problem for the server load for the use case at the moment. If the web application should be scaled up to include subjects for the entire university and all its students, or scaled up even further, the server should be split up into microservices, where each service is its own server serving a single purpose. For example, everything that has to do with login is its own server and everything that has to do with running an active session is on its own server. These microservices could be placed inside their own Docker[49] container and with the use of Kuberenetes[50] the servers would scale up and down depending on the amount of traffic. Since the amount of work required in R&D for

all the other features and technologies we used, this was something we wanted to do but did not have enough time to get done.

Another factor that limits the scalability for the server is the use of a map for connected users and a map for active sessions. After some testing, this is not a problem when the use case is a couple of classes and a couple of active session at the same time. If the traffic were to be scaled up for the entire university or multiple universities, there could be a problem since the limiting factor would be the available RAM on for the server. A way to avoid this would be to redesign how sessions and users are stored in the database, allowing the information that is currently stored in RAM to be stored directly in the database.

6.1.2 Database

Another feature that would suffer if the web application scaled up is the database. The database used is SQLite. The hardware running the server decides how many transactions per second can be used. If a regular disk drive is used, the limit is around 60 transactions [51]. This could be a limit if the traffic is scaled up enough, and it should be looked into using another type of database that uses a server to handle the requests. This would also allow the database to be split up into smaller databases and not a single schema for the entire application. Currently, the ids are incremental integers. In the future it would be beneficial to change these to GUID or UUID.[52]

6.1.3 Questions

Due to limited time, some algorithms and question types were dropped in favor of polishing the application. A natural step forward would be to make sure all the algorithms and data structures have their own question type. Some of the algorithms and data structures that were dropped:

1. Bellman ford
2. Heaps
3. Stacks
4. Hashtables
5. Lists

The application has been designed for the courses DAT200 and DAT110, and therefore if another course wants to use the application, it would be limited to text and multiple choice questions. A natural step is to add question types specific to other courses and finding a way to configure courses to limit what question types each course can use.

6.1.4 Unit-Tests

At the current state, the application does not have nearly as many unit-tests as it could have had. Due to limited time and resources writing unit tests became more of an afterthought in comparison to getting the server functionality working. The unit tests that the project currently has are unit tests for the different algorithms and data structures. These tests have been primarily used in order to check that the algorithm implementations work as they should. Areas in the application where there should be unit tests, but currently do not have any are in the Validation Checker and the Solution Checker residing on the server. In the future, it might be beneficial if some tests were made for each question type for both checkers. This would in turn help during debugging if any changes were to affect the checkers when implementing a new type of question.

6.1.5 End-To-End Tests

The developers of Cypress recommend setting data-attributes on the HTML elements that are going to be tested. This is to make it easier for Cypress to select and obtain the correct HTML element. This is normally a problem since the HTML elements tend to constantly change their attributes during development in a Vue application. Unfortunately, these data-attributes could potentially present a security risk if they were present in the production build. However, a method of removing them only in the production mode, while keeping them in testing mode was never discovered. This is something to keep in mind for the future development of the application. All the data-attributes used for the current tests have been well documented in a text file called `dataAttributes.txt` and can be found in the cypress directory.[53]

The future developers should also be aware that Cypress is intended to be used during development following closely the principles of test-driven development. This means that the test themselves need to be altered a lot to handle the changes done to the HTML elements on the site. Writing E2E-tests in Cypress is therefore recommended to be done early in development and not in the middle or at the end of the project.

6.1.6 Localization

When creating the locale files, it was designed in such a way that every Vue component has its own object. This has been a design that worked well, but during development, a design flaw was discovered. Some Vue components use the same text. This can, for instance, be seen on buttons and error messages. It could have been solved by not just storing locales divided into Vue components, but by also storing common locale in a separate object. Even though the file size for each locale file is relatively small, it can be reduced more. Another way to expand localization would be to add more locales.

