Implementation of an Application for Analyzing and Visualizing Benchmark Results for Optimization Solvers

Oscar Härtull
Master's Programme in Information Technology: Computer Engineering
Supervisor: Andreas Lundell
Faculty of Science and Engineering
Åbo Akademi University
Vasa, Finland 2023

Abstract

This master's thesis presents the development of a system for analyzing and visualizing benchmarking results for optimization solvers, with the goal of creating a portable and maintainable application where one can view these results in tabular and plotted formats. The system requirements, development planning, implementation, and testing of the application are covered in each chapter.

Based on the requirements created for the system, the development process consisted of creating an offline web application using open-source libraries to allow the user to interact with the benchmarking data, by performing actions such as filtering, sorting, and hiding results according to their needs. After the implementation phase, manual, unit and user interface tests were performed on the entire application. This ensured that the system worked properly and met the requirements criteria. In addition, the unit and automated user interface tests could be used to verify that further development was working as intended.

JavaScript-based libraries were used for both implementation and testing, with the application written in TypeScript and the static module bundler Webpack used to transpose the source code to JavaScript. Bootstrap, DataTables.js, and Chart.js were used for the user interface, while Jest and Playwright were used to test the functionality of the system.

This thesis gives an overview of how to develop a portable system using web technologies and what was learned from doing so. As a result, the created application meets the system requirements and can be used as a research support tool with the possibility of easily extending its features.

Keywords: Visualization, benchmark results, web development, optimization solvers.

Abstrakt

Denna magisteravhandling presenterar utvecklingen av ett system för att analysera och visualisera benchmarkingresultat för optimeringslösare, med målet att skapa en portabel och underhållbar applikation, där man kan se dessa resultat i både tabell- och grafformat. Systemkraven, utvecklingsplaneringen, implementeringen och testningen av applikationen behandlas per kapitel.

Baserat på de krav som ställts på systemet bestod utvecklingsprocessen av att skapa en offline-webbapplikation med hjälp av öppen källkods-bibliotek så att användaren av applikationen kan interagera med benchmarkingdata och använda funktioner som att filtrera, sortera och dölja resultat efter sina behov. Efter implementeringsfasen utfördes manuella tester, enhetstester och användargränssnittstester av hela applikationen. Detta säkerställde att systemet fungerade korrekt och uppfyllde kravkriterierna. Dessutom kunde enhets- och användargränsnittstesterna användas för att verifiera att programmet vid vidareutvecklingen fungerade som avsett.

JavaScript-baserade bibliotek användes för både implementerings- och testfaserna, med applikationen skriven i TypeScript och Webpack som kompilerar källkoden till JavaScript. Bootstrap, DataTables.js och Chart.js användes för gränssnittet, medan Jest och Playwright användes för testning.

Denna avhandling ger en översikt av hur ett portabelt system kan utvecklas med hjälp av webbteknik, och lärdomarna som erhållits vid genomförandet. Resultatet är en applikation som uppfyller systemkraven och kan användas som ett stödverktyg vid forskning och med möjlighet att enkelt utöka dess funktioner.

Nyckelord: Visualisering, benchmark resultat, webbutveckling, optimeringslösare.

Preface

Despite the first impression the application gives, a minimalist and clean interface, it has been a long and practical project that has diverged in many different directions, sometimes even expanding from its original goals, with many features being added or replaced. It was challenging to build something from scratch with free hands, and the quirks and oddities that came with that responsibility, but also rewarding to plan according to my own ideas.

In a way, this project has lived its own little life alongside my studies and work, always present and reminding me that something needs to be addressed or developed. A sense of wistfulness began to creep in as the final pieces were added and the project began to reach its finish line.

I would like to thank my supervisor, Andreas Lundell, for giving me a lot of freedom in the design and development of the project. In addition, with constructive and valuable feedback regarding the overall user experience.

Finally, I would like to also thank my partner, family, and friends for the support.

Table of Contents

List of Figures

List of Tables

List of Source Codes

1. Introduction

When evaluating and comparing benchmarking results, it is important to be able to analyze and visualize the data. Dealing with detailed and extensive static tables and plots containing the results is a tedious task, as one cannot obtain a good overview of the data categories of interest if the data presented consists of irrelevant categories for the current use case. In the scenario of this thesis, the benchmark data comes from solving mathematical programming optimization problems. Currently, tools such as PAVER 2.0 [50] can be used to visualize and summarize this benchmark data [2], or Mittelmann-Plots [32] can be used to compare results via interactive charts. However, using these tools and inspecting results may be challenging. In the first alternative, the installation of the tool requires specific versions of Python and its libraries, and the generated visualization and data comparison of the results are static. For instance, there is no built-in functionality that makes it easier for the end user to highlight the data they need from the summarized tables, such as filters to select results from a specified value interval. A system with features for manipulating the summarization and visualization of the data paves the way for interacting with it in a way that enables an end user to select those results that they want to focus on, and which are important to them. This thesis focuses on the implementation of a system to assist the end user with the ability to manipulate benchmarking data.

The main objective of this project is to develop an application with accessible controls and features that allows the end user to interact with and visualize the benchmarking data, hence the application name VIVA was created from the key features: visualize, interact, verify, and analyze. By creating the data summarization and visualization tool as a self-contained web application, it will be accessible to anyone with a web browser, unlike traditional desktop applications, where the underlying software or hardware may affect accessibility. The result of the developed application is shown in Figure 1.1. The development process will need to consider issues such as usability, implementation, and maintainability. The process of importing and transforming benchmarking data should be as flawless as possible, and the need for any third-party packages should be kept as low as possible,

to reduce maintenance requirements.

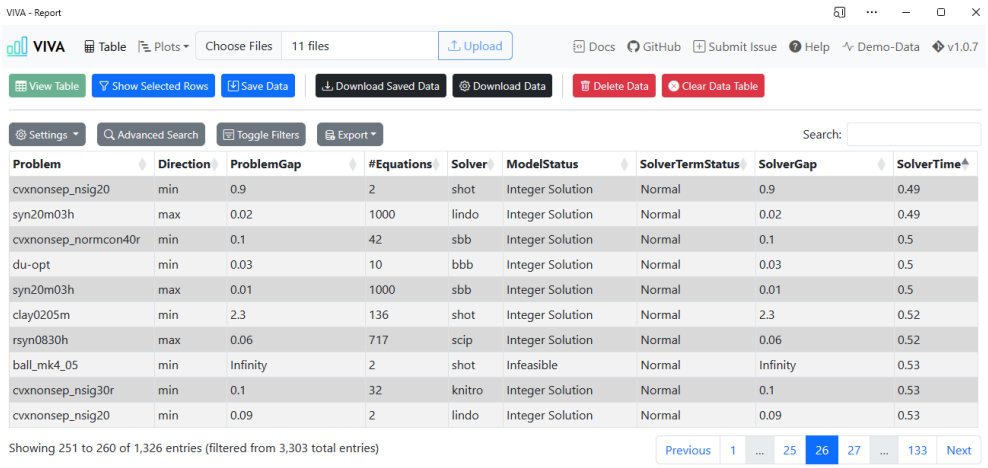


Figure 1.1: The developed application installed as a progressive web application.

1.1 Background

Mathematical optimization is used to find the best solution of a mathematical model, which represents an abstraction of a real-world problem. The problems can be found in the energy market, supply chain, production and logistics, or financial portfolio optimization, to give a few examples. A model can be composed of data, variables, constraints and the objective to solve the problem. These models often result in structured problems such as linear programming problems, mixed integer linear programming problems, nonlinear programming problems, and mixed integer nonlinear programming problems [15].

Systems such as General Algebraic Modeling System [10] (hereafter GAMS) translates the optimization problems to computer code, which can be solved by software. Solvers for these optimization problems exist either with open source or commercial licenses. For instance, SHOT [40] (Supporting Hyperplane Optimization Toolkit) and SCIP [38] for the first type, and Gurobi Optimizer [14] and IBM ILOG CPLEX Optimizer [5] for the latter type.

Benchmarking helps to highlight the difference between the solvers, validating algorithms, and test their scalability [44]. The test sets for benchmarking can be attained from problem libraries. Example of these are MINLPLib with small-scale literature models to large-scale real-world models [3] which currently hosts 1595 instances [34], and the MIPLIB 2017 library, including both a benchmark and a collection set, with 240 respectively 1065 instances [12].

1.2 Benchmarking Result Files

For this project, the benchmarking result files are in a so-called trace file format and contain data about the GAMS job and the solve statements, together with a header that defines the contained trace records and the associated trace record fields. The headers for these trace files can be modified according to the needs required [11]. These files are what the system under development will use as the default source files of the benchmark results.

Table 1.1: Data known from the trace file if no header is specified.

Header (trace file)	**Renamed Header**	**Description**
InputFileName	Problem	Problem instance name
ModelType	ModelType	Type of the model
SolverName	Solver	Name of the solver
NLP def		Default NLP solver
MIP def		Default MIP solver
JulianDate		Start day or time of job
Direction		Type of optimization process
NumberOfEquations	#Equations	Total number of equations
NumberOfVariables	#Variables	Total number of variables
NumberOfDiscreteVariables	#DiscreteVariables	Total number of discrete variables
NumberOfNonZeros	#NonZeros	Number of nonzeros
NumberOfNonlinearNonZeros	#NonlinearNonZeroes	Number of nonlinear nonzeros
OptionFile		Value corresponds to which option file is used
ModelStatus		GAMS model return status
SolverStatus		GAMS solver return status
ObjectiveValue	SolverPrimalBound	Lower bound on the optimal value
ObjectiveValueEstimate	SolverDualBound	Upper bound on the optimal value
SolverTime		Resource time used
NumberOfIterations		Number of solver iterations
NumberOfDomainViolations	#SolverIterations	Number of domain violations
NumberOfNodes	#SolverNodes	Number of nodes used
UserComment		User comments.

Based on the current run statistics that PAVER uses to generate table and chart data, the system to be built will use these same parameters [41] from the trace files as the basis for its result categories. These parameters can be declared as default headers to facilitate the file import and data processing steps. Trace files that do not follow this order of headers must include them at the beginning of the file in a comment format. Table 1.1 provides a summary of the run statistics that will be known and how they will be renamed on the table page. Listing 1.1 gives an example of the file structure from `shotALL.trc` [19], where nvs11, nvs12, and nvs15 are the optimization problems.

```
nvs11,MINLP,shot,conopt,CPLEX,43381.94718,0,4,4,3,16,3,1,8,1,-431,-431,0.044558615,0,0,0,#
nvs12,MINLP,shot,conopt,CPLEX,43381.94718,0,5,5,4,25,4,1,8,1,-481.2,-481.2,0.04748436,0,0,0,#
nvs15,MINLP,shot,NONE,CPLEX,43381.94718,0,2,4,3,10,3,1,8,1,1,1,0.071205586,0,0,0,#
```

Listing 1.1: Extract from `shotALL.trc`.

The direction indicates whether an instance is of a minimization or maximization type, and based on the value in the trace files, either 0 (min) or 1 (max). In a minimization instance, an upper bound on the optimal value of a solution is referred to as primal bound and the lower bound of the optimal value as dual bound. Similarly, in a maximization instance, the primal bound is the lowest limit on the best possible outcome and dual bound is the upper limit of the optimal outcome.

Since PAVER has expanded the result fields with calculations, these will also be implemented in the application, along with some new computations. Table 1.2 shows the data fields that are calculated based on the trace file content.

Table 1.2: Data fields to be added.

Name	**Header**	**Description**
Gap (Solver)	SolverGap	Gap between the primal and dual bound for a problem instance solved by a certain solver
Gap (Problem)	ProblemGap	Gap between best known primal and dual bounds for a problem instance if these are known
Primal gap	SolverPrimalGap[%]	Gap between primal bound of the solver and best known value of the instance
Dual gap	SolverDualGap[%]	Gap between dual bound of the solver and best known value of the instance

The gap is computed as the difference between the primal and dual bounds for the solver, according to the following formula, where the tolerance (tol) by default is 10^{-9}:

$$\text{gap}(a,b) = \begin{cases} 0, & \text{if } |a-b| < \text{tol}, \\ \infty, & \text{if } \min(|a|,|b|) < \text{tol}, \\ \infty, & \text{if } \max(|a|,|b|) > \infty, \\ \infty, & \text{if } ab < 0, \\ \frac{a-b}{\min(|a|,|b|)}, & \text{otherwise.} \end{cases}$$

The gap is zero if the absolute difference between a and b is smaller than the tolerance value. If the minimum of the absolute values of a and b is less than the tolerance, the gap is infinite. It is also infinite if the maximum of the absolute value of a and b is greater than infinity, or if the product of a and b is less than zero. Otherwise, it is the difference between a and b, divided by the minimum of the absolute values of a and b [18]. The gap for the problem follows the same formula, using its best known primal and dual bounds, while the primary and dual gaps are calculated for the primal bounds with the solver and the problem, and the dual bounds with the solver and the problem, respectively.

Additional information for the instances, such as properties and the best known primal and dual bounds for each instance will add more data fields, if the user chooses to load files containing this metadata. These files can be downloaded from MINLPLib [33]. Their structure and parsing are presented in Chapter 4.4.3.

1.3 Thesis Structure

The next chapter of the thesis explores the initial requirements for this system, detailing the needs that must be met. Planning of the underlying technology for the realization of the project, the design of the user interface (hereafter UI) and the fulfillment of the requirements are dealt with in the third chapter. Practical implementation follows in the fourth chapter, where the entire development process is discussed in detailed steps. The subsequent chapter is dedicated to the testing of the application. An overview of the completed application and its usage instructions will follow. The final chapter concludes the thesis with comments on how the project was carried out and the lessons learned throughout the process of creating the application. Lastly, a Swedish summary of the thesis is featured.

2. System Requirements

The system requirements will provide an indication of what is needed for the successful implementation of the project. These requirements are created based on the established goal of the project: the system must be able to manage data dynamically and it should be easy for the end user to start the system. The requirements can be divided into three different categories: technical, functional, and non-functional.

2.1 Technical Requirements

With reference to the introduction and the basic idea behind the system, the system will be a web-based application and thus easy to use for the end user, as there would be no need to depend on any software other than a browser to run it. Therefore, JavaScript or some derivative or variation of it must be used to handle the logic. The markup languages HTML and CSS would be used to structure and design the UI of this system.

2.2 Functional Requirements

To satisfy the functional requirements, one can assume what a user needs to do in the system to achieve his goal, which in this case is to interact and visualize the results. Since these functional requirements must be met in order to use the application at all, they will be extensively tested through both automated and manual testing, as discussed in Chapter 5. They are categorized into data import and export (I), tabular data presentation (II), and graphical data representation (III).

I. i. Selecting files The user should be able to select a benchmark result file in either a trace or JSON format, since we are limiting usage to those. If the result file is of the trace type, it should be possible to upload multiple files, provided they have the same header order. The data will take on the same form as the trace file in the JSON in its dataset value. This is accomplished

by assigning each element as a string value that contains all the comma-separated values for each row.

Other file extensions, such as CSV and solution files, should also be uploadable, as these formats are used for metadata, for instance, containing additional information about the properties of the instances and the best known primal and dual bounds for each instance.

I. ii. Confirming the upload After selecting a compatible file, the user should be able to confirm that they want to upload the file and proceed.

I. iii. Showing error message on a failed upload If the upload fails, the user must be notified with an error message describing why the upload failed and offering suggestions for resolving the problem.

I. iv. Showing success message on a non-failed upload If no problems occur during the upload, the end user should be notified that the upload was successful.

I. v. Downloading the data The data presented in the tables and graphs must be available for download in appropriate formats, both as complete results and as filtered results. In addition, users should be able to save the visualized data on the chart pages an image.

II. i. Sorting For the different result categories, it should be possible to sort the result columns in ascending or descending order.

II. ii. Filtering per column Filters should be applicable per column, where one can choose which results should be visible depending on a selected parameter.

II. iii. Paginating If there are many rows of data, table pagination could make it easier to navigate between results by breaking them up and putting them on separate pages.

II. iv. Searching in the displayed data The search in the result table should return only matching results. The search process would be applied to all rows for each result column. Ideally, it would be great to have a more refined search function that supports logical operators, i.e., applying negative terms.

II. v. Selecting results manually The user should be able to select specific solvers and problems to display only those and their corresponding data. The selec-

tion could be a combination of checkboxes and row selection. Depending on the structure of the result file, the filter options could be generated in different ways.

II. vi. Saving modified data in the system It should be possible to save the filtered and sorted data from the table for the current session by saving it to the browser's local cache. A notification would inform the user that the action was successful. This modified data could then be used on both the table and plot pages.

III. i. Visualizing The user should be able to create different charts from the uploaded file. The chart types would depend on the type of data being visualized, but it is expected that at least line and bar charts will be required.

2.3 Non-Functional Requirements

The following design features should be considered as non-functional requirements: maintainability, portability, compatibility, and usability.

Maintainability The source code should be maintainable, so that further work and updates can eventually cover everything that PAVER provides in the form of charts and statistical tables, as well as new functionality not present in it. To achieve this, the code needs to be well structured and documented, and automated testing would be required to warn if new or revised features break previous implementations.

Portability The application must have minimal setup complexity to be used without relying on separate installation of external packages.

Compatibility The application UI and user flow must be the same regardless of the choice of browser.

The data and its structure in the user-uploaded trace files should be compatible with the system functions that compute the new data fields to be added, discussed in Chapter 1.2.

Usability Since the main objective of the system is to read in a benchmark results file provided by the user, the process of uploading this file should be intuitive by having elements on the page that indicate the steps to be taken to succeed. The user should also be informed about which file types can be used and

whether there are any formatting restrictions. In addition, tooltips should be provided to guide the user through the functionality of buttons and fields.

Support for dynamic scaling provides the ability to display results on a wide range of devices, all with different resolutions and aspect ratios, as it is important that the UI can easily display many columns due to the number of headers the result files will have. Scrolling horizontally through a table should not be an unpleasant experience.

When the user interacts with the tabular data, such as selecting rows or columns, the selected data should be visually highlighted in a format that implies that the data has been selected.

3. Development Approach

This planning phase, as detailed in this chapter, establishes the framework for the project by considering the selection of tools and libraries to fulfill system requirements and determining the necessary structure for completing this development phase. The project setup is also briefly outlined prior to the subsequent implementation chapter.

3.1 Design

A CSS framework will be used to support the development and design of the UI for the application. This is a library with predefined styling and built-in responsive elements, and it will address the previously mentioned non-functional requirements of dynamic scaling, readability, and compatibility. Therefore, Bootstrap [1] will be a suitable CSS framework for this application, as it has a wide range of components, considerable documentation, and it is also an established choice when developing websites [7]. In addition, as other third-party libraries often use this framework, there is no need to spend time styling the components of such libraries utilized in the project.

Since the main purpose of the application is to obtain a better overview of the benchmark results, the design will result in a modest look. A navigation bar gives the end user direct access to the tabular data page and the different types of charts that can be applied to the results. The main page would be set to the tabular data view and the navigation bar and would link to the different chart pages.

Based on the functional requirements, the end user would need access to the following key components on the tabular data page: a file selector, a load button, a view table button, a button to save the data, a button to download the data, a button to delete the data, and a button to clear the data table and its applied filters, and a component for applying a selection of instances. As for the chart pages, most of the buttons would have the same functionality, except for the buttons that show the table and clear it. Instead, there would be an option to download the

chart data. To emphasize the functionality of these buttons and other elements, icons will be applied to them from the Bootstrap icons library.