6.1.7 GraphDrawer

If new question types are implemented which require the GraphDrawer to keep track of thousands of nodes at the same time, there might be performance issues on some systems. The nodes

are currently stored in an array. This means that if the array is searched for a node on the screen, all of the nodes need to be checked, even those on the opposite side of the world. Many operations starting on one node, and ending at a node close to the first node (e.g., selecting nodes by dragging the mouse) would be more performant if a spatial data structure[54] was used instead.

The current system is designed to work with both touch screens and cursors. Because of this, two choices were made to make the development simpler and less time-consuming.

1. When a user needs to enter text or numbers, the JavaScript alert popup is used because this works in all browsers. Some users with a keyboard would have been able to write faster if the GraphDrawer read the input directly. Mobile browsers do not allow a website to open the virtual keyboard from a script. If the mobile browsers change this in the future, a different method for retrieving the user input could be used. Some browsers allow the user to block popups. If this is done, the user can no longer enter text.

2. For simplicity, both desktop and mobile users control the GraphDrawer the same way. User testing can be done to find the gestures which feel the most natural. If the natural way to do something is different depending on the system, different handlers can be implemented.

Every time the world state changes, the canvas is redrawn. Changes outside the camera view should not make the GraphDrawer redraw the world. If only a small part of the world inside the camera view is changed, only that part needs to be redrawn. This has not been implemented because there have not been any performance issues with the current question types.

6.1.8 Python

If the Python question type needs to be expanded to include either more language features or other types of questions, the custom interpreter should not be used. Maintaining a Python interpreter which always matches the newest version of Python is very time consuming, and much knowledge about programming language development is needed. The CPython interpreter should, therefore, be used instead. If the interpreter is changed, the components related to Python questions need to be remade. Figuring out how to use the CPython interpreter and remaking the components is time-consuming, but it will be easier to maintain because all of the features are included and updated.

References

[1] Kahoot! (2019). Kahoot! — learning games — make learning awesome!, [Online]. Available: https://kahoot.com/ (visited on 06/05/2019).

[2] A. Deveria. (2019). Can i use... support tables for html5, css3, etc, [Online]. Available: https://caniuse.com (visited on 09/05/2019).

[3] w3schools. (2019). Html5 svg, [Online]. Available: https://www.w3schools.com/html/html5_svg.asp (visited on 07/05/2019).

[4] w3schools. (2019). Html5 canvas, [Online]. Available: https://www.w3schools.com/html/html5_canvas.asp (visited on 07/05/2019).

[5] N. Foundation. (2019). Node.js, [Online]. Available: https://nodejs.org/en/ (visited on 07/05/2019).

[6] i. npm inc. (2019). Npm, [Online]. Available: https://www.npmjs.com/ (visited on 07/05/2019).

[7] R. Dahl. (2019). Deno, [Online]. Available: https://github.com/denoland/deno (visited on 07/05/2019).

[8] N. Foundation. (2019). Express - node.js web application framework, [Online]. Available: https://expressjs.com/ (visited on 07/05/2019).

[9] M. D. G. Inc. (2019). Build apps with javascript — meteor, [Online]. Available: https://www.meteor.com/ (visited on 07/05/2019).

[10] D. C. Wilson. (2019). Node.js compression middleware, [Online]. Available: https://github.com/expressjs/compression#readme (visited on 07/05/2019).

[11] (2019). Gzip, [Online]. Available: http://www.gzip.org (visited on 07/05/2019).

[12] w3schools. (2019). The history object, [Online]. Available: https://www.w3schools.com/jsref/obj_history.asp (visited on 07/05/2019).

[13] B. Ripkens. (2019). Fallback to index.html for applications that are using the html 5 history api, [Online]. Available: https://github.com/bripkens/connect-history-api-fallback#readme (visited on 07/05/2019).

[14] D. C. Wilson. (2019). Parse http request cookies, [Online]. Available: https://github.com/expressjs/cookie-parser#readme (visited on 07/05/2019).