3.2 Complying with the Requirements

In order to meet all the requirements as a whole, a system for displaying the results based on HTML and JavaScript was chosen. As discussed in the introduction, an advantage of using this as the basis for the system is that it will be easy to use and set up for the end user, as keeping it within a browser environment limits the need for third party tools to use the system.

From a development perspective, HTML is a very flexible format for including large amounts of different content, as the system will have tables and graphical visualization. It is also easy to customize and style the UI using CSS and the number of libraries surrounding web development is large. Therefore, all the functional and non-functional requirements should be feasible.

3.3 Exploring PWA Compatibility and Support

Progressive Web Application [52], shorted PWA, is a web application that provide features such as offline functionality and can give a website a standalone user experience as they run in a standalone window. Because the requirements of a PWA intersect with the system requirements that our application will have, it is easy to include support for this application type, without spending too much additional development time. The responsive design and cross-browser compatibility is already ensured by using popular libraries. What is needed is two key components: a manifest, containing instructions for the browser regarding the appearance and behavior of the application, and a service worker, which is essentially acting as a bridge between the browser and network. The development of this service worker is discussed in more detail in Chapter 4.8.

A requirement for a web app to be installable is that the pages are served over HTTPS, or alternatively, over localhost. Therefore, if the project will be publicly hosted on a server, the installation would be quite simple. If not, the end user must host the page files oneself, for example, by running a local server and serving the pages there. [24]

Google Lighthouse [16], an automated tool for auditing the quality of web pages, can provide a report of whether the implementation of the PWA support is accord-

ing to best practices and standards. By addressing the issues highlighted, it will guarantee that the application usage will be a better experience.

As for our application, the manifest members we need, seen in Listing 3.1, are the `name`, `icons`, `start_url`, and `display` members, where the remaining members are optional. The `start_url` points to the URL that will be opened when the application is launched. `Display` controls the UI elements visible to the browser. `Orientation` sets the default orientation, which will be landscape. The `theme_color` sets the default theme of the browser UI and the `background_color` sets the splash screen color.

```
{
  "name": "VIVA",
  "description": "Benchmark data analysis for optimization solvers.",
  "icons": [
    {
      "src": "Src/CSS/tab_icon.png",
      "type": "image/png",
      "sizes": "512x512",
      "purpose": "any"
    },
    {
      "src": "Src/CSS/icon_maskable.png",
      "sizes": "512x512",
      "type": "image/png",
      "purpose": "maskable"
    }
  ],
  "start_url": "/report.html",
  "display": "minimal-ui",
  "orientation": "landscape",
  "theme_color": "white",
  "background_color": "white"
}
```

Listing 3.1: Overview of `manifest.json` file.

3.4 Choice of Tools

Initially, it is to be decided which tools to use to meet the specified requirements and facilitate the development process, like formatting and testing. From the core of the project, it is necessary to have Git as the version control software to keep track of code and file changes.

Table 3.1 gives a summary of all the tools chosen for this project. Since we have already determined that the system will be implemented to run in a web browser,

we can conclude that JavaScript will be used to handle the logic of the system, but it can be written in TypeScript [43] to ease the development process. It adds functionality such as type annotation and structuring mechanisms for code, which will improve the maintainability of the codebase. Webpack [51] takes care of transpiling and packaging the TypeScript files into a bundle. Because of this structure, one would be able to run it in a browser without using a server to host it or violating cross-origin resource sharing rules [21]. Math.js [20] is an extensive mathematical library that will handle the calculations required for the graphs and their corresponding statistics tables. Jest [31] will be used for unit testing that the functions that perform the calculations are correctly made, whereas Playwright [35] will be used to assert that the UI is working as intended. The CSS framework discussed in section 3.1 led us to use Bootstrap and Bootstrap-Icons for this project. They contain many components and will maintain a consistent look and feel throughout the system. TypeDoc [42] was chosen to document the project's code to aid in the maintenance and future development of the project. It provides documentation generation of variables and functions in the system source code. ESLint [6] will be used for analysing this code and its patterns during development. Lastly, Gulp.js [13] was selected to automate the packaging of the necessary files for creating a release version of the system.

3.4.1 Table Library

The main reason for using a JavaScript table library is to enhance the data visualization and interaction of a table, without having to develop the implementation from scratch, as this would be a time-consuming task that lies beyond the scope of this project. As a result of using a table library, the user experience will be improved when features such as filtering, sorting and pagination are available, as these are likely to be used most often. More advanced features, such as exporting the table to various formats, are also desirable.

Most of the wanted features are all available in a variety of JavaScript table libraries. Hence, choosing the table library for this project depends on how easy it would be to deploy and how long there is support available for it. Presumably, an update or a switch to another library would be smooth since all libraries are deployed in a similar way. It is unlikely that any major web reforms would affect support for the system in the long term.

DataTables.js is a JavaScript table library with a rich set of plugins to enhance the interactivity of table components. It can be applied as a wrapper around a

predefined HTML table to provide features such as sorting. It is compatible with a variety of styling frameworks. The modular system of DataTables.js allows the user to tailor the installation to the property packages that will be used. Some features that would be beneficial to use in the system are available as extensions, for instance FixedColumns, which can fix one or more columns to a horizontal scrolling table. This will be needed, as our tabular data will possibly span over a large number of columns. The styling framework for the system will be Bootstrap 5, for which DataTables.js has a separate compatible styling package.

The documentation is very comprehensive for DataTables.js [39], which makes it easy to implement. In addition, it can help to ensure that any future problems can be dealt with conveniently. Choosing this library for our system guarantees in such cases that the requirements that have been set will be met.

3.4.2 DataTables.js Extensions

Thanks to the extensive number of packages available for our chosen table library, numerous features can be added. These will enhance the user interaction experience when inspecting the benchmarking data presented in a tabular format.

datatables.net-bs5 This package applies Bootstrap 5 styling on the DataTables.js table and the external components belonging to the library.

datatables.net-buttons-bs5 This package contains functionality for the buttons, such as setting the number of visible rows, selecting the column visibility, toggling filter panes, toggling the advanced search builder, exporting the table data by printing or downloading it as a csv file, and copying it to the clipboard.

datatables.net-fixedcolumns-bs5 By using this package, the first column containing the problem name can be fixed in place to the left of the table, which will be useful due to the long horizontal length of our table, as the number of data fields will be large.

datatables.net-searchbuilder-bs5 This search builder will give the user options to create search queries that filter results based on the given value ranges and set conditions.

datatables.net-datetime Extends the search builder with a date picker option.

datatables.net-searchpanes-bs5 These search panes enable filtering options per

column, so that the user can easily pick specific values to filter.

datatables.net-select-bs5 This package enables the user to select rows, which can subsequently be saved as a new dataset.

3.4.3 Charting Library

Since the system should also be able to visualize benchmark data using charts, it is important to consider a library that meets the needs and has the longevity to not be deprecated in the near future, as the goal is to develop a long-term stable system that can be easily updated. As with table libraries, there are many open-source JavaScript chart libraries to choose from.

Key factors are documentation and customization, sorting and filtering. Performance when handling large benchmark results should not negatively affect the user experience. Thorough documentation will ease the implementation process and make it easier to keep the system up to date. To interact with the charts, there would be options to filter out the results we want to see, dynamic scaling of axes according to value limits, and some form of export of the charts, presumably as an image or other file type.

Based on these factors, we selected Chart.js [4]. This library stood out due to its extensive documentation, customization options, high performance, and interactive features

Table 3.1: Tools and libraries used in the project.

Usage	Name	Description
Programming language	TypeScript	Typed superset of JavaScript
Dependency manager	Node.js	Installation and usage of libraries
Code documentation	TypeDoc	Source code documentation
Code analysis	ESlint	Ensuring good code patterns
Code formatting	Prettier	Ensuring good code formatting
CSS framework	Bootstrap	Frontend toolkit
CSS Icons	Bootstrap-Icons	Frontend icons
Charting library	Chart.js	JavaScript charting library
Table library	DataTables.js	Plug-in to add features to tables
Math library	Math.js	JavaScript computation library
Testing library	Jest	JavaScript unit testing library
UI testing library	Playwright	JavaScript testing library
Module bundler	Webpack	Compilation and bundling
Release bundler	Gulp.js	Creating a release build

3.5 Project Structure

The structure of the project is intended to follow the standard for web application development. An overview of the project's source directory can be seen in Figure 3.1. The root level folders are `Dist` (distribution), `Docs` (documentation), `Src` (source), and `Tests` (tests). The bundled TypeScript files are available in `Dist` as a single JavaScript file, together with a bundled CSS file. The documentation and associated images are placed in the `Docs` folder, TypeDoc will also use this folder as the default output folder. All TypeScript files, CSS files, and non-landing pages are located in `Src`. Test files for function and UI testing are placed in the last folder, `Tests`. The landing page, `report.html`, is on the root level along with the `service.worker.js`. The JSON configuration files are also located here, for easy development access, even if they are not needed for running the completed application. The service worker will cache the files, which have been filtered for release, to make them usable in offline mode if the application is installed as a PWA.

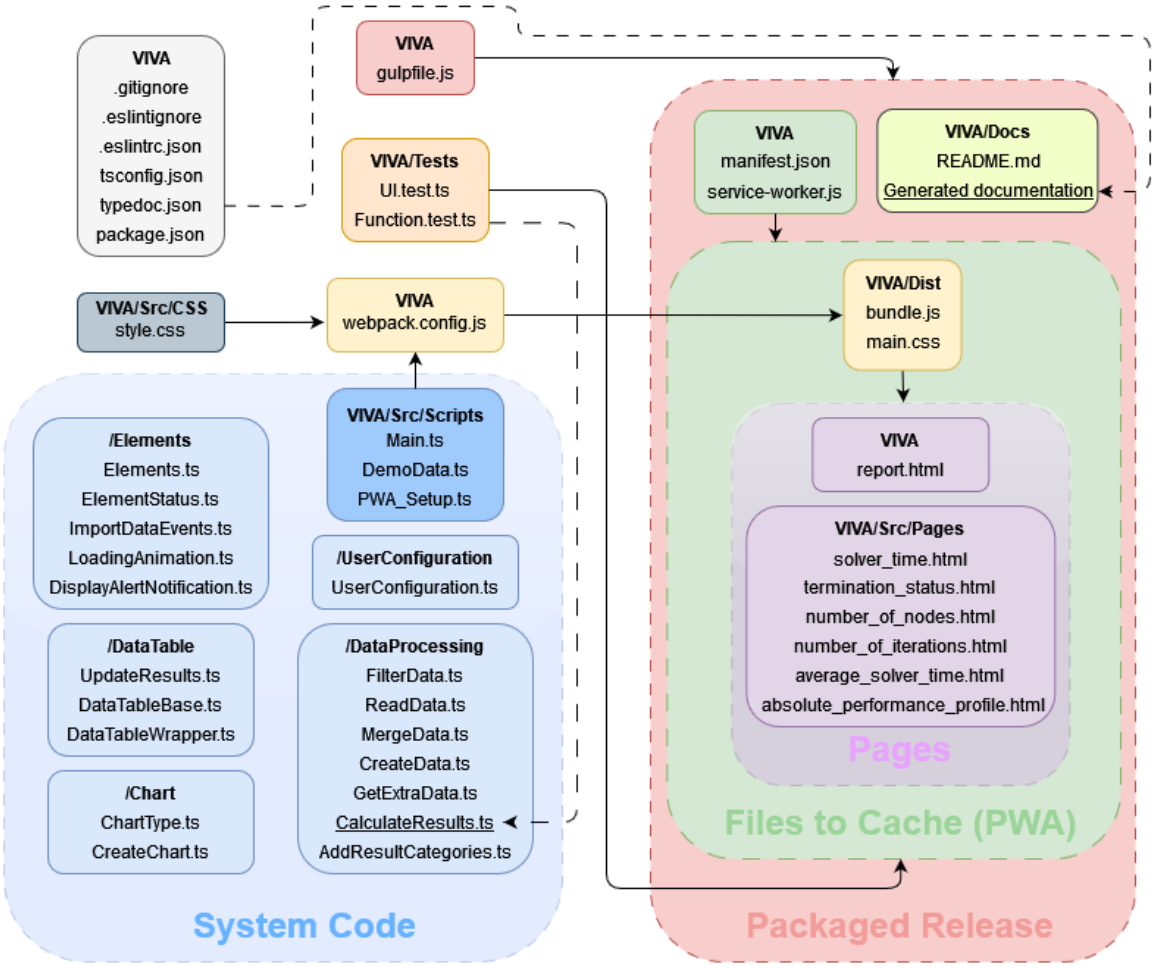


Figure 3.1: Structure of the source code.

3.6 Project Configuration

The following subsections will be dedicated to the configuration of the essential npm scripts and dependencies that streamline and automate aspects of the development process. Setting them up properly will ensure that the codebase maintains high quality. The tools that will be configured are ESLint with Prettier, Webpack, and TypeDoc, each with its own configuration file.

3.6.1 Development Scripts

To automate various development tasks and improve the overall workflow in the project, scripts are set up for the project in the `package.json` file; see Listing 3.2. Breaking down each script according to its listed order, `test:functions` and `test:ui` will run jest to test the calculation functions and UI of the system. `Lint` ensures

that the code quality is good by following specified TypeScript rules, and that all JavaScript, TypeScript and JSON files have the same formatting style. `Build` will handle bundling of the code using Webpack, whereas `build:docs` generates the documentation with TypeDoc. The last alias, `release`, creates a release version of the application by using Gulp to automate the packaging process.

```
"scripts": {
    "test": "jest",
    "test:functions": "jest Function.test.ts",
    "test:ui": "jest UI.test.ts",
    "lint": "eslint . --ext .ts,.tsx,.html,.css",
    "lint:fix": "eslint . --ext .ts,.tsx,.html,.css --fix",
    "build": "webpack --config webpack.config.js",
    "build:docs": "typedoc --options typedoc.json",
    "release": "gulp release",
  }
```

Listing 3.2: Extract of scripts from `package.json`.

3.6.2 ESLint and Prettier Configuration

The ESLint configuration will provide a robust linting setup for this project. By utilizing both recommended rule sets, `eslint:recommended` and `plugin:@typescript-eslint/recommended`, the coverage of the source code will be extensive. The formatting style of the code will be enforced through the application of Prettier options.

The first option in Listing 3.3 is set to use the TypeScript parser. The plugins permits ESLint to use TypeScript-specific linting rules, linting for HTML files, and format using Prettier. As for the rules for this project, those that are not in the recommended rule sets or have been modified are defined here. ESLint generates an error if a function lacks a defined return type. For the other rules, warnings are issued if naming conventions are not followed, if the logical branching limit is exceeded, or if the maximum lines of code exceeds 500. Finally, the Prettier options are configured to throw an error if any rules are violated.

```
{
  "parser": "@typescript-eslint/parser",
  "plugins": ["@typescript-eslint", "html", "prettier"],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:prettier/recommended"
  ],
  "rules": {
    "@typescript-eslint/explicit-function-return-type": "error",
    "@typescript-eslint/naming-convention": [ "..." ],
    "complexity": [
      "warn",
      {
        "max": 10
      }
    ],
    "max-lines": ["warn", {"max": 500, "skipComments": true}],
    "prettier/prettier": [
      "error",
      {
        "endOfLine": "auto",
        "printWidth": 80,
        "useTabs": true,
        "semi": true,
        "singleQuote": false,
        "trailingComma": "none"
      }
    ]
  }
}
```

Listing 3.3: Truncated version of `.eslintrc.json`.

3.6.3 Webpack Configuration

Listing 3.4 shows the Webpack configuration for transpiling TypeScript to JavaScript and bundling the source files to distributed files, in the form of bundled JavaScript and processed CSS. The mode is set to production to optimize the output and reduce its size. The entry point for our system is the `Main.ts` function and the output is the directory where the bundled JavaScript file is stored, together with the main CSS file, from which they are imported into the pages by their relative path to the distribution folder.

```
module.exports = {
  mode: "production",
  entry: "./Src/Scripts/Main.ts",
  output: { filename: "bundle.js", path: __dirname + "/Dist", },
  devtool: "source-map",
  resolve: { ... },
  module: { ... },
  plugins: [ ... ],
  optimization: { ... },
};
```

Listing 3.4: Truncated version of `webpack.config.js`.

3.6.4 TypeDoc Configuration

As seen in Listing 3.5, the documentation will have its entry points in the directory called `Src/Scripts` and will process all files located there, as this is the only relevant code for which documentation can be generated. The `README.md` file will also be included in the system documentation, containing a description of the system, requirements, and user steps. This generated documentation will be available in the `Docs` directory at the root of the project and linked to in the navigation bar of the application. The structure and content of the documentation is further presented in Chapter 6.5.

```
{
  "entryPoints": ["Src/Scripts/*"],
  "entryPointStrategy": "expand",
  "out": "Docs",
  "cleanOutputDir": false,
  "name": "System Documentation",
  "readme": "Docs/README.md"
}
```

Listing 3.5: The `typedoc.json` configuration file.

4. Implementation

This chapter provides a review of all stages in the practical development process, in the following order: structuring the page layout, creating the main function flow, developing the functionality for importing and processing the user-uploaded benchmark results from the trace file, calculating the new data field types, implementing the result storage using the JSON format, generating the data table and charts, overview of the plot types that will be needed, usage of demo data, registering and configuring the service worker for PWA, and finally, creating the application release packaging. This gives a complete picture of how the system is developed to meet the requirements using the development methodology in the previous chapter. Subsequently, the testing and overview of this development and the finished product is done in Chapter 5 and Chapter 6.

4.1 Page Layout

Listing 4.1 shows a simplified and reduced version of the HTML code used for both the table and plot pages. In accordance with standard web design, the navigation menu is placed at the top of the page, and contains the following elements: navigation links for the table and plot pages, input elements for selecting and confirming the files, external navigation links for accessing the documentation, the GitHub repository, and a link to the issue submission form, a help button that opens a modal dialog with information about the supported file formats and standard headers for the trace files, and lastly, an indicator for the current release version. Since the table page is also the landing page for the application, it includes a button for activating demo data. For a detailed breakdown, see Chapter 6.1.

A button container is placed below the navigation bar. It contains the necessary buttons to provide the required functionality with the following actions: view the table or plot, view the selected rows on the table page, save the data to local storage, download the results data, download a customized version of the results data, download the chart data on the plot pages, delete the data from local storage, and clear the current data table. The slight difference in appearance between the

pages can be seen by comparing Figure 4.1 for the table page and Figure 4.2 for the plot pages. Refer to Chapter 6.3 for more details on these buttons.

The following section on each page will contain the specific material for each page, which for this project will be the table and the different plot types, and if applicable to the plot type, their summary statistics table. The last section contains the informational modal, which is identical for all pages, both in appearance and content.

Bootstrap classes are added to the elements to apply styling and positioning rules, along with our own CSS rules to adjust margins and padding between certain elements. The combination of these makes for a responsive and consistent UI throughout the system.

```
<body>
    <nav class="navbar"> <!-- Navigation bar with links. --> </nav>
    <div class="d-flex flex-wrap justify-content-start gap-2">
        <div col="auto"> <button>Functionality 1</button> </div>
        <div col="auto"> <button>Functionality 2</button> </div>
        <div col="auto"> <button>Functionality ...</button> </div>
    </div>
    <table> <!-- Table location --> </table>
        <!-- Alternatively if it is a plot: -->
    <canvas> <!-- Plot location --> </canvas>
    <table> <!-- Statistics table location --> </table>
    <div class="modal"> <!-- Modal with file info --> </div>
    <div class="modal"> <!-- Modal with configuration options --> </div>
    <script id="MainFunction" src="Dist/bundle.js"></script>
</body>
```

Listing 4.1: Simplified HTML code of the general layout.

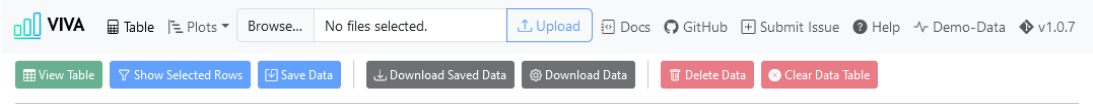


Figure 4.1: Navigation bar and button menu on the table page.

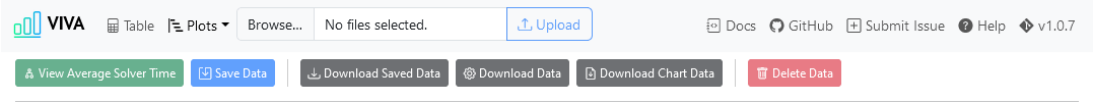


Figure 4.2: Navigation bar and button menu on the chart pages.

4.2 Referencing the Elements

An exportable constant variable of the specified type is created for each HTML element, so that they can be accessed by other modules within the code base by importing it when it is needed. This is achieved by using the `document.getElementById()` method to obtain the reference to the element by its unique ID and saving it as an exported constant variable. Type assertion is utilized to determine what kind of element is being referenced. This practice aids in catching type errors during development, for instance, by using the input element incorrectly when attempting to access a property that does not exist on `HTMLButtonElement`. A sample of the file with the element references can be seen in Listing 4.2, where the elements are an input, a button, a div, an icon, and a span.

All referenced elements are kept in a single file, because this project uses only a few elements between the table and plot pages. If there were no referencing system given the current implementation, maintaining unique references in the code would take more time.

```
export const fileInput: HTMLInputElement = document.getElementById(
    "fileInput"
) as HTMLInputElement;

export const importDataButton: HTMLButtonElement = document.getElementById(
    "importDataButton"
) as HTMLButtonElement;

export const alertNotification: HTMLDivElement = document.getElementById(
    "alertNotification"
) as HTMLDivElement;

export const alertIcon: HTMLElement = document.getElementById(
    "alertIcon"
) as HTMLElement;

export const alertMessage: HTMLSpanElement = document.getElementById(
    "alertMessage"
) as HTMLSpanElement;

export const closeAlertButton: HTMLButtonElement = document.getElementById(
    "closeAlertButton"
) as HTMLButtonElement;

export const viewTableButton: HTMLButtonElement = document.getElementById(
    "viewTableButton"
) as HTMLButtonElement;
```

Listing 4.2: `Elements.ts`, example of the element references.

4.3 Main Function Flow

The main file entry point is structured to execute functions according to a user flow, visible in Figure 4.3. To maintain readability, simplify maintenance, and make it easy to extend the system with new functionality, the methods are organized in files that are in subfolders according to event type. The first thing that takes effect when the user opens an page in the application and the main script file is executed is the import of the referenced elements, methods, CSS, and libraries. After that, the registration of the service worker function is run, followed by the initialization of seven global variables that will store various data, file extension of the loaded result type, and a default time and gap limit, which will be used when plotting the absolute performance profile.

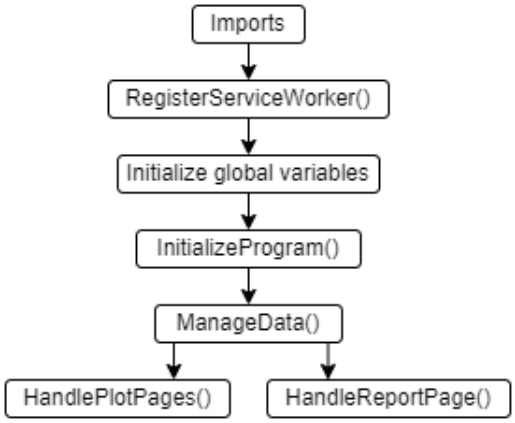


Figure 4.3: Overview of the content and order in `Main.ts`.

`DataFileType` is a string that stores the type of extension used for the imported data, as the system performs different functions depending on the format of the benchmark results. As already mentioned, the main format of the result files is trace, while JSON can be used if it has the correct structure, hence the string in this variable will either be `trc` or `json`. `DefaulTime` is used for setting the default fallback time used when rendering the chart of the absolute performance profile. The `gapLimit` will set the percentage limit for the gap limit in the same chart as well. The variable `unprocessedData` stores the imported and unprocessed benchmark results as an array of lines from the file. The variable `unprocessedInstanceInformationData` stores an unprocessed comma-separated values file containing additional properties and their values for the instances. The variable `unprocessedSolutionData` stores an unprocessed solution file that provides the best known primal

and dual bounds for each instance. However, the latter two are only used if these optional files are loaded. Lastly, `chartData` will store the processed data used when rendering the different charts.

The first function of the user flow, `InitializeProgram()`, follows the initialization of these variables, where it mainly initializes the methods needed to run the system. It resets the seven mentioned global variables to their initial state. For instance, this action will also occur when the table is cleared and `InitializeProgram()` is executed again. The state of the system buttons is reset to ensure that they are enabled and disabled according to the page type the user is currently on, either the table or any of the plot pages, which is checked by simply comparing the page title to a string. Otherwise, their states may be cached wrongly. It also clears the file input text box to avoid confusion for the end user as to whether files have been selected or not if the input value is cached. See Listing 4.3 for an example.

```
export function ElementStatesTablePage(): void {
    loaderContainer.innerHTML = "";
    dataTable.style.visibility = "hidden";
    fileInput.value = "";
    importDataButton.disabled = true;
    viewTableButton.disabled = true;
    showSelectedRowsButton.disabled = true;
    saveLocalStorageButton.disabled = true;
    downloadConfigurationButtonLayer.disabled = true;
    configurationSettingsButton.disabled = true;
    deleteLocalStorageButton.disabled = true;
    clearTableButton.disabled = true;
}
```

Listing 4.3: `ElementStatesTablePage()`, setting the elements initial states.

Following these events, the system tries to retrieve a stored configuration from the browser's local storage, as seen in Listing 4.4. The configuration contains the benchmark data, the file type used, and an optional default time. If a configuration file is found and read successfully, the user is informed by a notification alert that a saved user configuration has been loaded and the system moves forward calling `ManageData()`. If no such configuration exists, the system remains idle until one or more files are loaded.

```
try {
    [unprocessedData, dataFileType, defaultTime, primalGap] = GetUserConfiguration();
    if (localStorage.getItem("DemoData") === "true") {
        ImportDataEvents("Using demo mode!", "json");
        NotifyDemoMode();
    } else {
        ImportDataEvents("Found cached benchmark file!", "json");
    }
    saveLocalStorageButton.disabled = true;
    deleteLocalStorageButton.disabled = false;
    downloadConfigurationButtonLayer.disabled = false;
    sessionStorage.setItem("savedStorageNotification", "true");
    ManageData();
} catch {
    console.info("No saved configuration data found.");
}
```

Listing 4.4: Retrieving data from local storage and updating the button states.

Once the user has selected the files they want to use, see Listing 4.5, the contents are read into the previously mentioned variables. The Upload button is then enabled and when the user clicks it, the state of it and the rest of the buttons are updated, and the system state is taken to the `ManageData()` function, as seen in Listing 4.6.

```
fileInput.addEventListener("change", () => {
    dataFileType = GetDataFileType();
    ReadData(unprocessedData, unprocessedInstanceInformationData, unprocessedSolutionData);
});
```

Listing 4.5: Functions to execute when the change action is triggered.

```
importDataButton.addEventListener("click", () => {
    sessionStorage.removeItem("savedStorageNotification");
    const fileNames = Array.from(fileInput.files)
        .map((file) => {return file.name;}).join(", ");
    ImportDataEvents(
        "Benchmarks loaded with following files: \n".concat(fileNames)
    );
    ManageData();
});
```

Listing 4.6: Display a successful alert with the loaded files and move the system state forward.

The data is assigned to variables based on the file type used to upload it. If the uploaded data is in a JSON file, the system retrieves the data from local storage in Listing 4.7, because it will be automatically stored to the local storage when loaded files are read and a JSON file is found. If the prioritized trace format is used for the result file, it will apply metadata, explained in Chapter 4.4.3, if either the file is provided for instance information properties, or the solution file is provided with the best known primal and dual bounds, which can be seen in Listing 4.8. A more detailed discussion of transforming the data follows in Section 4.4.

```
if (dataFileType == "json") {
    [unprocessedData, dataFileType, defaultTime, gapLimit] = GetUserConfiguration();
    traceData = ExtractTraceData(unprocessedData);
}
```

Listing 4.7: Functions run when using a JSON file.

```
if (dataFileType === "trc") {
    traceData = ExtractTraceData(unprocessedData);

    if (unprocessedInstanceInformationData.length !== 0) {
        instanceInfoData = GetInstanceInformation(
            unprocessedInstanceInformationData
        );
        traceData = MergeData(traceData, instanceInfoData);
    }

    if (unprocessedSolutionData.length !== 0) {
        soluData = GetBestKnownBounds(unprocessedSolutionData);
        traceData = MergeData(traceData, soluData);
    }
    AddResultCategories(traceData);
}
```

Listing 4.8: Functions run when using trace file(s).

Since this can be considered the end of the main flow, any remaining functions that are executed are due to interaction with any of the buttons. Depending on the page the user is currently on, a function manages the functionality of these, showcased in Listing 4.9. This allows the rest of the features in the program to be run, giving the user access to view and interact with the benchmarking content in the form of their choice, either in a tabular format or as plots. The flow can be reset by executing `InitializeProgram()`, responsible for the initialization phase.

```
function HandleReportPage(
    traceData: object[],
    traceDataFiltered: object[]
): void {
    viewTableButton.addEventListener("click", () => {
        viewTableButton.disabled = true;
        TableLoadingAnimation();
        DisplayDataTable(traceData);
    });

    showSelectedRowsButton.addEventListener("click", () => {
        showSelectedRowsButton.disabled = true;
        traceDataFiltered = UpdateResults();

        if (traceDataFiltered.length === 0) {
            DisplayWarningNotification("No rows selected for filtering.");
            showSelectedRowsButton.disabled = false;
        } else {
            DisplayDataTable(traceDataFiltered);
        }
    });

    saveLocalStorageButton.addEventListener("click", () => {
        function RemapObjectProperties(traceData: object[]): object[] {
            return traceData.map((obj) => {
                const remappedObj = {};
                for (const key in obj) {
                    const newKey = ReversedTraceHeaderMap[key] || key;
                    remappedObj[newKey] = obj[key];
                }
                return remappedObj;
            });
        }

        let newRawData = [];
        if (dataFileType === "trc") {
            dataFileType = "json";
        }
        if (traceDataFiltered.length === 0) {
            newRawData = CreateNewTraceData(traceData);
        } else {
            const remappedData = RemapObjectProperties(traceDataFiltered);
            newRawData = CreateNewTraceData(remappedData);
        }
        CreateUserConfiguration(newRawData, dataFileType);
        deleteLocalStorageButton.disabled = false;
        downloadConfigurationButtonLayer.disabled = false;
    });

    clearTableButton.addEventListener("click", () => {
        DestroyDataTable();
        InitializeProgram();
    });
}
```

Listing 4.9: `HandleReportPage()`, showcasing functions binding to a button.

4.4 Data Processing

First, when the files are loaded and the processing steps begin, the system checks the file formats from the file input and counts the number of extensions used in Listing 4.10. Either one of the files needs to be a trace or a JSON file, and if there are multiple JSON files, or a file with a non-supported extension, it throws an error and notifies the user by an error notification, see Listing 4.11. If the verification passes, it returns the extension file type, either trc or JSON.

```
function CheckFileExtension(
    file: File,
    fileCounts: {
        [x: string]: number;
        csv?: number;
        solu?: number;
        json?: number;
    }
): string {
    const extension = file.name.split(".").pop();

    if (!IsValidExtension(extension)) {
        DisplayErrorNotification(
            "Invalid file extension. Only .trc, .json, .solu, and .csv allowed."
        );
        throw new Error(
            "Invalid file extension. Only .trc, .json, .solu, and .csv allowed."
        );
    }

    if (extension in fileCounts) {
        fileCounts[extension]++;
        if (fileCounts[extension] > 1) {
            DisplayErrorNotification(`Cannot upload multiple .${extension} files.`);
            throw new Error(`Cannot upload multiple .${extension} files.`);
        }
    }

    return extension;
}
```

Listing 4.10: `CheckFileExtension()`, verifying that loaded file types are valid.

Each uploaded file is read based on its extension according to Listing 4.12. If the file is a JSON file, the content is parsed to verify that the required properties exist and then saved to local storage, after which the user is notified whether the actions were successful or not. The other file types are read line by line and pushed into separate arrays. The only difference is the specific line endings

which must be accounted for. After processing all files, the function returns an object containing the `unprocessedData`, `unprocessedInstanceInformationData`, and `unprocessedSolutionData` string arrays. The following subsections describe the modifications made to these arrays of strings before they can be used to produce results on the table and plot pages.

```
export function GetDataFileType(): string {
    const files = fileInput.files;
    const extensions = [];
    const fileCounts = { csv: 0, solu: 0, json: 0 };

    for (let i = 0; i < files.length; i++) {
        const extension = CheckFileExtension(files[i], fileCounts);

        if (["trc", "solu", "json", "csv"].includes(extension)) {
            extensions.push(extension);
        }
    }

    if (extensions.length === 0) {
        DisplayErrorNotification("No .trc, .json or .solu files found.");
        throw new Error("No .trc, .json or .solu files found.");
    }

    const hasTRCOrJSON = extensions.some((ext) => {
        return ext === "trc" || ext === "json";
    });

    if (!hasTRCOrJSON) {
        DisplayErrorNotification("At least one .trc or .json file required.");
        throw new Error("At least one .trc or .json file required.");
    }

    if (extensions.includes("trc") && extensions.includes("json")) {
        DisplayErrorNotification(
            "Cannot upload both .trc and .json files simultaneously."
        );
        throw new Error("Cannot upload both .trc and .json files simultaneously.");
    }

    return extensions.find((ext) => {
        return ext === "trc" || ext === "json";
    });
}
```

Listing 4.11: `GetDataFileType()`, verifying that correct file types are loaded.

```
export function ReadData(
    unprocessedData: string[],
    unprocessedInstanceInformationData: string[],
    unprocessedSolutionData: string[]
): {
    UnprocessedData: string[];
    UnprocessedInstanceInformationData: string[];
    UnprocessedSolutionData: string[];
} {
    importDataButton.disabled = false;

    for (let i = 0; i < fileInput.files.length; i++) {
        const reader = new FileReader();
        const file = fileInput.files[i];
        const fileName = file.name;
        const fileExtension = fileName.split(".").pop();

        reader.addEventListener("load", function () {
            if (fileExtension === "json") {
                const dataJSON = <string>reader.result;
                const parsedData = JSON.parse(dataJSON);

                VerifyConfigurationProperties(parsedData);
            } else if (fileExtension === "trc") {
                const lines = (<string>reader.result)
                    .split(/\r?\n/)
                    .map((line) => line.trim());
                for (let i = 0; i <= lines.length - 1; i++) {
                    const line = lines[i];
                    unprocessedData.push(line);
                }
            } else if (fileExtension === "csv") {
                const lines = (<string>reader.result).split("\n");
                for (let i = 0; i <= lines.length - 1; i++) {
                    const line = lines[i];
                    unprocessedInstanceInformationData.push(line);
                }
            } else if (fileExtension === "solu") {
                const lines = (<string>reader.result).split(/\r?\n/);
                for (let i = 0; i <= lines.length - 1; i++) {
                    const line = lines[i];
                    unprocessedSolutionData.push(line);
                }
            }
        });
        reader.readAsText(file);
    }

    return {
        UnprocessedData: unprocessedData,
        UnprocessedInstanceInformationData: unprocessedInstanceInformationData,
        UnprocessedSolutionData: unprocessedSolutionData
    };
}
```

Listing 4.12: `ReadData()`, abbreviated overview of file processing.

4.4.1 Trace File

The processing of the raw string array created from a loaded trace file begins with checking if an asterisk is in the first row of the array, followed by a comma-separated list of headers starting on the following rows. If this list exists, it is used to map the data to objects, otherwise a default list of headers from Table 1.1 is used, see Listing 4.13. A line processing function loops through the rest of the raw data array of comma-separated data values in Listing 4.14, creating an empty object for each element, storing the value of each element as a property in the object, and using the corresponding header element as the property name. If the combination of a problem and a solver has been used before, the loop is skipped. If the header is `ObjectiveValue` or `ObjectiveValueEstimate`, it will convert the value to an exponential notation, to improve readability of these values in the table. This also applies to the solver time value, which will be rounded to two decimal places.

```
export function ExtractTraceData(data: string[]): object[] {
    const firstLine = data[0].split(",");
    let traceData: object[] = [];

    if (firstLine[0].startsWith("*")) {
        const headers = ExtractHeaders(data);
        const startIdx = data.findIndex((line) => !line.startsWith("*"));
        traceData = ProcessLines(headers,data, startIdx);
    } else if (!firstLine[0].startsWith("*")) {
        traceData = ProcessLines(defaultHeaders, data, 0);
    }
    return traceData;
}
```

Listing 4.13: `ExtractTraceData()`, action sequence depending on header usage.