[15] J. Foundation and contributors. (2019). Mocha - the fun, simple, flexible javascript test framework, [Online]. Available: https://mochajs.org/ (visited on 07/05/2019).

[16] T. jQuery Foundation. (2019). Qunit: A javascript unit testing framework, [Online]. Available: https://qunitjs.com/ (visited on 07/05/2019).

[17] P. Labs. (2019). Jasmine, behavior-driven javascript, [Online]. Available: https://jasmine.github.io/ (visited on 07/05/2019).

[18] S. Motte. (2019). Dotenv, [Online]. Available: https://www.npmjs.com/package/dotenv (visited on 07/05/2019).

[19] J. Foundation and other contributors. (2019). Eslint - pluggable javascript linter, [Online]. Available: https://eslint.org/ (visited on 07/05/2019).

[20] J. Long and contributors. (2019). Prettier - opinionated code formatter, [Online]. Available: https://prettier.io/ (visited on 07/05/2019).

[21] PassportJS. (2019). Passport, [Online]. Available: http://www.passportjs.org/ (visited on 07/05/2019).

[22] E. D. Hardt. (2012). The oauth 2.0 authorization framework, [Online]. Available: https://tools.ietf.org/html/rfc6749 (visited on 07/05/2019).

[23] Uninett. (2017). Passportjs strategy for openid connect, [Online]. Available: https://github.com/Uninett/passport-openid-connect (visited on 07/05/2019).

[24] Uninett and Utdanningsdirektoratet. (2019). Feide — sikker innlogging og datadeling i utdanning og forskning, [Online]. Available: https://www.feide.no/ (visited on 07/05/2019).

[25] Okta. (2018). Oauth 2.0 and openid connect (in plain english), [Online]. Available: https://www.youtube.com/watch?v=996OiexHze0 (visited on 07/05/2019).

[26] O. Foundation. (2019). Welcome to openid connect, [Online]. Available: https://openid.net/connect/ (visited on 07/05/2019).

[27] Uninett. (2018). Feide documentation, [Online]. Available: https://docs.feide.no/ (visited on 07/05/2019).

[28] Socket.io. (2019). Socket.io, [Online]. Available: https://socket.io (visited on 07/05/2019).

[29] Mozilla and individual contributors. (2019). The websocket api (websockets), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (visited on 07/05/2019).

[30] G. Rauch. (2018). Engine.io: The realtime engine, [Online]. Available: https://github.com/socketio/engine.io (visited on 09/05/2019).

[31] SQLite. (2019). Sqlite home page, [Online]. Available: https://www.sqlite.org/index.html (visited on 07/05/2019).

[32] E. You. (2019). Vue, [Online]. Available: https://vuejs.org/ (visited on 07/05/2019).

[33] E. You. (2019). Vue-cli, [Online]. Available: https://cli.vuejs.org/ (visited on 07/05/2019).

[34] E. You. (2019). Vue-router, [Online]. Available: https://router.vuejs.org/ (visited on 07/05/2019).

[35] E. You. (2019). Vuex, [Online]. Available: https://vuex.vuejs.org/ (visited on 07/05/2019).

[36] Babel. (2019). Babel is a javascript compiler., [Online]. Available: https://babeljs.io (visited on 07/05/2019).

[37] Bootstrap. (2019). Bootstrap, [Online]. Available: Bootstrap (visited on 07/05/2019).

[38] BootstrapVue. (2019). Bootstrap + vue, [Online]. Available: https://getbootstrap.com (visited on 07/05/2019).

[39] Cypress.io. (2019). The web has evolved. finally, testing has too., [Online]. Available: https://www.cypress.io/ (visited on 18/04/2019).

[40] Cypress.io. (2019). Testing has been broken for too long., [Online]. Available: https://www.cypress.io/how-it-works/ (visited on 18/04/2019).