The additional data fields mentioned in Chapter 1.2 are calculated and added on a per-object basis, see Listing 4.15 for an example. The calculations to be performed to obtain the new result types are discussed in the next sub-sections. All calculations were directly based on their implementation in PAVER [2]. If the best known primal and dual bounds are provided via a metadata file, they are applied to the data, as discussed in Chapter 4.4.3.3, before these result categories are added, which can be seen in Listing 4.8. After these modifications have been performed, the results can be used in the system.

```
function ProcessLines(
    headers: string[],
    data: string[],
    startIdx: number
): object[] {
    const traceData = [];
    const previousRow = {};
    for (let i = Math.max(startIdx, 0); i < data.length; i++) {
        if (data[i].startsWith("*")) continue;

        const currentLine = data[i].split(",");
        const fileName = currentLine[0];
        const solverName = currentLine[2];

        if (previousRow[fileName] === solverName) {
            continue;
        }
        previousRow[fileName] = solverName;

        if (currentLine.some((cell) => cell.trim() !== "")) {
            const obj = {};
            for (let j = 0; j < headers.length; j++) {
                let value = currentLine[j];

                if (
                    headers[j] === "ObjectiveValue" ||
                    headers[j] === "ObjectiveValueEstimate"
                ) {
                    value = Number(value).toExponential(6);
                }

                if (headers[j] === "SolverTime") {
                    value = (Math.round(Number(value) * 100) / 100).toString();
                }

                obj[headers[j]] = value;
            }
            traceData.push(obj);
        }
    }
    return traceData;
}
```

Listing 4.14: `ProcessLines()`, setting values to corresponding headers.

```
export function AddResultCategories(traceData: object[]): void {
    for (const obj of traceData) {
        obj["Direction"] = CalculateDirection(obj["Direction"]);

        obj["PrimalBoundSolver"] = CalculatePrimalBound(
            obj["ObjectiveValue"],
            obj["Direction"]
        );
        obj["DualBoundSolver"] = CalculateDualBound(
            obj["ObjectiveValueEstimate"],
            obj["Direction"]
        );

        obj["ModelStatus"] = SetModelStatus(obj["ModelStatus"] as string | number);
        obj["SolverStatus"] = SetSolverStatus(obj["SolverStatus"] as string | number);

        if (!obj.hasOwnProperty.call("PrimalBoundProblem") || !obj["PrimalBoundProblem"]) {
            obj["PrimalBoundProblem"] = CalculatePrimalBound(
                obj["PrimalBoundSolver"],
                obj["Direction"]
            );
        }
        if (!obj.hasOwnProperty.call("DualBoundProblem") || !obj["DualBoundProblem"]) {
            obj["DualBoundProblem"] = CalculateDualBound(
                obj["DualBoundSolver"],
                obj["Direction"]
            );
        }

        obj["Gap_Solver"] = CalculateGap(
            obj["PrimalBoundSolver"],
            obj["DualBoundSolver"],
            obj["Direction"]
        );
        obj["Gap_Problem"] = CalculateGap(
            obj["PrimalBoundProblem"],
            obj["DualBoundProblem"],
            obj["Direction"]
        );
        obj["PrimalGap"] = CalculateGap(
            obj["PrimalBoundSolver"],
            obj["PrimalBoundProblem"],
            obj["Direction"]
        );
        obj["DualGap"] = CalculateGap(
            obj["DualBoundSolver"],
            obj["DualBoundProblem"],
            obj["Direction"]
        );
    }
}
```

Listing 4.15: `AddResultCategories()`, data fields modified and added to the results.

4.4.1.1 Direction

The function that returns the new calculated optimization direction has to initially convert the input direction into a number and then run the operation $1 - 2 \times \text{Direction}$. The result is returned as a string to be in a more readable form in the table. The direction is set to `max` if the value equals -1, a maximization problem, if it is anything else it is set to `min`, a minimization problem. See Listing 4.16.

```
export function CalculateDirection(direction: number | string): string {
    direction = 1 - 2 * Number(direction);
    if (direction === -1) {
        direction = "max";
        return direction;
    } else {
        direction = "min";
        return direction;
    }
}
```

Listing 4.16: `CalculateDirection()`, calculation of direction.

4.4.1.2 Primal and Dual Bound

The primal and dual bounds for the solver are based on the `ObjectiveValue` or `ObjectiveValueEstimate` headers, whereas the primal and dual bound for the problem is set from the external solution file, or by basing it on the value of the solvers primal and dual bounds, as seen in Listing 4.15. The function for calculating the primal bound, Listing 4.18, is based on the same conditions as in PAVER [48].

It converts the argument into a corresponding numerical value based on its type and value. If the argument is an empty string, or any other variation of not a number, it returns `-Infinity` for `max` direction and `Infinity` for `min` direction. If the argument is a variation of infinity, it returns `Infinity`, and for negative infinity, it returns `-Infinity`. Otherwise, it converts the current value into a number. The function for calculating the dual bound is similar, except that an unknown value results in `Infinity` if the direction is `max`, and `-Infinity` for `min`, as seen in Listing 4.17.

```
dualBound = direction === "max" ? Infinity : -Infinity;
```

Listing 4.17: Dual bound value for unknown argument.

```
export function CalculatePrimalBound(
    primalBound: number | string,
    direction: string
): number | string {
    if (typeof primalBound === "string") {
        switch (primalBound.toLowerCase()) {
            case "":
            case "na":
            case "nan":
            case "-nan":
                primalBound = direction === "max" ? -Infinity : Infinity;
                break;
            case "inf":
            case "+inf":
                primalBound = Infinity;
                break;
            case "-inf":
                primalBound = -Infinity;
                break;
            default:
                primalBound = Number(primalBound);
        }
    }
    return primalBound.toExponential(6);
}
```

Listing 4.18: `CalculatePrimalBound()`, calculation of primal bound.

4.4.1.3 Gaps

Calculations to obtain the gaps are done according to the function in Listing 4.19. By default, when calculating the gap for the solver and problem, argument `a` is the primal bound and `b` is the dual bound for a min direction, or if calculating a primal or dual gap, `a` is the bound for the solver and `b` the bound for the problem. The two values are switched if the direction is `max`. The tolerance level is used when verifying if the two number arguments are approximately equal and it is set by default to 1×10^{-9}. If `a` and `b` are approximately equal within this tolerance, the function returns `0`. If the minimum absolute value of `a` and `b` is less than the tolerance, or if either `a` or `b` is `Infinity`, or if `a` and `b` have different signs, or if either `a` or `b` is not a number, it returns `Infinity`. Otherwise, it returns the relative difference between `a` and `b` divided by the minimum absolute value of `a` and `b` [47].

```
export function CalculateGap(a: number, b: number, dir: string, tol = 1e-9): number
{
    if (dir === "max") { [a, b] = [b, a]; }

    if (Math.abs(a - b) < tol) { return 0.0; }

    if (
        isNaN(a) || isNaN(b) || Math.min(Math.abs(a), Math.abs(b)) < tol ||
        (a === Infinity && b === Infinity) ||
        a === Infinity || b === Infinity ||
        a * b < 0
    ) { return Infinity; }

    const result = Math.abs(
        Math.round(((a - b) / Math.min(Math.abs(a), Math.abs(b))) * 10000) / 100
    );
    return result;
}
```

Listing 4.19: `CalculateGap()`, calculation of the gap.

4.4.1.4 Model and Solver Status

The model [8] and solver [9] status, is given as numbers or string inputs, where each value represents the cause for termination. An object is used to serve as a mapping table, like in Listing 4.20, to map the numeric keys to status message, which is returned, where the corresponding causes can be seen in Table 4.1. A value other than the ones listed returns a string with the value set to `Unknown Error`. This mapping is also based on the implementation i PAVER [49].

```
const statusMap: { [key: number]: string } = {
    1: "Normal Completion",
    2: "Iteration Interrupt",
    3: "Resource Interrupt",
    4: "Terminated By Solver",
    5: "Evaluation Interrupt",
    6: "Capability Problems",
    7: "Licensing Problems",
    8: "User Interrupt",
    9: "Error Setup Failure",
    10: "Error Solver Failure",
    11: "Error Internal Solver Failure",
    12: "Solve Processing Skipped",
    13: "Error System Failure"
};
```

Listing 4.20: `statusMap`, dictionary for mapping numbers to messages.

Table 4.1: Termination input value and corresponding message.

Expression	Model	Solver
1	Optimal	Normal Completion
2	Locally Optimal	Iteration Interrupt
3	Unbounded	Resource Interrupt
4	Infeasible	Terminated By Solver
5	Locally Infeasible	Evaluation Interrupt
6	Intermediate Infeasible	Capability Problems
7	Feasible Solution	Licensing Problems
8	Integer Solution	User Interrupt
9	Intermediate Non-integer	Error Setup Failure
10	Integer Infeasible	Error Solver Failure
11	Licensing Problem - No Solution	Error Internal Solver Failure
12	Error Unknown	Solve Processing Skipped
13	Error No Solution	Error System Failure
14	No Solution Returned	
15	Solved Unique	
16	Solved	
17	Solved Singular	
18	Unbounded - No Solution	
19	Infeasible - No Solution	
default	Unknown Error	Unknown Error

4.4.2 JSON File

As previously mentioned, if the results are in the form of a JSON file, a function is called to store the dataset in the browser's local storage immediately after loading the file's contents. This is done by converting the `userData` to a string representation that includes the dataset, file type, and optional default time and gap limit, as shown in Listing 4.21.

The user is notified if saving the configuration was successful and can continue interacting with the results immediately after this action, as no other processes are performed on the data, as long as it was previously processed properly when saved to a JSON file.

The size limit of the local storage is 5 MiB [29], which is sufficient since a trace file with 100 results is about 10 KB in size, but if the quota is exceeded, the user will be notified about it with an error notification.

```
interface UserData {
    dataSet: string[] | object[];
    dataFileType: string;
    defaultTime?: number | undefined;
    gapLimit?: number | undefined;
}

export const userData: UserData = {
    dataSet: [],
    dataFileType: "",
    defaultTime: undefined,
    gapLimit: undefined
};
```

Listing 4.21: Definition of the interface `UserData` and `userData` object.

4.4.2.1 Loading and Deleting the Configuration.

The user configuration is retrieved by calling `GetUserConfiguration()` in Listing 4.22. The function locates the `UserConfiguration` key from the browser's local storage, then parses the retrieved object with `JSON.parse()` to convert it from a string to a JavaScript object. The properties are extracted from the parsed object and returned to their respective variables.

```
export function GetUserConfiguration(): [string[], string, number, number] {
    let userConfig: UserData;
    try {
        userConfig = JSON.parse(localStorage.getItem("UserConfiguration"));
    } catch (error) {
        // ...Shortened version.
    }
    const unprocessedData = [];
    userConfig.dataSet.forEach((value) => {
        unprocessedData.push(value);
    });

    const dataFileType = userConfig.dataFileType;
    const defaultTime = userConfig.defaultTime;
    const gapLimit = userConfig.gapLimit;
    return [unprocessedData, dataFileType, defaultTime, gapLimit];
}
```

Listing 4.22: `GetUserConfiguration()`, loading the user configuration.

Removing the `UserConfiguration` key is easily done with the following line in a called function, the user is also here notified about the action:

```
localStorage.removeItem("UserConfiguration");
```

4.4.2.2 Downloading the Configuration.

Downloading the configuration is accomplished by converting the configuration object into a downloadable file by creating a `new Blob` object. The `href` and `download` attributes are set to the URL and filename of the blob, respectively, as shown in Listing 4.23.

```
export function DownloadUserConfiguration(): void {
    const userConfig = JSON.parse(localStorage.getItem("UserConfiguration"));
    if (userConfig) {
        const downloadAbleFile = JSON.stringify(userConfig);
        const cfg = new Blob([downloadAbleFile], { type: "application/JSON" });

        downloadConfigurationButton.href = window.URL.createObjectURL(cfg);
        downloadConfigurationButton.download = "UserConfiguration.JSON";
    } else {
        DisplayErrorNotification("No saved configuration found!");
    }
}
```

Listing 4.23: `DownloadUserConfiguration()`, downloading the user configuration.

4.4.2.3 Downloading the Customized Configuration.

The code for managing the creation and download of a custom configuration file is based on the solvers selected by the user and the optional default time and gap limit specified, as these should cover the main customization needs. First, the solvers are retrieved from the loaded benchmark data and a list of them is created and presented in a modal when the user clicks the Download Configuration button. In this modal, there are also two input elements for entering the default time and the gap limit.

The code in Listing 4.24 initially checks if the `selectedSolvers` array is empty. If it is, it uses the only solver available. If the solvers have been selected, it filters the benchmark data based on the selected solvers and creates a new array of strings with the data to save. Finally, the download of the customized configuration is triggered by calling the `DownloadCustomizedUserConfiguration()` function with the new array of strings and, if provided, the default time and gap limit from the input fields. If not specified, they are set to `1000` and `0.01`, respectively.

```
downloadCustomConfigurationButton.addEventListener("click", () => {
    console.log(selectedSolvers);
    if (selectedSolvers.length === 0) {
        selectedSolvers[0] = solverSelector.value;
    }
    const customizedTraceData = traceData.filter((solver) => {
        return selectedSolvers.includes(solver["SolverName"]);
    });
    const newRawData: string[] = CreateNewTraceData(customizedTraceData);

    defaultTime = Number(defaultTimeInput.value);
    if (!defaultTime) {
        defaultTime = 1000;
    }

    gapLimit = Number(gapLimitInput.value);
    if (!gapLimit) {
        gapLimit = 0.01;
    }

    DownloadCustomizedUserConfiguration(
        newRawData,
        defaultTime,
        gapLimit
    );
});
```

Listing 4.24: Events run for downloading the customized user configuration.

4.4.3 Optional Files

The optional metadata files from MINLPLib can be loaded by the user in conjunction with trace files. These instance properties contain 78 headers for 1595 instances and are stored in a CSV file. The best known primal and dual bounds for each instance are stored in a solution file, consisting of 2166 records as of August 16th, 2023 [33].

4.4.3.1 Instance Information Properties

The header information is extracted from the first line of the file, from which the header labels are created. The function loops through the raw instance information file, starting at the seconds line, and creates a new object for each row of data. The header labels are used to map the values to the corresponding keys in the object. If there is no value for a particular key, an empty string is assigned as the value. Each object is added to the instanceInfo array on each iteration. The array containing the objects is returned after all rows have been iterated and the user is notified that the instance information has been applied. (Listing 4.25).

```
export function GetInstanceInformation(
    unprocessedInstanceInformationData: string[]
): object[] {
    const instanceInfo = [];
    let header = unprocessedInstanceInformationData[0].split(";");

    header = header.map((x) => {
        if (x === "primalbound") { return "PrimalBoundProblem"; }
        if (x === "dualbound") { return "DualBoundProblem"; }
        return x;
    });

    for (let i = 1; i < unprocessedInstanceInformationData.length; i++) {
        const obj = {};
        const currentLine = unprocessedInstanceInformationData[i].split(";");
        for (let j = 0; j < header.length; j++) {
            obj[header[j]] = currentLine[j] || "";
        }
        instanceInfo.push(obj);
    }
    DisplayAlertNotification("Instance information succesfully loaded!");
    return instanceInfo;
}
```

Listing 4.25: `GetInstanceInformation()`, parsing the metadata for instances.

4.4.3.2 Data from Solution File

Each string is parsed using a regular expression pattern to extract the values, as well as the filename: `const regexPattern = /^=(.*?)=\s+(.*?)\s+(.*?)$/;`. It extracts three pieces of information per line, keyword, instance, and bound. See Listing 4.26 of the solution file structure from the MINLPLib library [33].

```
=best=     4stufen                                116329.6706000000
=opt=      alan                                        2.9250000000
=opt=      alkylation                               1768.8069640000
=best=     ann_compressor_tanh                     22331.9006400000
=bestdual= ann_cumene_exp                         -14140.3905100000
```

Listing 4.26: Extract from the `minlp.solu` file.

For each matching row, this function creates a new object and adds a key and its value. If the first column has the keyword `best`, then the third column is the primal bound. If the value of the first column is `bestdual`, then it is a dual bound value, and finally, if the first column value is `opt`, then the third column indicates that it is both primal and dual. If the keyword is `inf`, then it sets the bounds to

infinite, see Listing 4.27. The object is then pushed into the `soluData` array and the processing of the raw data array continues.

```
switch (match[1]) {
    case "inf":
        try {
            obj["PrimalBoundProblem"] = Infinity;
            obj["DualBoundProblem"] = Infinity;
        } catch (err) {
            console.error(err);
            obj["PrimalBoundProblem"] = NaN;
            obj["DualBoundProblem"] = NaN;
        }
        break;
    case "best":
        try {
            obj["PrimalBoundProblem"] = Number(match[3]).toExponential(6);
        } catch (err) {
            console.error(err);
            obj["PrimalBoundProblem"] = NaN;
        }
        break;
    case "bestdual":
        try {
            obj["DualBoundProblem"] = Number(match[3]).toExponential(6);
        } catch (err) {
            console.error(err);
            obj["DualBoundProblem"] = NaN;
        }
        break;
    case "opt":
        try {
            obj["PrimalBoundProblem"] = Number(match[3]).toExponential(6);
            obj["DualBoundProblem"] = Number(match[3]).toExponential(6);
        } catch (err) {
            console.error(err);
            obj["PrimalBoundProblem"] = NaN;
            obj["DualBoundProblem"] = NaN;
        }
        break;
    default:
        break;
}
```

Listing 4.27: Extract from `GetBestKnownBounds()`, setting the bound values.

4.4.3.3 Merging Metadata to Results

The merging of the metadata into the trace results, as seen in Listing 4.8, is performed in the same way for both CSV and solution files. The function in Listing 4.28 iterates through each object in `traceData` and each object in the external data, and merges them based on the `InputFileName` or `name` properties. If

it matches the property of `traceData`, the function creates a new merged object, and adds it to the array with other merged objects. After all object pairs have been checked, the function returns the array of merged data if it is not empty.

```
export function MergeData(traceData: CategoriesObj[], data: CategoriesObj[]
): object[] {
    const mergedData = [];

    for (const traceDataObj of traceData) {
        let isMatchFound = false;
        for (const dataObj of data) {
            if (
                (dataObj["InputFileName"] &&
                    traceDataObj["InputFileName"] === dataObj["InputFileName"]) ||
                (dataObj["name"] && traceDataObj["InputFileName"] === dataObj["name"])
            ) {
                const mergedObj = Object.assign({}, traceDataObj, dataObj);
                mergedData.push(mergedObj);
                isMatchFound = true;
                break;
            }
        }
        if (!isMatchFound) {
            mergedData.push(traceDataObj);
        }
    }
    return mergedData;
}
```

Listing 4.28: `MergeData()`, merging additional information to the trace results.

4.5 Generating the Data Table and Statistics Tables

A layout for the table is created using the processed results on the table exclusive page. This table layout is used as the basis for the DataTables.js wrapper, which extends the functionality of interacting with the table by adding options such as sorting in ascending and descending order, searching for results, pagination, hiding and showing columns as desired, selecting multiple rows, and filtering the results. By using this as a wrapper, it is easy to switch to another table library or custom solution if needed.

A function following the same principle generates the statistical metrics tables for charts discussed in Chapter 4.6.1, but without the DataTable.js wrapper. Each solver gets its own row for these statistics, consisting of average, minimum, maximum, standard deviation, sum and percentile values of the results.

4.5.1 Creating the Table Layout

The `TableData()` function in Listing 4.29 dynamically generates a new table to display trace data and sets it in the designated HTML div location.

```
export function TableData(traceData: object[]): void {
    const dataTableDiv = dataTable;
    dataTableDiv.innerHTML = "";
    const dataTableHeaders = document.createElement("thead");
    dataTableHeaders.classList.add("thead-dark");

    const headerRow = document.createElement("tr");
    const sortedKeys = Object.keys(traceData[0]).filter(
        (k) => k !== "InputFileName" && k !== "SolverName"
    );
    sortedKeys.sort();
    sortedKeys.unshift("InputFileName", "SolverName");
    for (const key of sortedKeys) {
        const th = document.createElement("th");
        th.textContent = key || "NA";
        headerRow.appendChild(th);
    }
    dataTableHeaders.appendChild(headerRow);

    const dataTableContent = document.createElement("tbody");
    for (const obj of traceData) {
        const resultRow = document.createElement("tr");
        const sortedKeys = Object.keys(obj).filter((k) =>
        k !== "InputFileName" && k !== "SolverName"
        sortedKeys.sort();
        sortedKeys.unshift("InputFileName", "SolverName");
        for (const key of sortedKeys) {
            const value = obj[key];
            const td = document.createElement("td");
            td.textContent = value.toString();
            resultRow.appendChild(td);
        }
        dataTableContent.appendChild(resultRow);
    }

    const newDataTable = document.createElement("table");
    newDataTable.classList.add("table", "table-bordered", "table-sm");
    newDataTable.id = "dataTableGenerated";
    newDataTable.appendChild(dataTableHeaders);
    newDataTable.appendChild(dataTableContent);
    dataTableDiv.appendChild(newDataTable);
}
```

Listing 4.29: `TableData()`, the table generation process with trace results.