[41] T. S. Project. (2019). Test automation for web applications, [Online]. Available: https://www.seleniumhq.org/docs/01_introducing_selenium.jsp (visited on 17/04/2019).

[42] d. f. e. Wikipedia. (2019). Selenium (software), [Online]. Available: https://en.wikipedia.org/wiki/Selenium_(software) (visited on 17/04/2019).

[43] J. Colantonio. (2017). Cypress.io vs selenium test automation, [Online]. Available: https://www.joecolantonio.com/cypress-io-vs-selenium-test-automation/ (visited on 18/04/2019).

[44] A. contributors. (2019). One framework. mobile & desktop., [Online]. Available: https://angular.io (visited on 08/05/2019).

[45] F. Inc. (2019). A javascript library for building user interfaces, [Online]. Available: https://reactjs.org (visited on 09/05/2019).

[46] Mozilla and individual contributors. (2019). ¡canvas¿: The graphics canvas element, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas#Maximum_canvas_size (visited on 05/05/2019).

[47] voidvol...@gmail.com. (2014). Issue 339725: Canvas maximum size ¿32767px, [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=339725 (visited on 05/05/2019).

[48] T. P. S. Foundation. (2019). Cpython, [Online]. Available: https://github.com/python/cpython (visited on 17/04/2019).

[49] D. Inc. (2019). Enterprise container platform for high-velocity innovation, [Online]. Available: https://www.docker.com (visited on 09/05/2019).

[50] T. K. Authors. (2019). What is kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 09/05/2019).

[51] SQLite. (2019). Frequently asked questions, question 19, [Online]. Available: https://www.sqlite.org/faq.html#q19 (visited on 05/04/2019).

[52] P. Leach. (2005). A universally unique identifier (uuid) urn namespace, [Online]. Available: https://tools.ietf.org/html/rfc4122 (visited on 07/05/2019).

[53] Cypress.io. (2019). Best practices, [Online]. Available: https://docs.cypress.io/guides/references/best-practices.html (visited on 18/04/2019).

[54] L. Toma. (2008). Cs 340: Spatial data structures, [Online]. Available: http://www.bowdoin.edu/~ltoma/teaching/cs340/spring08/ (visited on 07/05/2019).

List of Figures

A Sorting algorithms

A.1 Insertion Sort

Insertion sort is the simplest sorting algorithm taught in the algorithms and data structures course. Insertion sort uses a simple algorithm that has to go through every entry in a list. The algorithm checks whether or not the current entry has a smaller value than the previous entry. If the entry has a smaller value, then the previous entries have to re-sorted with the entry until either an entry that is smaller is found or the start of the list is reached.

The insertion sort is simple to implement, only consisting of two loops. Because of its simple design, it is a sorting algorithm that is quite often used. Insertion sort is however quite slow compared to most other sorting algorithms. It has an $O(n)$ average run time and a runtime of $O(n^2)$ in terms of a worst-case scenario.

Insertion sort is quite effective when it is used to sort almost fully sorted lists. This means that a lot of other sorting algorithms uses the insertion sort algorithm. It is usually used as a final step in other sorting algorithms. Examples of this being the sorting algorithms Shell sort, Merge sort and Quick sort.

A.2 Shell Sort

Shell sort is a sorting algorithm that is based on insertion sort. The key difference between the two sorting algorithms is in performance. Shell sort will have both an average case and a worst case of $O(n)$ runtime. The main reason for this is that shell sort will sort entries in the list that are at certain intervals from each other. The algorithm is essentially dividing list entries into smaller sublists in order to make the sorting more efficient. Through this method, shell sort avoids having to check every entry in the list, which allows the algorithm to remove the main drawback of using insertion sort.