The header row is created from the first element of the result array, which contains all the header columns for the table. It is sorted using the function in Listing 4.30,

and then and renamed where the enum used is seen in Listing 4.31. The sorting is applied to make the categories in the table more readable, while still preserving the naming structure from the trace file. For each object in the `traceData` array, it creates a row of sorted values from the object into the columns of the table row. After iterating through the array, the data is appended to the element named `tbody`. Finally, a list of classes and an ID is set to the table, and the contents are appended to it.

```
function SortKeysByEnum(obj): string[] {
    const enumKeys = Object.keys(TraceHeaderMap);
    const objKeys = Object.keys(obj);

    const sortedEnumKeys = objKeys.filter((key) => { return enumKeys.includes(key); })
        .sort((a, b) => { return enumKeys.indexOf(a) - enumKeys.indexOf(b); });

    const nonEnumKeys = objKeys.filter((key) => { return !enumKeys.includes(key); })
        .sort();

    return [...sortedEnumKeys, ...nonEnumKeys];
}
```

Listing 4.30: `SortKeysByEnum()`, function for sorting keys based on the enum.

```
export enum TraceHeaderMap {
    InputFileName = "Problem",
    Direction = "Direction",
    ModelType = "ModelType",
    DualBoundProblem = "ProblemDualBound",
    PrimalBoundProblem = "ProblemPrimalBound",
    Gap_Problem = "ProblemGap",
    NumberOfVariables = "#Variables",
    NumberOfDiscreteVariables = "#DiscreteVariables",
    NumberOfEquations = "#Equations",
    NumberOfNonzeros = "#NonZeros",
    NumberOfNonlinearNonZeros = "#NonlinearNonZeroes",
    SolverName = "Solver",
    ModelStatus = "ModelStatus",
    SolverStatus = "SolverStatus",
    DualBoundSolver = "SolverDualBound",
    DualGap = "SolverDualGap[%]",
    PrimalBoundSolver = "SolverPrimalBound",
    PrimalGap = "SolverPrimalGap[%]",
    Gap_Solver = "SolverGap",
    SolverTime = "SolverTime",
    NumberOfIterations = "#SolverIterations",
    NumberOfNodes = "#SolverNodes"
}
```

Listing 4.31: `TraceHeaderMap`, sorting and remapping properties.

4.5.2 Applying the DataTables.js Wrapper

The initialization and configuration of the DataTable.js is done in `DataTablesConfiguration()` 4.33 after the base for the table has been created. It locates the table with the ID `dataTableGenerated` and applies the wrapper. This wrapper provides functionality that meets our criteria for making the result data more interactive. The Document Object Model (hereafter DOM) structure is set up in Listing 4.32, with buttons (`B`) and a search element (`f`) above the table (`tr`), and information (`i`) and pagination (`p`) below, as shown in Figure 6.1.

```
"<'row'<'col-sm-12 col-md-6'B><'col-sm-12 col-md-6'f>>" +
"<'row'<'col-sm-12'tr>>" +
"<'row'<'col-sm-12 col-md-5'i><'col-sm-12 col-md-7 mt-2'p>>",
```

Listing 4.32: Setting up the DOM for the wrapper.

```
function DataTablesConfiguration(): void {
    const table = $("#dataTableGenerated").DataTable({
        destroy: true,
        stateSave: true,
        searchPanes: { ... },
        dom: ...,
        lengthChange: true,
        lengthMenu: [ "..." ],
        select: { ... },
        responsive: { ... },
        scrollY: "",
        scrollX: true,
        scrollCollapse: true,
        paging: true,
        fixedColumns: true,
        columnDefs: [ ... ],
        language: { ... }
        buttons: [ ... ],
        initComplete: function () {
            $("#loaderContainer").css("display", "none");
            $("#loaderContainer").hide();
            $("#dataTable")
                .css("opacity", "0")
                .css("visibility", "visible")
                .animate({ opacity: 1 }, 500);
            const columnNamesToShow = DEFAULT_VISIBLE_HEADERS;
            this.api()
                .columns()
                .every(function () {
                    const columnName = this.header().textContent;
                    if (!columnNamesToShow.includes(columnName)) {
                        this.visible(false);
                    }
                });
        }
    });

    $(".dataTables_length select").addClass("custom-select custom-select-sm");
    table.searchPanes.container().prependTo(table.table().container());
    table.searchPanes.resizePanes();
    table.searchPanes.container().toggle();
}
```

Listing 4.33: `DataTablesConfiguration()`, truncated overview of options used.

4.5.3 New Data Based on Visible Table Results

When the user clicks the Show Selected Rows button, all visible headers from the table and all rows with the `row-selected-problems` class applied are retrieved and stored in an array. For each selected row, a new object is created and the values in each cell in the row are mapped to the corresponding key header. After

all rows have been processed, the updated array is returned, see Listing 4.34, and the table base is regenerated based on this new data. To undo the filtering, the Clear Data Table button can be clicked, returning the data to its original state. If the user proceeds to save this filtered data, it will overwrite the currently loaded result data.

```
export function UpdateResults(): object[] {
    const headers = Array.from(document.querySelectorAll(".thead-dark th")).map(
        (header) => {
            return header.textContent;
        }
    );

    const traceData = Array.from(
        document.querySelectorAll(".row-selected-problems")
    ).map((row) => {
        return Array.from(row.querySelectorAll("td")).reduce((obj, cell, j) => {
            obj[headers[j]] = cell.textContent;
            return obj;
        }, {});
    });

    return traceData;
}
```

Listing 4.34: `UpdateResults()`, filtering the rows.

4.6 Creating a New Chart

The function to create the chart can be seen in Listing 4.35, which makes it easier to handle the different types for all pages by using defined arguments to render the correct chart types. Each time this function is called, it first checks if a chart already exists and destroys it before creating a new one. As for the configuration, the chart has the arguments `type`, `data`, `label`, `title`, and optionally the `scale` options to customize each chart.

The types used in this system are `bar` and `line`, which are passed as parameters, with the former being used on the average solver time, number of iterations, number of nodes, and termination status pages, and the latter on the absolute performance profile and solver time pages. These two pages are also the pages where the scaling options are specified, which determine the titles on the x- and y-axis and the length of the x-axis. The following subsections explain how the data processing is implemented for these different graph types, as the result data cannot be used directly in the graphs.

```
export function CreateChart(
    type,
    data,
    label,
    title,
    scaleOptions = null
): void {
    if (myChart) {
        myChart.destroy();
    }

    const chartCanvas = document.getElementById("myChart") as HTMLCanvasElement;
    myChart = new Chart(chartCanvas, {
        type: type,
        data: {
            labels: typeof label === "string" ? [label] : label,
            datasets: data
        },
        options: {
            responsive: true,
            scales: scaleOptions,
            plugins: {
                title: {
                    display: true,
                    text: title
                }
            },
            tooltips: {
                // ...Shortened version.
            }
        }
    });
}
```

Listing 4.35: `CreateChart()`, base of the chart creation function.

4.6.1 Average Value Charts

The values are extracted from the specified category for each solver by a function. These categories are the `SolverTime`, `NumberOfNodes` and `NumberOfIterations` data fields. The statistical measures are calculated from the values and the average, minimum, maximum, standard deviation, sum and percentile values are returned. These returned values are mapped to a new format, where each entry becomes an object with a `label`, `data`, `borderColor`, and `backgroundColor`, according to Listing 4.36. The `label` is the key from the data, and the data used in the bar chart is the average of the values. If the chart is rendered together with the statistics table, the remaining statistical measures in the table are used.

```
const data = AnalyzeDataByCategory(traceData, category);
const colors = PickColor(20);
const chartData = Object.entries(data).map(([key, value], index) => ({
    label: key,
    data: [value.average],
    borderColor: colors[index % colors.length],
    backgroundColor: colors[index % colors.length]
}));

CreateChart(type, chartData, label, title);
StatisticsTable(data, title);
return chartData;
```

Listing 4.36: Creating the average value chart and statistics table.

4.6.2 Solver Status Chart

The solver statuses are extracted from each instance and grouped by solver, as shown in Listing 4.37. In total, a maximum of 14 different states can be mapped per solver. The data is displayed as a stacked bar graph, where each bar represents one solver.

```
export function ExtractStatusMessages(traceData: object[]): string[] {
    const result = [];
    const nameErrorMap = new Map();

    traceData.forEach((obj) => {
        const existingEntry = nameErrorMap.get(obj["SolverName"]);

        if (existingEntry) {
            existingEntry[obj["SolverStatus"]] =
                (existingEntry[obj["SolverStatus"]] || 0) + 1;
        } else {
            const newEntry = {
                SolverName: obj["SolverName"],
                [obj["SolverStatus"]]: 1
            };
            nameErrorMap.set(obj["SolverName"], newEntry);
            result.push(newEntry);
        }
    });
    return result;
}
```

Listing 4.37: `ExtractStatusMessages()`, extracting solver status messages.

4.6.3 Solver Time Chart

When creating the solver time data, the `ExtractAllSolverTimes()` function in Listing 4.38 reduces the result data to contain only the solver name and its corresponding time. If the time does not exist or is not a valid number, the instance is not added. This extracted data is then transformed, where each entry is mapped to an object with a `label` and `data` properties, where the x and y properties are set to the problem name and the average time, respectively. This transformed data is then used to create a graph with the data points, with the solver times on the y-axis and the problem names on the x-axis.

```
export function ExtractAllSolverTimes(traceData: object[]): object {
    const result = traceData.reduce(
        (
            acc: { [key: string]: { time: number; InputFileName: string }[] },
            obj: object
        ): { [key: string]: { time: number; InputFileName: string }[] } => {
            if (!acc[obj["SolverName"]]) {
                acc[obj["SolverName"]] = [];
            }
            if (obj["SolverTime"] !== "NA") {
                const time = Number(obj["SolverTime"]);
                if (!isNaN(time)) {
                    const inputFileName = obj["InputFileName"];
                    acc[obj["SolverName"]].push({ time, InputFileName: inputFileName });
                }
            }
            return acc;
        },
        {}
    );
    return result;
}
```

Listing 4.38: `ExtractAllSolverTimes()`, extracting all solver times.

4.6.4 Absolute Performance Profile Chart

Creating the stepped line graph for the absolute performance profile requires several steps. First, the solver times per solver must be extracted, which is done in the function in Listing 4.39. It first reduces the trace data array and returns an object that maps solver names to an array of solver times. If the default time or the gap limit is available from a configuration file, they will be set as the maximum default time and gap limit. If the user has entered values in the input elements on the graph page, those values will be used instead, along with the selected gap type

to compare to, which can be either the primal gap, dual gap, or solver gap. The filter conditions check whether the selected type of gap is less than the gap limit and whether the time exceeds the maximum default time threshold. If these conditions are met, the result is added to the object with the valid extracted values. If the solver time is not valid, it is set to the maximum default time.

```
const result = traceData.reduce(
    (
        acc: { [key: string]: { time: number; InputFileName: string }[] },
        obj: object
    ): { [key: string]: { time: number; InputFileName: string }[] } => {
        if (!acc[obj["SolverName"]]) {
            acc[obj["SolverName"]] = [];
        }
        if (!isNaN(Number(obj["SolverTime"]))) {
            if (
                obj[selectedGapType] <= primalGapToCompare &&
                Number(obj["SolverTime"]) <= defaultMaximumTime &&
                obj["SolverStatus"] === "Normal Completion"
            ) {
                acc[obj["SolverName"]].push({
                    time: Number(obj["SolverTime"]),
                    InputFileName: obj["InputFileName"]
                });
            }
        } else {
            acc[obj["SolverName"]].push({
                time: defaultMaximumTime,
                InputFileName: obj["InputFileName"]
            });
        }
        return acc;
    },
    {}
);
return result;
```

Listing 4.39: Getting all solver times which fulfills the criteria.

This performance profile data needs to be converted into a more suitable format for rendering in a stepped line graph. Therefore, the next step is to group the object entries into buckets based on their solver time. Those buckets that are unique are pushed into an array containing all unique values, as seen in Listing 4.40. After iterating through all the solver times, these unique keys are sorted and a new array is created with the cumulative counts of the unique keys. Next, the chart data is sorted and the scaling options are set in Listing 4.41. All labels are updated with the unique x-values, then the data is sorted by the labels and all x-axis values are updated from this sorted data.

```
const data = (
    Object.entries(absolutePerformanceProfileSolverTimes) as [
        string,
        { time: number; InputFileName: string }[]
    ][]
).map(([key, values]) => {
    const bucketCounts: { [key: number]: number } = {};
    const uniqueKeys: number[] = [];

    // Group times into buckets.
    values.forEach(({ time }) => {
        let bucket: number;

        if (time < 0.1) bucket = 0.1;
        else if (time < 1) bucket = 1;
        else bucket = Math.floor(time) + 1;

        if (!bucketCounts[bucket]) {
            bucketCounts[bucket] = 0;
            uniqueKeys.push(bucket);
        }
        bucketCounts[bucket]++;
    });

    uniqueKeys.sort((a: number, b: number) => a - b);

    let cumulativeCount = 0;
    const cumulativeCounts: { x: number; y: number }[] = [];

    // Create cumulative counts.
    uniqueKeys.forEach((key) => {
        cumulativeCount += bucketCounts[key];
        cumulativeCounts.push({ x: key, y: cumulativeCount });
    });

    return {
        label: key,
        data: cumulativeCounts,
        showLine: true,
        spanGaps: true
    };
});
```

Listing 4.40: Mapping the absolute performance profile data and creating the cumulative counts.

```
const chartData = data.map((dataset) => {
    dataset.data.forEach((point) => {
        if (!allLabels.includes(point.x)) {
            allLabels.push(point.x);
        }
    });

    const sortedData = allLabels
        .map((label) => {
            const point = dataset.data.find((d) => d.x === label);
            return point || { x: label, y: null };
        })
        .sort((a, b) => parseFloat(a.x) - parseFloat(b.x));

    sortedData.forEach((point) => {
        if (!allXValues.includes(point.x)) {
            allXValues.push(point.x);
        }
    });

    return { ...dataset, data: sortedData };
});

const scaleOptions = {
    x: {
        title: {
            display: true,
            text: "SolverTime [s]"
        },
        type: "linear",
        min: 0,
        max: defaultTime ? Number(defaultTime) + 100 : 1100,
        ticks: {
            stepSize: 50
        }
    },
    y: {
        title: {
            display: true,
            text: "Number of instances"
        }
    }
};

CreateChart(
    "line",
    chartData,
    null,
    `Absolute performance profile (${selectedGapType} <= ${
        gapLimit * 100 || 1
    }% and not failed)`,
    scaleOptions
);
```

Listing 4.41: Sorting the chart data and setting the scale options.

4.6.5 Download of Chart and Chart Data

The chart and its data can be downloaded as either as an image or as a JSON file. As for the first option, it can be saved simply by right-clicking on the chart and selecting Save as Image using the browser's or PWA's built-in menu option. When a chart function is called, it also returns the data used to render the chart. Clicking the save button for the chart data triggers the code in Listing 4.42, which converts the data into a JSON-formatted string and downloads it. If no such data is found, an error notification is displayed to the user.

```
downloadChartDataButton.addEventListener("click", () => {
    if (chartData) {
        const downloadAbleFile = JSON.stringify(chartData);
        const chd = new Blob([downloadAbleFile], { type: "application/json" });

        downloadChartDataButton.href = window.URL.createObjectURL(chd);
        downloadChartDataButton.download = "ChartData.json";
    } else {
        DisplayErrorNotification("No chart data found!");
    }
});
```

Listing 4.42: Downloading chart data.

4.7 Demo Data Activation

A demonstration mode has been developed to better demonstrate the capabilities of the application, eliminating the need to access the trace files or optional files. Upon clicking on the Demo-Data button, the scenario is activated when a dataset is loaded into local storage, see Listing 4.43, which is obtained from an exported object identified as `demoData`. The user can then view the data in a tabular format or in one of the graph types. PAVER [45] uses the same example data.

```
function ActivateDemoMode() {
    localStorage.setItem("UserConfiguration", JSON.stringify(demoData));
    localStorage.setItem("DemoData", "true");
    location.reload();
}
```

Listing 4.43: `ActivateDemoMode()`, activation of demo data.

Activation of the data is indicated to the user by reloading the browser window, displaying a new notification message that the demo data has been loaded, and

disabling and changing the color of the Demo-Data button to green. To exit the mode, the user can simply click on the Delete Data button, which will remove the demo data from local storage.

4.8 PWA Configuration

Listing 4.44 checks if the browser supports service workers, if the support exists, it registers the service-worker.js after loading the page. Upon successful registration, the scope of the service worker is logged. The interface `ServiceWorkerGlobalScope` [25], referenced within the service worker as self, will provide the install, fetch, and activate event handlers. Listing 4.45 shows these three events and their actions.

The service worker itself caches the files in Listing 4.46 for offline use once as it is installed, when the install event is fired [28]. The fetch event is fired on each network request by the application, if a match is found in the cache, then the cached response is returned [27]. The activate event is fired when the `ServiceWorkerRegistration` acquires a new `ServiceWorkerRegistration.active` worker [26]. It gets a list of all existing cache names and deletes those that are not included in the list.

```
export function RegisterServiceWorker(): void {
    if ("serviceWorker" in navigator) {
        window.addEventListener("load", () => {
            navigator.serviceWorker
                .register("/service-worker.js")
                .then((registration) => {
                    console.info(
                        "ServiceWorker registration successful with scope: ",
                        registration.scope
                    );
                })
                .catch((err) => {
                    console.warn("ServiceWorker registration failed: ", err);
                });
        });
    }
}
```

Listing 4.44: `RegisterServiceWorker()`, registering the service worker.

```
self.addEventListener("install", function (event) {
    event.waitUntil(
        caches
            .open(CACHE_NAME)
            .then(function (cache) {
                console.log("Opened cache.");
                return cache.addAll(filesToCache);
            })
            .catch(function (error) {
                console.error("Failed to open cache: ", error);
            })
    );
});

self.addEventListener("fetch", function (event) {
    event.respondWith(
        caches.match(event.request).then(function (response) {
            if (response) {
                return response;
            }
            return fetch(event.request).catch(function () {
                return new Response(
                    "You are offline. Some resources may not be available."
                );
            });
        })
    );
});

self.addEventListener("activate", (event) => {
    const cacheAllowlist = [CACHE_NAME];
    event.waitUntil(
        caches.keys().then((cacheNames) => {
            return Promise.all(
                cacheNames.map((cacheName) => {
                    if (cacheAllowlist.indexOf(cacheName) === -1) {
                        return caches.delete(cacheName);
                    }
                })
            );
        })
    );
});
```

Listing 4.45: `service-worker.js`, events.

```
const filesToCache = [
    "/",
    "report.html",
    "Dist/bundle.js",
    "Dist/main.css",
    "Src/CSS/tab_icon.png",
    "Src/Pages/absolute_performance_profile.html",
    "Src/Pages/average_solver_time.html",
    "Src/Pages/number_of_iterations.html",
    "Src/Pages/number_of_nodes.html",
    "Src/Pages/solver_time.html",
    "Src/Pages/termination_status.html"
];
```

Listing 4.46: `filesToCache`, all files cached for offline use.

4.9 Application Release Packaging

A release package of the application is created by running Gulp. It picks up the selected files that are needed to use the application and preserves their paths. Development configurations and source code are left out, as they are not needed to use the system. These selected files are archived together in a folder with a name and version number, as seen in Listing 4.47.

```
const { task, src, dest } = require("gulp");
const zip = require("gulp-zip");
const packageJSON = require("./package.json");
const version = packageJSON.version;

task("release", function () {
    return src(
        [
            "Dist/*.{js,css,txt}",
            "Docs/**/*",
            "Src/CSS/*.{png,jpg}",
            "Src/Pages/*.html",
            "report.html",
            "manifest.json",
            "service-worker.js"
        ],
        { base: "." }
    )
        .pipe(zip(`VIVA-release-${version}.zip`))
        .pipe(dest("Release"));
});
```

Listing 4.47: `gulpfile.js`, task for creating a release package of the application.

5. Testing

The purpose of testing our application is to ensure that it meets the functional requirements defined in the project scope and to identify defects that could cause the system to misbehave in use. Automated unit and UI testing is used, in addition to manual testing, that covers the requirements of the project. The former two can also be run on demand, as new functionality is added and the system evolves, reporting any broken functionality during updates.

Manual testing of the application was conducted in parallel with development, testing both the browser-based version and the PWA implementation simultaneously. The focus was on resolving most of the usability issues of the system and getting the application to work properly according to the requirements. As the system expanded in size and functionality, there was a need to move from manual testing to automated UI testing to cover all the different scenarios of using the application.

The test data used for unit testing is based on existing real-world benchmark data from the collection of benchmarks for MINLP problems [17], visible in Table 5.1. During the UI testing, trace files from this benchmark collection were used in conjunction with a solution file from MINLPLib to simulate real-world usage of the application.

5.1 PWA Testing

Testing of the PWA focused on offline capabilities and the installation and uninstallation of the PWA. Interaction with the table and charts were performed manually within the PWA to test that all the functional requirements were met and that there were no disparities to the other version of the system. The application passed the Google Lighthouse PWA audits with a valid service worker registration and a web application manifest that met the installability requirements [36]. Furthermore, the application passed all PWA optimization checks [37], ensuring a seamless user experience.

Table 5.1: MINLP problems for gap tests.

Solver	**Problem**	**PB-S**	**DB-S**	**PB-P**	**DB-P**	**Gap-S**	**Gap-P**	**PrimGap**	**DualGap**	**Dir**
Scp2804s	alan	2.925	2.925	2.925	2.925	0.0	0.0	0.0	0.0	min
Scp2804s	pedigree_ex485	-2.184812e4	-2.226831e4	-2.193157e4	-2.193157e4	1.92	0.0	0.38	1.54	min
Scp2804s	syn30h	1.381603e2	1.384918e2	1.381598e2	1.381598e2	0.24	0.0	0.0	0.24	max
Scp2804s	ball_mk4_15	Infinity	-Infinity	NaN	2.121536e1	Infinity	NaN	Infinity	NaN	min
Scp2804s	flay06m	6.693264e1	60	6.69328e1	6.517733e1	11.55	1.75547	NaN	NaN	min
aecp	gams01	2.842727e4	-1.170954e5	2.13802e4	1.851874e3	Infinity	19528.326	NaN	NaN	min
aecp	tls12	Infinity	9.207843	1.088e2	7.788233	Infinity	NaN	NaN	NaN	min
aoa	st_miqp4	Infinity	Infinity	-4.574e3	-4.574e3	0.0	NaN	Infinity	Infinity	min
baron	alan	2.925	2.922078	2.925	2.925	0.1	0.0	0.0	0.1	min
baron	smallinvDAXr1b150-165	8.813741e1	8.80825e1	8.810493e1	8.810493e1	0.06	0	0.04	0.03	min

Note: **Solver** is the name of the solver, **Problem** is the name of the input file,

PB-S and **DB-S** are the primal and dual bound for the solver, **PB-P** and **DB-P** are the primal and dual bound for the problem,
Gap-S is the solver gap, **Gap-P** is the problem gap, **PrimGap** is the primal gap, **DualGap** is the dual gap, and **Dir** is the direction.

5.2 Unit Testing

Accurate calculations are critical. The unit tests, 95 in total, comprehensively cover all functions that add or modify result values, and test them each in isolation by specifying input values and their expected output values to see if they behave correctly and produce valid results. Essentially, these were the functions that added the data fields and the functions that extracted the correct data for the line and bar charts.

Mock data tests the edge cases for all functions, while real data from benchmarks verifies that the calculations for the gap values were correct. Listing 5.1 provides a clear overview of how the tests where wrapped for each function. The `describe` block is used to organize edge cases and tests with a realistic setting. Each `it` block contains a test description, conditions, and expected values to ensure the correct behavior of the tested function.

```
describe("CalculateGap with mock-up data.", () => {
    it("should return Infinity when either a or b is Infinity", () => {
        expect(CalculateGap(Infinity, 1, "max")).toBe(Infinity);
        expect(CalculateGap(Infinity, 1, "min")).toBe(Infinity);
        expect(CalculateGap(1, Infinity, "max")).toBe(Infinity);
        expect(CalculateGap(1, Infinity, "min")).toBe(Infinity);
    });
});

describe("CalculateGap with real benchmark data.", () => {
    it("Solver baron and problem smallinvDAXr1b150-165", () => {
        expect(
            CalculateGap(
                minlpBenchmarksData[2]["PrimalBoundSolver"],
                minlpBenchmarksData[2]["DualBoundSolver"],
                minlpBenchmarksData[2]["Direction"]
            )
        ).toBe(minlpBenchmarksData[2]["Gap_Solver"]);
    });

    it("Solver Scp2804s and problem syn30h", () => {
        expect(
            CalculateGap(
                minlpBenchmarksData[4]["PrimalBoundProblem"],
                minlpBenchmarksData[4]["DualBoundProblem"],
                minlpBenchmarksData[4]["Direction"]
            )).toBe(minlpBenchmarksData[4]["Gap_Problem"]);
    });
});
```

Listing 5.1: `Function.test.ts`, example of the two types of unit tests used.

5.3 UI Testing

To ensure that the system would perform as expected, Playwright tests were run that simulated a variety of user flow scenarios. In practice, most of the tests were based on instructions to open the application, load a result file, and ensure that the visibility and status of elements were correct at each step of the process. For some scenarios, the order of the instruction differed to cover a variety of user flows. For example, instead of following the most common user flow, the test would load a result file, save it to the local storage, close and reopen the page, and then load the results from the local storage to assure that the test case was successful.

All the scenarios that were created can be seen in Table 5.3. By running all of these scenarios, it was possible to test all of the functional requirements presented in Chapter 2.2 and summarized in Table 5.2, where the order number is used in the covered requirement column for Table 5.3.

Table 5.2: Functional requirements.

Number	**Page Type**	**Requirement**
1	All pages	Selecting files
2	All pages	Confirming the upload
3	All pages	Showing error message on a failed upload
4	All pages	Showing success message on a non-failed upload
I. v.	All pages	Downloading the data
II. i.	Table page	Sorting
II. ii.	Table page	Filtering per column
II. iii.	Table page	Paginating
II. iv.	Table page	Searching in the displayed data
II. v.	Table page	Selecting results manually
II. vi.	Table page	Saving modified data in the system
III. i.	Plot pages	Visualizing

The code provided in Listing 5.2 demonstrates an example of using the Playwright library for browser automation. The browser is launched and navigates to the local HTML file specified by the path. From here, it interacts with the elements to load a result file and wait for the table to be displayed. Finally, it takes a screenshot of the view and closes the browser.

```
describe("UI tests", () => {
    let browser: Browser;
    let context: BrowserContext;
    let page: Page;

    beforeAll(async () => {
        browser = await chromium.launch({ headless: true });
        context = await browser.newContext();
        page = await context.newPage();
    });

    afterAll(async () => {
        await browser.close();
    });

    describe("Table Page", () => {
        const filePath = "../report.html";
        const absoluteFilePath: string = path.resolve(__dirname, filePath);
        const fileUrl = `file://${absoluteFilePath}`;

        beforeEach(async () => {
            await page.goto(fileUrl);
        });

        async function RunTableOperations(page: Page, notification: string): Promise<void> {
            await CheckNotification(page, "#alertNotification", notification);
            await WaitForElementAndClick(page, "#viewTableButton");
            await page.waitForSelector("#dataTableGenerated_wrapper", {
                state: "visible",
                timeout: 60000
            });
        }

        test("Handle multiple trace files", async () => {
            await page.goto(fileUrl);
            await uploadFile(page, [
                "./solvedata/TraceFiles/shotALL.trc",
                "./solvedata/TraceFiles/scipALL.trc",
                "./solvedata/TraceFiles/pavitoALL.trc"
            ]);
            await checkNotification(
                page,
                "#alertNotification",
                "Benchmark file succesfully loaded!"
            );
            await waitForElementAndClick(page, "#viewAllResultsButton");

            await page.waitForSelector("#dataTableGenerated_wrapper", {
                state: "visible",
                timeout: 5000
            });
        }, 10000);
    });
});
```

Listing 5.2: `UI.test.ts`, extracted UI test overview.

Table 5.3: UI test scenarios for Playwright.

Location	Scenario	Expected Results	Requirement Covered
Application	Navigation between pages	User can navigate between pages and confirm that titles are correct.	N/S
All Pages	Multiple trace files are loaded	User can load multiple trace files without errors and view the table or plot.	1, 2, 4
All Pages	Trace and additional files are loaded	User can load multiple file types without errors and view the table or plot.	1, 2, 4
All Pages	JSON-file is loaded	User can load a JSON-file without errors and view the table or plot.	1, 2, 4
All Pages	Interaction with local storage	After saving the data, the user should be able to reload the page and load it from local storage. Thereafter they can delete it.	1, 2, 4, 11
All Pages	Wrong file format is selected	Loading non-supported file format should notify the user by an alert notification.	1, 3
Table Page	Interact with filters	Selecting a number of rows from the table and clicking on the filter button should result in new modified data getting created, showing only the rows that were selected.	1, 2, 4, 10
Table Page	Button states changes after viewing the table	The state of the buttons should be changed after user has loaded a benchmarking file and clicked on view table button.	1, 2, 4, 10
Table Page	Download of the table	User can download the currently visible table.	1, 2, 4, 5
Table Page	Sorting and filtering the table	User can sort and filter in the columns.	1, 2, 4, 6, 7
Table Page	Pagination usage in the table	User can navigate by using pagination in the table.	1, 2, 4, 8
Table Page	Search for results in the table	user can search for results.	1, 2, 4, 9
Plot Pages	Table with statistics is viewable	Ensure that the table is displayed and the cells are not empty.	N/S
Plot Pages	Plot is viewable	The line or bar plot should be visible and the user can interact with it.	1, 2, 4, 12

6. System Overview

The purpose of this chapter is to describe the implementation of the application, including its features and user flow for both the table and plot pages. Furthermore, the linked documentation, which contains instructions for using the system and the source code, is reviewed.

6.1 General

The most recent version of the system, seen in Figure 6.1, is able to meet the requirements created during the planning phase. All these requirements combined provide the end user with an easy-to-navigate and functional system for analyzing and visualizing the results of optimization solvers. To start using the system, one can either install it as a PWA or use one of the HTML files to open it from the available release package. If the user chooses to install it as a PWA, they need to host it initially on their own server before proceeding with the installation.

The main type of result files for the system are trace files, along with supporting files containing information about the instances and their best known primal and dual bounds. These results can be saved to local storage or alternatively downloaded in JSON format, for reuse of the original or modified data.

For quick information regarding the file types, the user can click on the question mark on the right side of the navigation bar to open a modal, as shown in Appendix A and Figure A.5, with a brief summary of the supported file formats and the default header row that will be used if there are no headers provided in the loaded trace files. For a more detailed description of the functionality of the buttons, the end user can hover over any button to display a tooltip, regardless of the page they are on.

To learn and evaluate the features of the application, the demo data can be used by activating it from the link in the navigation bar. This will apply the results directly to the system, eliminating the need for the end user to acquire trace files to simply test the application's capabilities. The link will be disabled and the color

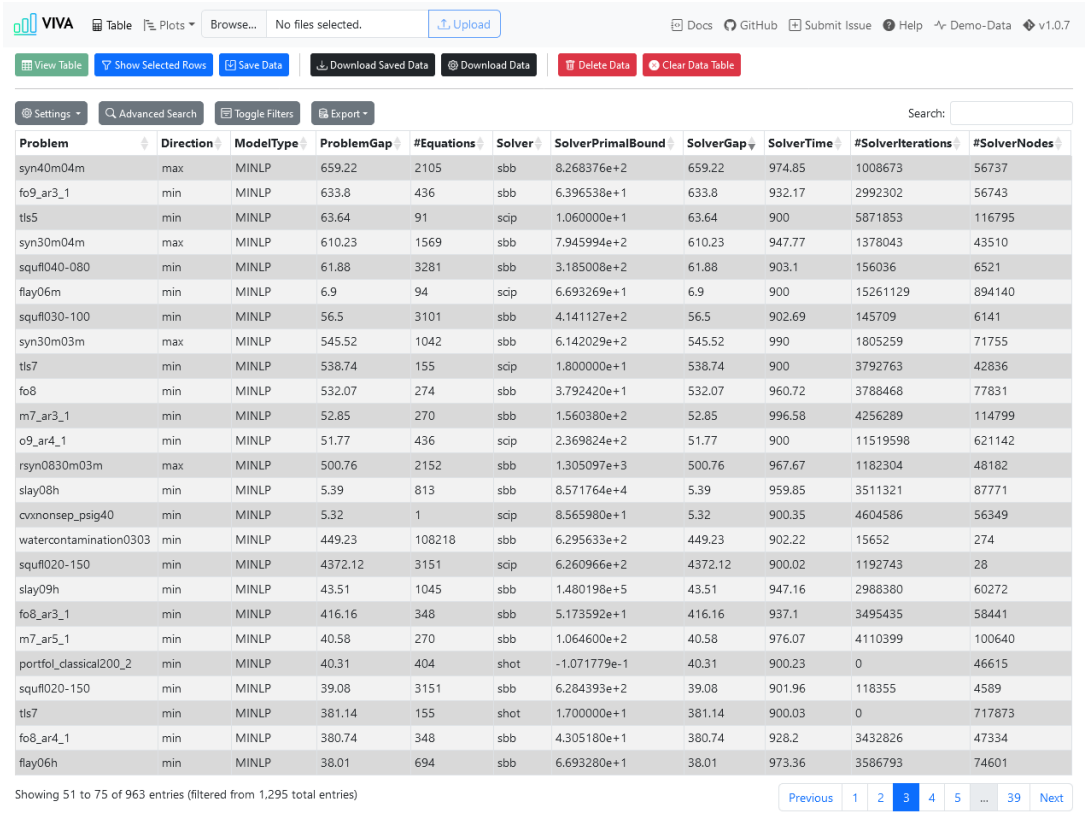


Figure 6.1: Table page of the application.

will change to green to indicate that the mode has been successfully enabled, as shown in Appendix A and Figure A.1.

The current version of the application is displayed as a link in the navigation bar. This link leads to the release page for the packaged version, where it can be downloaded.

The user flow is structured so that the main filtering is done on the table page. A user who wants to filter out specific instances must perform the operation on this page, save it, and switch back to the plot page of their choice. The table and plot pages are otherwise identical in functionality and work on the same principle when loading files.

6.2 User Flow

When the user is active on the table page or one of the plot pages and has selected and loaded files, they will be notified if the file(s) loaded successfully, as in Appendix A and Figure A.2, or if an error occurred, as in Figure A.3. Alterna-

tively, if the system has found a saved configuration in the local storage, they can proceed to the next step. On the data table page, the user can click on the View Table button, which creates a table base and applies the DataTable-wrapper to the base table. Also from this state, when the table is visible, the end user can perform actions from buttons that directly affect the interaction with the data table, described in Section 6.4.

If they are on any of the plot pages, they can choose to view the plot either in a line format if they are on the absolute performance profile page in Appendix C and Figure C.1 or the solver time page in Figure C.2, or in a bar format if they are on either the average solver time, number of nodes in, number of iterations, or termination status page, as shown in Figures C.3, C.4, C.5, and C.6, respectively. The data used for the charts can also be downloaded as a JSON file containing the dataset and options for the chart. In addition to the chart, each average bar plot page contains a smaller table with statistics for the modified outcome data used in the graphs. The categories highlighted are average, minimum, maximum, standard deviation, sum, and percentiles of the result type.

6.3 Buttons for the Table and Plot Pages

The following is a list of the buttons and the actions associated with them.

View Table (Table page) Creates and displays a table.

View Plot (Plot pages) Creates and displays a plot.

Show Selected Rows (Table page) Creates a new dataset with the visible headers and selected rows and displays it.

Save Data Saves the current data to local storage.

Download Saved Data Downloads the data from local storage as a JSON file.

Download Data Opens the modal seen in Appendix A and Figure A.4. In this view, the data can be downloaded as a JSON file by filtering selected solvers, with the option to set a default time and a gap limit for the absolute performance profile page.

Download Chart Data (Plot pages) Downloads the chart data used as a JSON file.

Delete Data Deletes stored data from local storage.

Clear Data Table (Table page) Clears the table and all filters applied.

6.4 Buttons for the Data Table

If the data table is visible, the user can utilize features from DataTables.js to interact with the results. These features are listed in order of appearance in Figure 6.1 and are referenced in Appendix B.

Settings Dropwdown Adjust the number of visible rows and change column visibility, as shown in Figure B.1.

Advanced Search Values can be searched for using the SearchBuilder plugin, which is visible in Figure B.2. This plugin allows for the search of values based on custom conditions.

Toggle Filters Filter types per column, visible in Figure B.3.

Export Dropdown Print a data sheet of the table, copy the data to the clipboard, or download a CSV file with the data, visible in Figure B.1.

Search Quick search for a matching result in the table.

Pagination Navigate between table pages.

6.5 Documentation

The documentation page consists of the requirements that the result files must meet to ensure proper functioning of the application, instructions on how to operate the application, and general knowledge about the entire project. Instructions are provided for developing the system and using the included scripts to facilitate the development phase and generate the application source code. The TypeDoc generated source code documentation is also accessible within this documentation. From here one can get a quick overview of the functions, variables, and their operations in the system. With all this knowledge, it will be easy to add new features to the application or update existing functionality as new web standards emerge.

7. Conclusion

The current state of the application is satisfactory, but there is considerable room for improvement and further development. This applies in particular to PWA support, where native features of the operating system could be used, such as notifying [23] the user when the data table has finished loading and storing even larger amounts of structured data using IndexedDB [22]. One could also expand the current configuration file with several settings to be saved, such as which data fields should be displayed by default in the table. The support for different file formats is also expandable, should the need arise. To improve the system usability, a feature could be added that allows the end user to select a problem library, e.g., MINLPLib, and obtain the primal and dual bounds and other problem properties for all instances instantly, without the need to load separate files.