The starting interval used in a shell sort is usually based on approximately half length of the list. The algorithm uses the interval to divide up the entries in the list. The entries chosen together are then compared to each other and sorted based on their value. Once the list has been effectively sorted on the current k-interval, the interval's value is halved and the sorting process starts anew. Eventually, the interval will reach a value of 1, which causes shell sort to finish sorting the list using the insertion sort algorithm. Of course, since the list should at this point be almost completely sorted, and therefore the insertion sort should never achieve a run-time of $O(n^2)$.

A.3 Merge Sort

The merge sort algorithm is a divide and conquer sorting algorithm. The algorithm works by splitting the array into many smaller arrays, sorting the smaller arrays and then merging them back together into a sorted version of the original array. This project contains an implementation

which splits the array recursively until a given limit is hit, or the array size is 1. If the limit is given, the array will be sorted using insertion sort, before merging.

A.4 Quick Sort

Quicksort is an efficient comparison sort that uses a divide and conquer algorithm. The algorithm works by choosing a pivot point, then dividing the list into two lists and based on their value compared to the pivot point. All values less then the pivot point goes into a list usually named left, and all values equal or higher then the pivot point goes into a list usually called right. The pivot point can be chosen in many different ways, but the recommended way is to take three random indexes and take the median. After the list have been split into left and right, those are then sent into the same function where they are divided until there is only one value left. Then the value is returned and the list is concatenated left, pivot point and right.

B Tree Data Structures

B.1 Binary Tree

A binary tree data structure is a simple data structure whose structure resembles a tree. A binary tree consists of nodes each having a unique value. Each node has a reference to its parent node and its children nodes. The previous node that links to the current node is called a parent node. The child nodes are the next nodes in line after the currently selected node. A tree node can only have up to two children node, and every node must have only one parent node. The exception to this rule is the very first node in the tree, which is called the root node. There can only be one root node in the tree. Since a node can only have up to two children nodes, the child nodes are usually referred to as either the left child node or the right child node. A node that has no child nodes will always be at the bottom of the tree and are referred to as leaf nodes. The binary tree data structure is notably used for file systems.

B.2 Binary Search Tree

A Binary Search Tree (BST) tree is an evolved form of the Binary Tree, that is structured in a way that is more beneficial for storing large amounts of data. The criteria for a Binary tree to be a binary search tree is the following conditions:

- All left children nodes need to have a value lower than its parent node.
- All right children nodes need to have a value higher than its parent node.
- There cannot be any nodes with duplicate values in the tree.

Having these requirements helps the BST search for the correct node, since the time it takes to find the correct node is shortened tremendously. Inserting new nodes in the tree is also quite simple, just traverse the tree, where the direction is dependent on the new nodes value. Once a leaf node has been reached, put it as either a left child or a right child based on whether the new node's value is lower or greater than the lead node's value.

B.3 AVL

An AVL tree is a binary search tree, that can balance itself when it is needed. An AVL tree has only one additional condition compared to a normal BST. AVL trees can only have a height of one or lower difference between the left and right subtree in any of the nodes in the tree. This means for instance that a node that has a left subtree of height one and a right subtree of height three is not a qualified subtree. In order to keep this height balance between the subtrees, the AVL tree has to rebalance itself by rotating in either the left or right direction.

There are four different rotations that can be performed:

- Left
- Right

- Double Left, also referred to as right left rotation
- Double Right, also referred to as left right rotation

C Other Data Structures and Algorithms

C.1 Dijkstra

Dijkstra's algorithm is an algorithm that is used to determine the shortest path through a network of nodes (graph) connected by a nonnegative weighted edge. The algorithm has the start and the end node and works its way through the network keeping track of the current route weight to each node in a table. The algorithm keeps a list of processed nodes and a list of unprocessed nodes. For each iteration, it takes the node from the list of unprocessed nodes with the shortest path and processes it. Then it looks for edges and their weight and updates the table. This is done until all paths to the end node has been explored. When the algorithm is finished, the shortest path has been found.