Maintainability was achieved by using good documentation and linting rules in the project. The use of TypeScript during development was worthwhile to reduce the number of bugs in the application, even if it takes longer to set everything up correctly for development. Additionally, further work and updates to this system are possible, as a large part of the code could be reused if this application were converted to use a framework, thanks to being written in plain TypeScript. Alternatively, this project can be used as an inspiration to build a new system according to one's needs for analyzing and visualizing the results of optimization solvers.

7.1 Constraints

For the most part, the goals of the project have been vague and thus different functions have been developed, only to be replaced by a new idea subsequently. In principle, there has been some trial and error in adding features to the system. Hence, it was challenging to determine where to allocate development resources. Finding a good balance between all areas of the project proved to be difficult, resulting in a thesis that is broad in the technological scope rather than focused on a specific area.

It's difficult to determine if using a framework for this project, which had straightforward functionality, would have introduced any improvements considering the size of the system and the development requirements. Perhaps the flexibility of the system and the reduced dependence on third-party dependencies outweigh any negative aspects of not relying on a framework, even if the source code structure might have been less obscure.

A disadvantage of not using a framework led to some non-conventional designs for solving common problems that frameworks handle easily, for instance, managing the data in local storage for the current active page that the end user is located on, and the statuses of elements when enabling and disabling the buttons depending on the current state of the system. The navigation bar was hard coded too per page, which makes it slightly cumbersome to manually add new pages to the system, as one must modify the existing navigation structure on each page. Related to the previous point, when adding new pages to the application, one must update numerous segments in the source code. However, as this will rarely occur, it would not be too much of a negative point.

Initially, the supported file formats also included generated text files from PAVER, which are created from the trace files. However, it was determined that the number of end users who would use this format would be small. After reevaluating the option, it was removed from the project. In hindsight, better planning would have resulted in dropping the support for these files earlier, and focus should have been put on the trace format instead. As the project began, a considerable amount of time was devoted to parsing the text file structure. The current file management system can be extended to be compatible with other formats, as it was designed and developed with these text files of PAVER in mind.

7.2 Technical Difficulties

The technical difficulties encountered and accumulated in the development and implementation process were also significant. Although these identified issues did not hinder development completely, they were still annoyances that affected the thought process behind the choice of some techniques and solutions applied on the system. Those mentioned here are the ones that caused the most issues.

Occasionally, there would be some performance issues when the user uploaded an optional file to obtain additional information regarding the instances, as these files contained numerous data rows to be parsed and matched with the results, which

resulted in the application taking longer than usual to be able to display the data. A loading animation was added to indicate to the user that the system was still parsing the loaded information, as the table would not appear immediately.

On this issue, the number of information categories per instance would grow, which made the rendered table to be cumbersome to navigate due to the sheer number of columns that would appear, which results in the end user having to scroll a long distance horizontally to attain an overview of all instance information. Fortunately, the default visible column headers could be set in DataTables.js, and the user also has the option to select which columns they want to be visible in the table.

Computing the new result fields for instances which were terminated due to an error was difficult, as these calculations were based on their implementation in the PAVER tool. The source code for that system was based on Python and an older version of the Pandas library, which is a data analysis library. Therefore, some of the features were not applicable to the web-based environment this system was built in.

Local storage is handled differently between their implementation in browsers, when HTML files are loaded directly from the user's local file system, as there are no defined requirements for the behavior [30]. For instance, saving to the local storage when being active on the table page, and then navigating to a chart, can result in the data not being stored locally if the browser separates the data to different partitions, even if it is from the same origin. For another browser, however, it may be cross-compatible across all pages. In cases where the first mentioned example occurs, one must either save the data currently being worked on by downloading the configuration file, or by reloading the result files.

Despite the above-mentioned limitations and technical difficulties, the overall result is an application that can be used as a research support tool that performs its tasks very well, with a solid technological foundation for further development. It is available as an open-source project at https://doi.org/10.5281/zenodo.10363897 under the MIT license.

8. Svensk sammanfattning

Implementering av en applikation för analysering och visualisering av benchmarkresultat för optimeringslösare

Denna avhandling behandlar utvecklingen av en webbapplikation för visualisering och analysering av benchmarkresultat från optimeringslösare, för att möjliggöra jämförelse av skillnader i resultat. Optimeringslösare används till att hitta de bästa värdena för att lösa matematiska modeller, som representerar abstraktioner av verkliga problem. Dessa problem finns bland annat inom optimeringen av leveranskedjor och finansportföljer [15].

System som General algebraic modeling system (härefter GAMS) översätter optimeringsproblemen till en datorkod som kan lösas av optimeringslösare. De resultatfiler som genereras via GAMS innehåller information om problemet och lösarens resultat, samt vilka datafält som har fastställts att finnas i resultatfilerna [11]. Strukturen i en sådan fil, `shotALL.trc`, syns i kodlistning 1.1, där optimeringsproblemen är nvs11, nvs12 och nvs15, medan lösaren för alla dessa instanser är SHOT [19]. Resterande datafält beskrivs i tabell 1.1. Målet med applikationen som utvecklas är att kunna läsa in resultatfilerna och utifrån dem skapa dynamiska tabeller och grafer, varpå slutanvändare behändigt ska kunna analysera och interagera med dem.

För jämförelse av optimeringslösare finns verktyg som PAVER 2.0 [50], som kan visualisera och sammanställa statiska benchmarkresultat [2], och Mittelmann-Plots [32] för jämförelse av resultat via interaktiva diagram. Dock är den förstnämnda inte optimal när man vill filtrera fram de resultat som ska framhävas tydligt, eftersom de är statiska. Hanteringen av statiska tabeller och diagram är besvärlig om resultaten presenteras med irrelevanta kategorier för det aktuella användningsområdet. Därtill är PAVER 2.0 utmanande att installera, eftersom det kräver specifika versioner av Python och de tillhörande programbiblioteken. På grund av dessa brister utvecklas projektet som beskrivs i denna avhandling som en webbapplikation för smidigare tillgänglighet och underhåll, samt med stöd för dynamiska resultat genom filtreringsfunktionalitet.

8.1 Krav

Ett av de funktionella krav som används som grund för att utforma systemet, är att applikationen bör kunna läsa in ett flertal resultatfiler, samt vid behov filer med metadata. Resultaten ska visas i både tabell- och grafformat på separata sidor. Dessutom bör det finnas möjlighet till filtrering och sortering av alla data enligt behov. Slutligen ska användaren kunna spara allting i en modifierad resultatfil, som antingen kan användas i framtida bruk eller för distribution.

De icke-funktionella kraven är att applikationen ska vara användarvänlig och enkel att ta i bruk för slutanvändare. De resultatfiler som används ska expanderas med nya datafält på tabellsidan. Baserat på kända datafält och deras värden kommer de nya fälten, som listas i tabell 1.2, att beräknas. Beräkningarna för dessa datafält grundar sig på deras implementering i PAVER 2.0.

8.2 Utvecklingsplanering

För att underlätta utvecklingsprocessen väljs TypeScript [43] som systemkod för applikationen. Detta medför statisk typindelning som kommer att trygga underhållbarheten för projektet på lång sikt. Beslutet att inte använda något ramverk grundade sig på att de uppgifter som systemet ska sköta är så pass få att en flersidesapplikation borde mer än väl vara lämplig för detta projekt. I princip behövs det bara ett par funktioner vid läsning av resultatfilerna, utförandet av beräkningar på den inlästa informationen, samt renderingen av resultatet i antingen tabell- eller grafformat.

De systemkrav som utformats för webbapplikationen sammanfaller med de krav som behövs för implementeringen av systemet som en Progressive Web Application, förkortat PWA [52]. Därför är det enkelt att inkludera stöd för denna typ av applikation, vilket gör att systemet kan köras som en fristående applikation.

Eftersom programkoden ska vara kompatibel med webbläsare måste TypeScript kompileras till JavaScript genom programbiblioteket Webpack. Samtidigt paketeras även projektets alla TypeScriptfiler till en enskild optimerad och minimerad JavaScript-fil, som importeras per sida i applikationen från arkivet Dist enligt figur 3.1. I samma figur visas vilka filer som blir paketerade för publicering i ett applikationspaket och vilka filer som kommer att sparas lokalt när systemet installeras som en PWA.

De programbibliotek som kommer att användas vid framställning och visualisering av benchmarkresultaten är tabellbiblioteket DataTables.js [39] och grafbiblioteket Chart.js [4]. Den förstnämndas egenskaper kommer att användas på tabellsidan, med funktionalitet som avancerad sortering, filtrering och sökning av resultat. Den sistnämnda tillämpas för rendering av grafer, där de olika graftyperna kommer att bestå av linje- och stapeldiagram.

8.3 Implementering

Biblioteket Bootstrap används för styling och layout av sidorna i applikationen. Strukturen består av ett övre navigationsfält, knappar och den sektion som antingen har en tabell eller graf, beroende på vald sida i enlighet med innehållet i kodlistning 4.1. Alla nödvändiga element refereras till i systemkoden, samt deras typ, för att kunna nås av olika kodmoduler enligt kodlistning 4.2.

Användarflödet utförs i filen `Main.ts` som är programkodens ingångspunkt, där innehållet i figur 4.3 illustrerar de sektioner som denna fil innehåller. Sammandraget importeras först de funktioner, filer och tillägg som behövs för systemets funktionsduglighet, innan någon annan kod exekveras. Funktionen `RegisterService Worker()` (kodlistning 4.44) aktiveras för att kontrollera om applikationen kan installeras som en PWA [24], om den använda webbläsaren stöder det. De globala variablerna initialiseras och därefter väntar systemet på att en eller flera filer ska läsas in, innan vidare funktionalitet aktiveras. När detta har skett bearbetas informationen, som erhållits genom kodlistning 4.12, från de inlästa filerna beroende på de filtyper som har valts. Slutligen aktiveras knapparna för systemet, som alla har sina dedikerade ändamål, exempelvis visning av tabell eller nerladdning av resultatdata.

Bearbetningen av datafilerna separeras enligt deras filändelse. Då en eller flera trace-filer används kalkyleras de nya datafälten och läggs till i tabellen. Om filer som innehåller metadata används, kommer även dessa att fusioneras och användas i resultaten, se kodlistning 4.8. Ifall en JSON-konfigurationsfil används, sparas den automatiskt i webbläsarens lokala lagring utan att bearbetas och informationen läses in enligt kodlistning 4.7.

Bastabellen för den dedikerade tabellsidan och statistiktabellerna för grafsidorna genereras genom att applicera resultatdata i tabellelement enligt kodlistning 4.29. DataTables.js implementeras genom att appliceras ovanpå den skapade tabellen på den dedikerade tabellsidan, enligt kodlistning 4.33. I denna konfigurationsfunk-

tion tillämpas inställningar och de funktioner som DataTables.js använder sig av för utökning av tabellinteraktionen med olika alternativ.

Skapandet av linje- och stapeldiagram utgår från samma basfunktion, kodlistning 4.35, där Chart.js renderar ett nytt diagram varje gång funktionen anropas. Argumenten för denna funktion är diagramtyp, de bearbetade resultatdata som ska användas, etiketter, titel och skalningsinställningar. Kodlistning 4.38 visar processen för bearbetning av data, där lösarnas tidsresultat extraheras för användning i ett linjediagram.

8.4 Testning

Testningen av systemet som helhet utförs med både manuell och automatiserad testning, bestående av enhetstestning och testning av användargränssnitt med testbiblioteken Jest [31] och Playwright [35]. Den manuella testningen används under hela utvecklingsskedet, medan den automatiserade testningen tas i bruk mot slutskedet av utvecklingen, efter att applikationen har växt i storlek och funktionalitet.

Avsikten med enhetstesterna är att verifiera de funktioner som lägger till och modifierar datafält, samt de som extraherar data för linje- och stapeldiagram. Dessa använder sig av data baserade på tillgängliga benchmarkresultat från en samling av MINLP-problem [17], presenterade i tabell 5.1. Användargränssnittstestningens verifierar de funktionella kraven som återfinns i tabell 5.2 och säkerställer att dessa fullgörs i applikationen. Alla de testfall som används presenteras i tabell 5.3.

8.5 Slutsats

Sammantaget är utvecklingsresultatet en webbapplikation som är kapabel att läsa in resultatfiler och låter användare interagera med dessa resultat i dynamiska tabell- och grafformat, dessutom är det möjligt att spara resultaten både från tabellsidan och grafsidorna. Även om riktlinjerna för projektet har varit vaga, har de utformade systemkraven uppfyllts och verifierats genom den omfattande testningen. Därtill kan applikationens funktionalitet utökas eller återanvändas i nya projekt, tack vare dess utveckling i TypeScript utan ett specifikt ramverk. Stödet för resultatfiler kan smidigt breddas till nya filtyper, med tanke på den separata hanteringen av data- och tilläggsfiler med metadata.

References

[1] "Bootstrap". (2023), [Online]. Available: https://getbootstrap.com/ (visited on 11/27/2023).

[2] M. R. Bussieck, S. P. Dirkse, and S. Vigerske, "PAVER 2.0: an open source environment for automated performance analysis of benchmarking data", *Journal of Global Optimization*, vol. 59, pp. 259–275, 2014.

[3] M. R. Bussieck, A. S. Drud, and A. Meeraus, "MINLPLib — a collection of test models for mixed-integer nonlinear programming", *INFORMS Journal on Computing*, vol. 15, no. 1, pp. 114–119, 2003.

[4] "Chart.js". (2023), [Online]. Available: https://www.chartjs.org/ (visited on 04/23/2023).

[5] "Cplex". (2023), [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer (visited on 11/21/2023).

[6] "ESLint". (2023), [Online]. Available: https://eslint.org/ (visited on 11/29/2023).

[7] S. S. Gaikwad and P. Adkar, "A Review Paper on Bootstrap Framework", *IRE Journals*, vol. 2, no. 10, pp. 349–351, 2019.

[8] GAMS Software GmbH. "Model Status". (2023), [Online]. Available: https://www.gams.com/44/docs/UG_GAMSOutput.html#UG_GAMSOutput_ModelStatus (visited on 09/06/2023).

[9] GAMS Software GmbH. "Solver Status". (2023), [Online]. Available: https://www.gams.com/44/docs/UG_GAMSOutput.html#UG_GAMSOutput_SolverStatus (visited on 09/06/2023).

[10] GAMS Software GmbH. "System Overview". (2023), [Online]. Available: https://www.gams.com/products/gams/gams-language/ (visited on 11/21/2023).

[11] GAMS Software GmbH. "Trace File". (2023), [Online]. Available: https://www.gams.com/latest/docs/UG_SolverUsage.html#UG_SolverUsage_TraceFile (visited on 10/02/2023).

[12] A. Gleixner *et al.*, "MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library", *Mathematical Programming Computation*, vol. 13, no. 3, pp. 443–490, 2021.

[13] "Gulp.js". (2023), [Online]. Available: https://gulpjs.com/ (visited on 08/24/2023).

[14] "Gurobi". (2023), [Online]. Available: https://www.gurobi.com/solutions/gurobi-optimizer/ (visited on 11/21/2023).

[15] J. Kallrath, *Modeling languages in mathematical optimization*. Springer Science & Business Media, 2004, vol. 88.

[16] "Lighthouse Overview". (2022), [Online]. Available: https://developer.chrome.com/docs/lighthouse/overview/ (visited on 11/21/2023).

[17] A. Lundell. "MINLP benchmarks – Convex MINLP trace files". (2018), [Online]. Available: https://github.com/andreaslundell/minlpbenchmarks/tree/master/2018-10-ConvexMINLP/TraceFiles (visited on 07/19/2023).

[18] A. Lundell. "MINLP benchmarks – Documentation". (2018), [Online]. Available: https://andreaslundell.github.io/minlpbenchmarks/2018-10-ConvexMINLP/PaverReports/ALL/documentation.html (visited on 08/06/2023).

[19] A. Lundell. "MINLP benchmarks – shotALL.trc". (2018), [Online]. Available: https://github.com/andreaslundell/minlpbenchmarks/blob/master/2018-10-ConvexMINLP/TraceFiles/shotALL.trc#L118-L120 (visited on 10/02/2023).

[20] "Math.js". (2022), [Online]. Available: https://mathjs.org/ (visited on 04/23/2023).

[21] MDN-contributors. "CORS errors". (2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors (visited on 01/30/2022).

[22] MDN-contributors. "IndexedDB API". (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (visited on 10/16/2023).

[23] MDN-contributors. "Integrate with the operating system". (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Best_practices#integrate_with_the_operating_system (visited on 10/16/2023).

[24] MDN-contributors. “Making PWAs installable”. (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Making_PWAs_installable (visited on 08/13/2023).

[25] MDN-contributors. “ServiceWorkerGlobalScope”. (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope (visited on 08/19/2023).

[26] MDN-contributors. “ServiceWorkerGlobalScope: activate event”. (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope/activate_event (visited on 08/19/2023).

[27] MDN-contributors. “ServiceWorkerGlobalScope: fetch event”. (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope/fetch_event (visited on 08/19/2023).

[28] MDN-contributors. “ServiceWorkerGlobalScope: install event”. (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope/install_event (visited on 08/19/2023).

[29] MDN-contributors. “Storage quotas and eviction criteria”. (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_eviction_criteria (visited on 08/13/2023).

[30] MDN-contributors. “Window: localStorage property”. (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage#description (visited on 10/10/2023).

[31] Meta Platforms and affiliates. “Jest”. (2023), [Online]. Available: https://jestjs.io/ (visited on 08/21/2023).

[32] M. Miltenberger. “Visualizations of Mittelmann benchmarks”. (2023), [Online]. Available: https://mattmilten.github.io/mittelmann-plots/ (visited on 05/31/2023).

[33] “MINLPLib”. (2023), [Online]. Available: https://minlplib.org/index.html (visited on 08/15/2023).

[34] “MINLPLib - Instances”. (2023), [Online]. Available: https://www.minlplib.org/instances.html (visited on 08/15/2023).

[35] “Playwright”. (2023), [Online]. Available: https://playwright.dev/ (visited on 08/21/2023).

[36] "PWA Audits: Installable". (2023), [Online]. Available: https://developer.chrome.com/en/docs/lighthouse/pwa/#installable (visited on 10/14/2023).

[37] "PWA Audits: PWA optimized". (2023), [Online]. Available: https://developer.chrome.com/en/docs/lighthouse/pwa/#pwa-optimized (visited on 10/14/2023).

[38] "Scip". (2023), [Online]. Available: https://www.scipopt.org/index.php#about (visited on 11/21/2023).

[39] SpryMedia. "DataTables Reference Options". (2023), [Online]. Available: https://datatables.net/reference/option/ (visited on 03/24/2023).

[40] "The Supporting Hyperplane Optimization Toolkit". (2023), [Online]. Available: https://shotsolver.dev/shot/ (visited on 11/21/2023).

[41] "Trace File Parameters". URL accessed on 2023-03-23; URL is no longer available. (2023), [Online]. Available: https://www.gamsworld.org/performance/trace.htm (visited on 03/23/2023).

[42] "TypeDoc". (2023), [Online]. Available: https://typedoc.org/ (visited on 08/21/2023).

[43] "TypeScript". (2023), [Online]. Available: https://www.typescriptlang.org/ (visited on 11/27/2023).

[44] K. Varelas *et al.*, "Benchmarking large-scale continuous optimizers: The bbob-largescale testbed, a COCO software guide and beyond", *Applied Soft Computing*, vol. 97, p. 106 737, 2020, ISSN: 1568-4946. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S156849462030675X.