D Python

Python is a dynamically-typed object-oriented programming language developed by The Python Software Foundation. Dynamically-typed means that it is not necessary to specify the type of variables, because the type is determined at runtime. The syntax is very similar to JavaScript; however, Python uses indentation instead of curly brackets to separate code.

Classes are defined using the `class` keyword. Functions are defined using the `def` keyword. Methods (functions inside classes) are defined as functions inside a class with the first argument begin a reference to the instance of the class. A class constructor can be added by defining a method with the name `__init__`. Example of a class:

```
# This is a comment
class Person:
    def __init__(self, first_name, last_name, age):
        self.age = age
        self.setName(first_name, last_name)

    def setName(self, first, last):
        self.name = first + " " + last

p = Person("C", "H", 88)
```

E User manual

E.1 GraphDrawer

E.1.1 Graphs & Trees

To add a node:

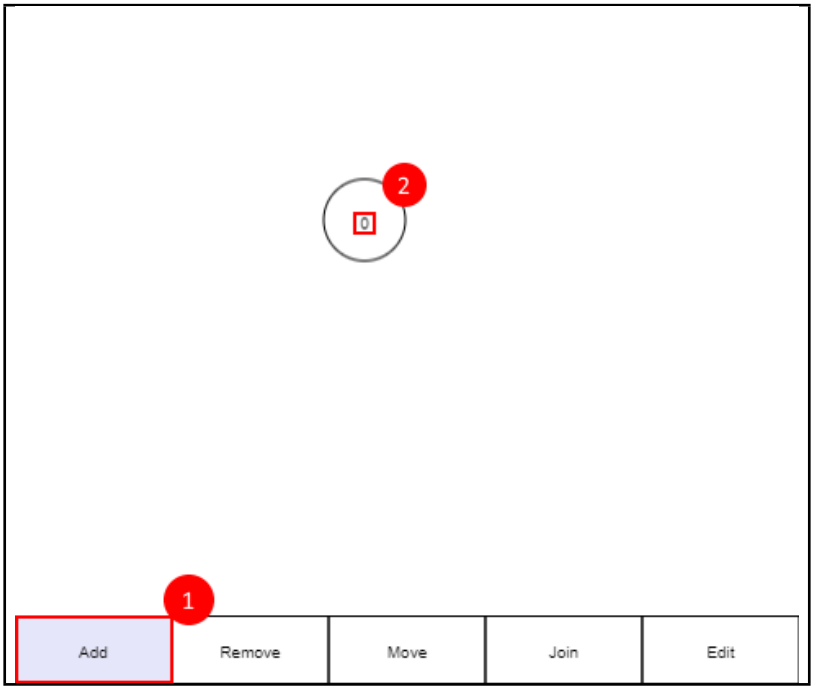


Figure E.1

Step I. Click the "Add" button (1). (Figure: E.1)

Step II. Click the position where you want the node to be (2). (Figure: E.1)

To remove a node:

Step I. Click the "Remove" button.

Step II. Click the node.

To move a node:

Step I. Click the "Move" button.

Step II. Press the mouse button while the cursor is inside the node.

Step III. Move the cursor to the new position.

Step IV. Release the mouse button.

To join two nodes:

Step I. Click the "Join" button.

Step II. Press the mouse button while the cursor is inside the first node.

Step III. Move the cursor inside the other node.

Step IV. Release the mouse button.

To edit the value of a node:

Step I. Click the "Join" button.

Step II. Click the node.

Step III. Type the new value in the prompt.

Step IV. Click the "OK" button.

E.1.2 Sorting

To select nodes in an array:

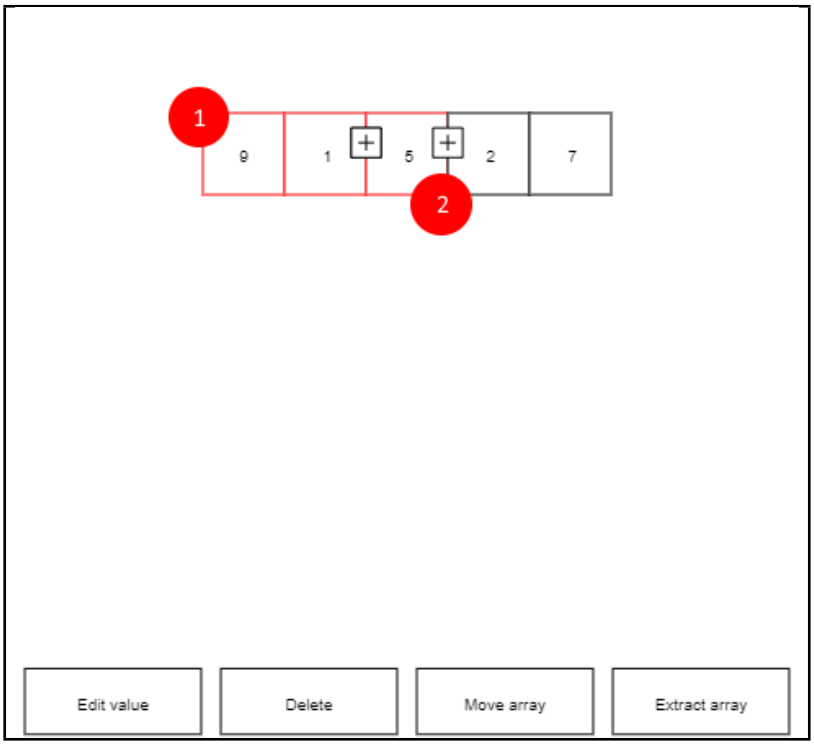


Figure E.2

Step I. Press the mouse button while the cursor is inside the first node (1). (Figure: E.2).

Step II. Move the cursor over the other nodes (2). (Figure: E.2)

Step III. Release the mouse button.

To edit the value of a node:

Step I. Select the node.

Step II. Click the "Edit value" button.

Step III. Type the new value in the prompt.

Step IV. Click the "OK" button.

To delete a node:

Step I. Select the node.

Step II. Click the "Delete" button.

To move an array:

Step I. Select a node in the array.

Step II. Click the "Move array" button.

Step III. Move the cursor to the new position.

Step IV. Click the mouse button to move the array.

To extract nodes from an array:

Step I. Select the nodes.

Step II. Click the "Extract array" button.

E.1.3 Dijkstra

To link two nodes:

Step I. Move the cursor inside the first node.

Step II. Press the mouse button.

Step III. Move the cursor inside the other node.

Step IV. Release the mouse button.

To undo or redo an action:

Step I. Click the "¡– Undo" button to undo an action.

Step II. Click the "Redo –¿" button to redo an action.

E.1.4 Python

Add Variable:

Step I. Click the "Add Variable" button.

Step II. Click the position where you want the variable to be.

Step III. A prompt for writing the variable name will appear.

Step IV. Click "OK" after typing the name.

Add Objects:

Step I. Click the "Add Object" button.

Step II. Click the position where you want the object to be.

Step III. A prompt will ask you for the object type.

Step IV. Click "OK" after writing the type name.

Step V. If you picked a reference type object, the object is added.

Step VI. If you picked a value type, another prompt will ask you for the value.

Step VII. Click "OK" after entering the value.

Link variable to object:

Step I. Click the "Join" button.

Step II. Press the mouse button while the cursor is inside the variable.

Step III. Move the cursor inside the object.

Step IV. Release the mouse button.

Link object to object:

Step I. Click the "Join" button.

Step II. Press the mouse button while the cursor is inside the property object.

Step III. Move the cursor inside the other object.

Step IV. Release the mouse button.

Step V. A prompt will ask you which property you want to link the object to.

Step VI. Enter the property name and click "OK".

Remove:

Step I. Click the "Remove" button.

Step II. Click a node or edge to remove it.