[45] S. Vigerske. "PAVER 2.0: MIPLIB2010". (2014), [Online]. Available: https://github.com/coin-or/Paver/tree/master/examples/miplib2010 (visited on 03/09/2023).

[46] S. Vigerske. "PAVER 2.0: Direction". Referenced code lines: 261-L262. (2019), [Online]. Available: https://github.com/coin-or/Paver/blob/783a6f5d0d3782a168d0ef529d01bcbda91ea8a4/src/paver/readgamstrace.py#L261-L262 (visited on 04/24/2023).

[47] S. Vigerske. "PAVER 2.0: Gaps". Referenced function: `computeGap(upper, lower, tol = 1e-9)`. (2019), [Online]. Available: https://github.com/coin-or/Paver/blob/783a6f5d0d3782a168d0ef529d01bcbda91ea8a4/src/paver/utils.py#L46-L59 (visited on 04/24/2023).

[48] S. Vigerske. "PAVER 2.0: Primal and Dual Bound". Referenced code lines: 275-282. (2019), [Online]. Available: https://github.com/coin-or/

Paver/blob/783a6f5d0d3782a168d0ef529d01bcbda91ea8a4/src/paver/readgamstrace.py#L275-L282 (visited on 04/24/2023).

[49] S. Vigerske. "PAVER 2.0: TermStatus". Referenced dictionary: SOLVERSTAT2TERMSTAT. (2019), [Online]. Available: https://github.com/coin-or/Paver/blob/783a6f5d0d3782a168d0ef529d01bcbda91ea8a4/src/paver/readgamstrace.py#L90-L103 (visited on 04/24/2023).

[50] S. Vigerske. "PAVER 2.0". (2023), [Online]. Available: https://github.com/coin-or/Paver (visited on 03/23/2023).

[51] "Webpack". (2023), [Online]. Available: https://webpack.js.org/ (visited on 11/27/2023).

[52] "What are Progressive Web Apps?" (2023), [Online]. Available: https://web.dev/articles/what-are-pwas (visited on 11/21/2023).

Appendix A: Application Elements

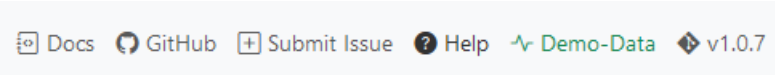


Figure A.1: Active demo data indicator.

Figure A.2: Success notification.

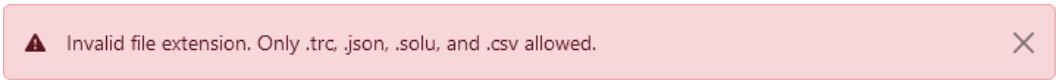


Figure A.3: Error notification.

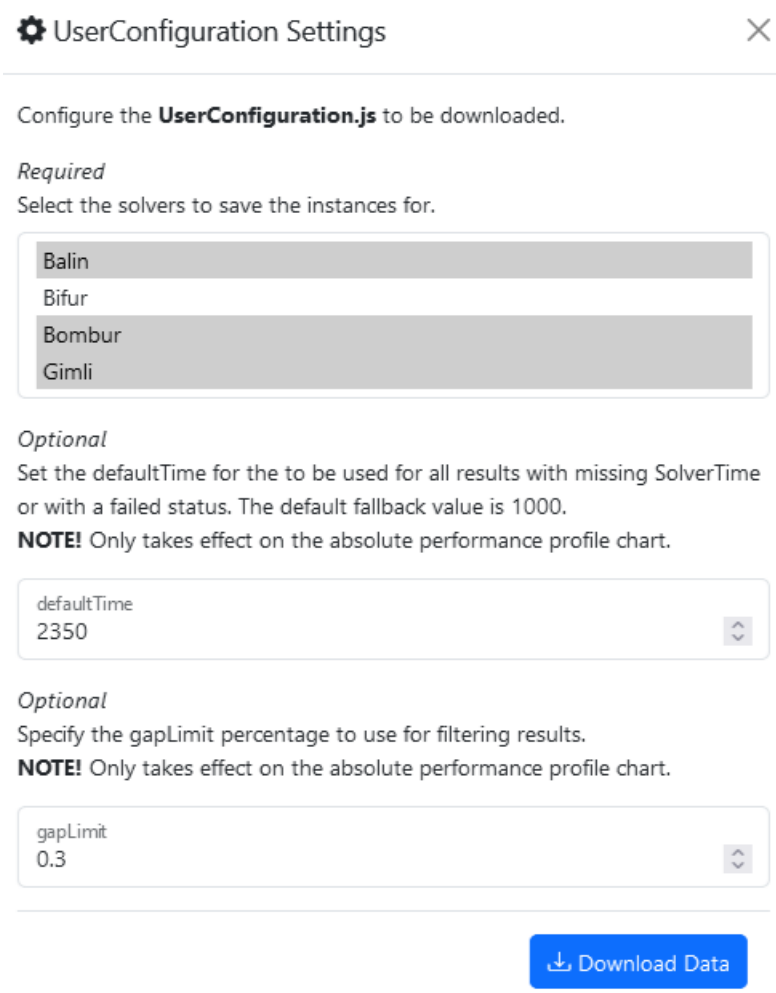


Figure A.4: Configuration modal.

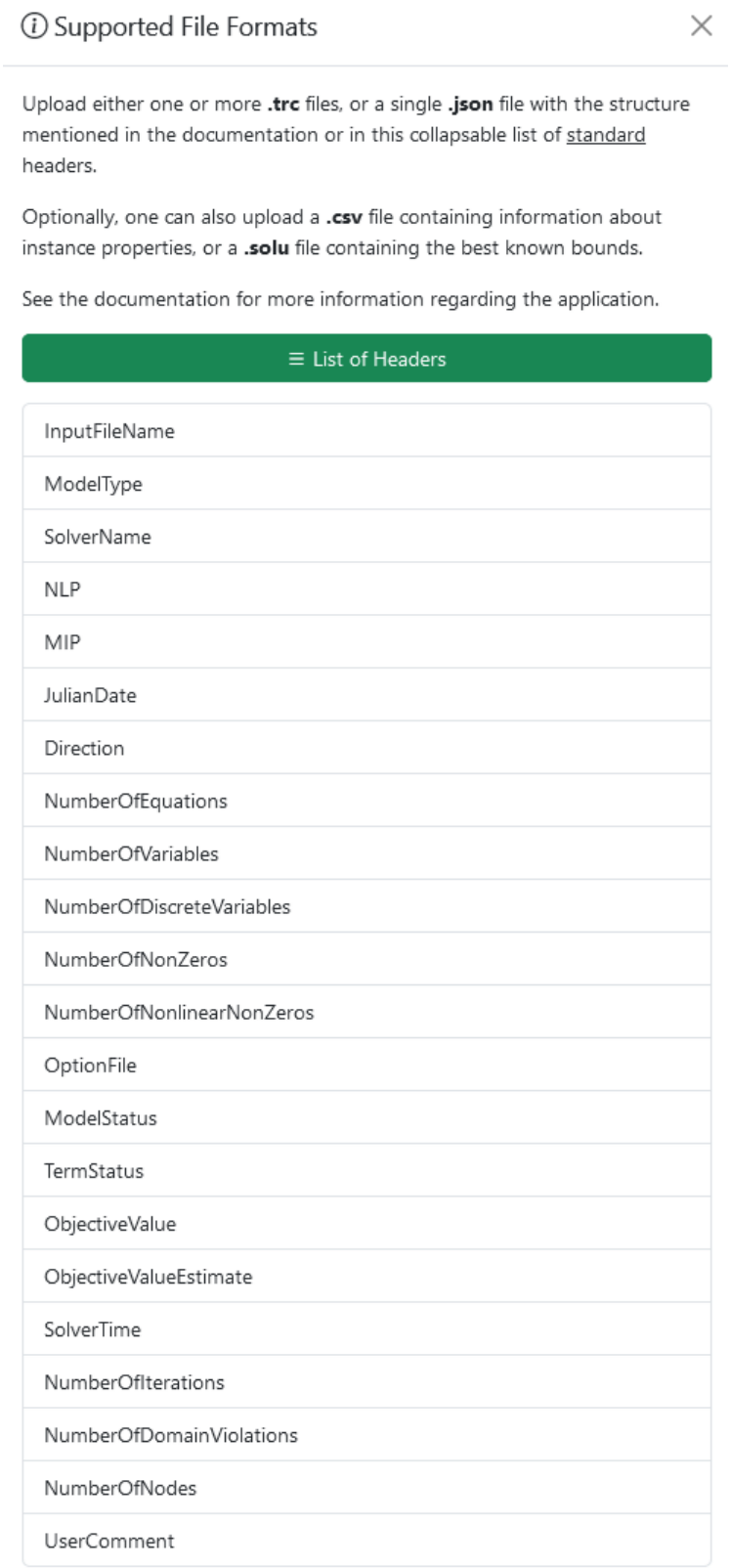


Figure A.5: Modal window with the supported file information.

Appendix B: Data Table Elements

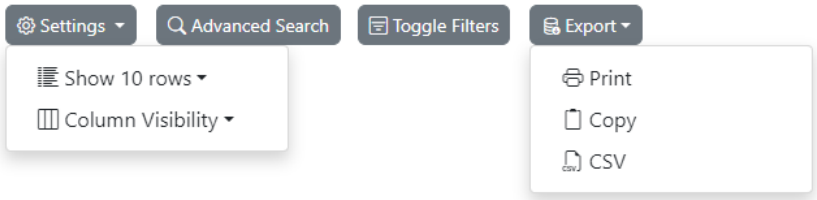


Figure B.1: Drop-down menus.

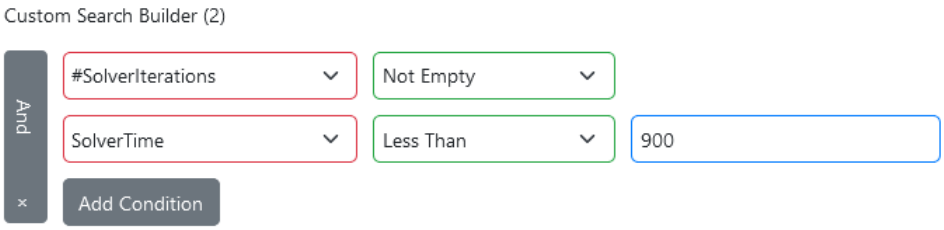


Figure B.2: Custom condition filtering.

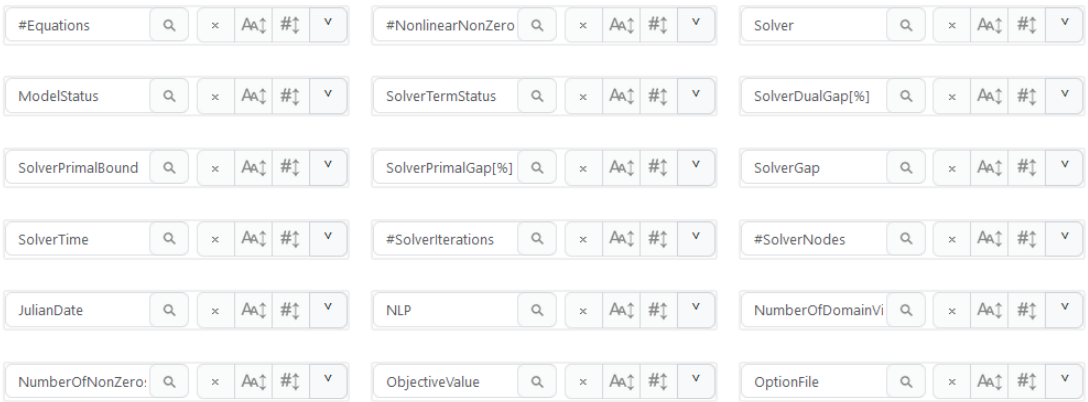


Figure B.3: Filter options available for each column.

Appendix C: Chart Types

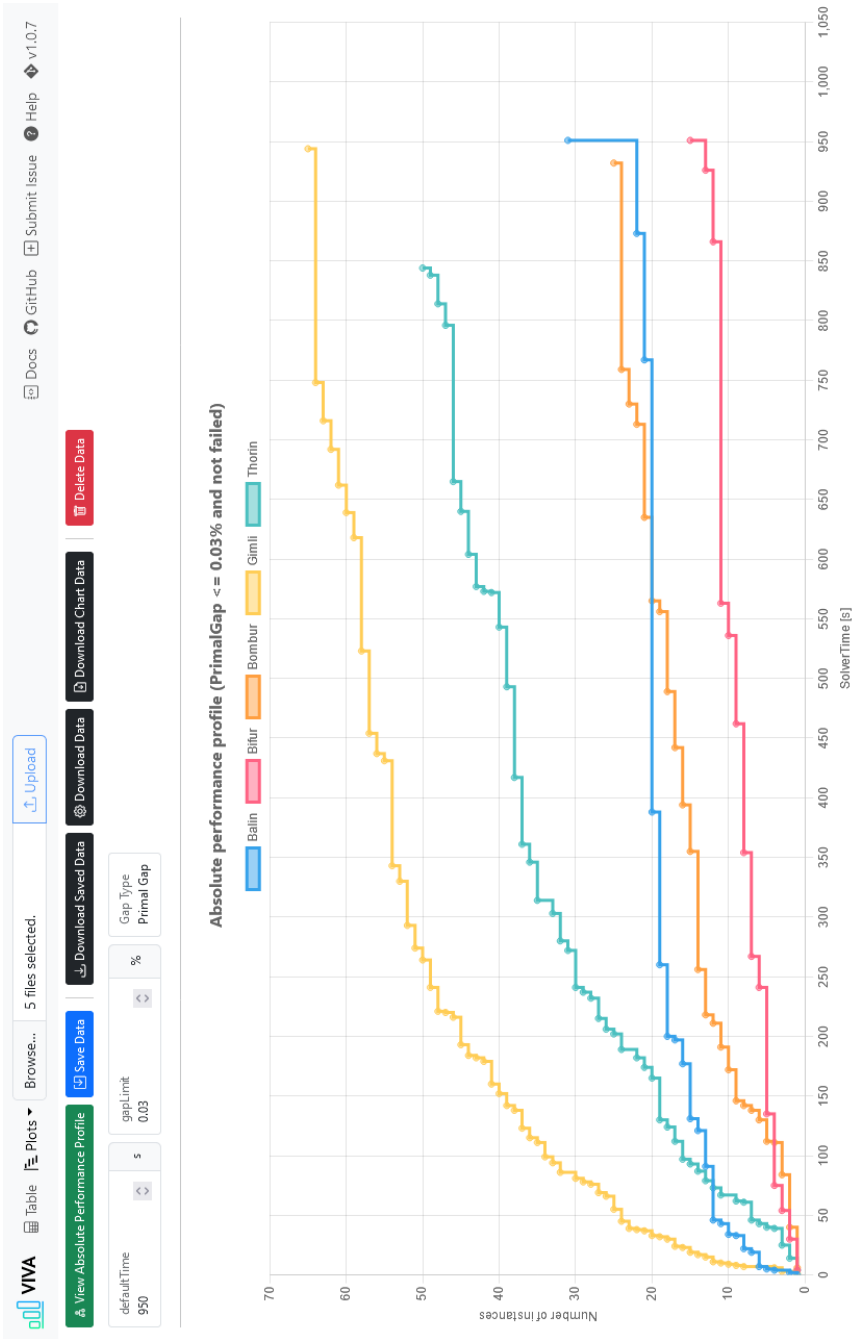


Figure C.1: Absolute performance profile plot.

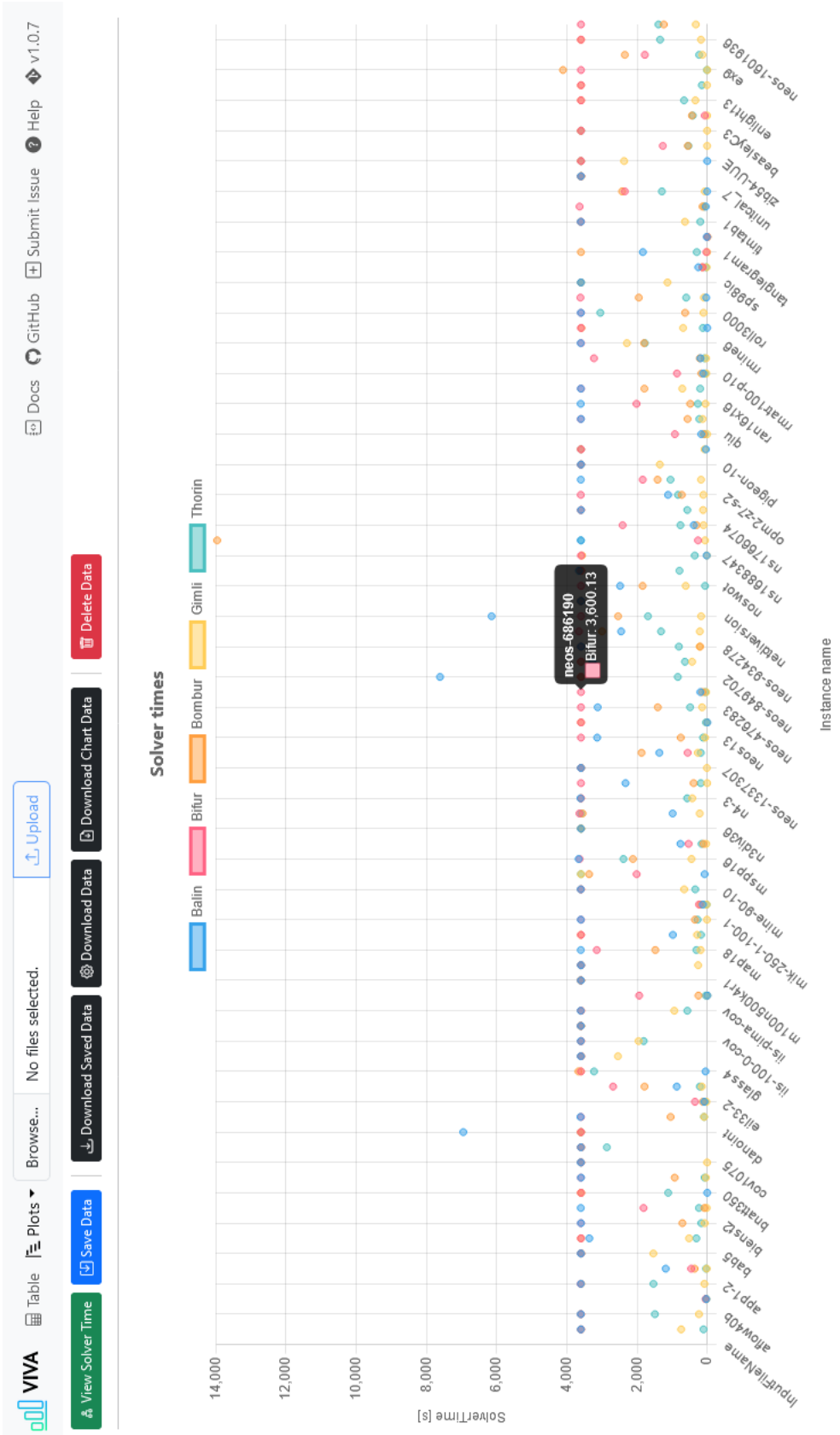
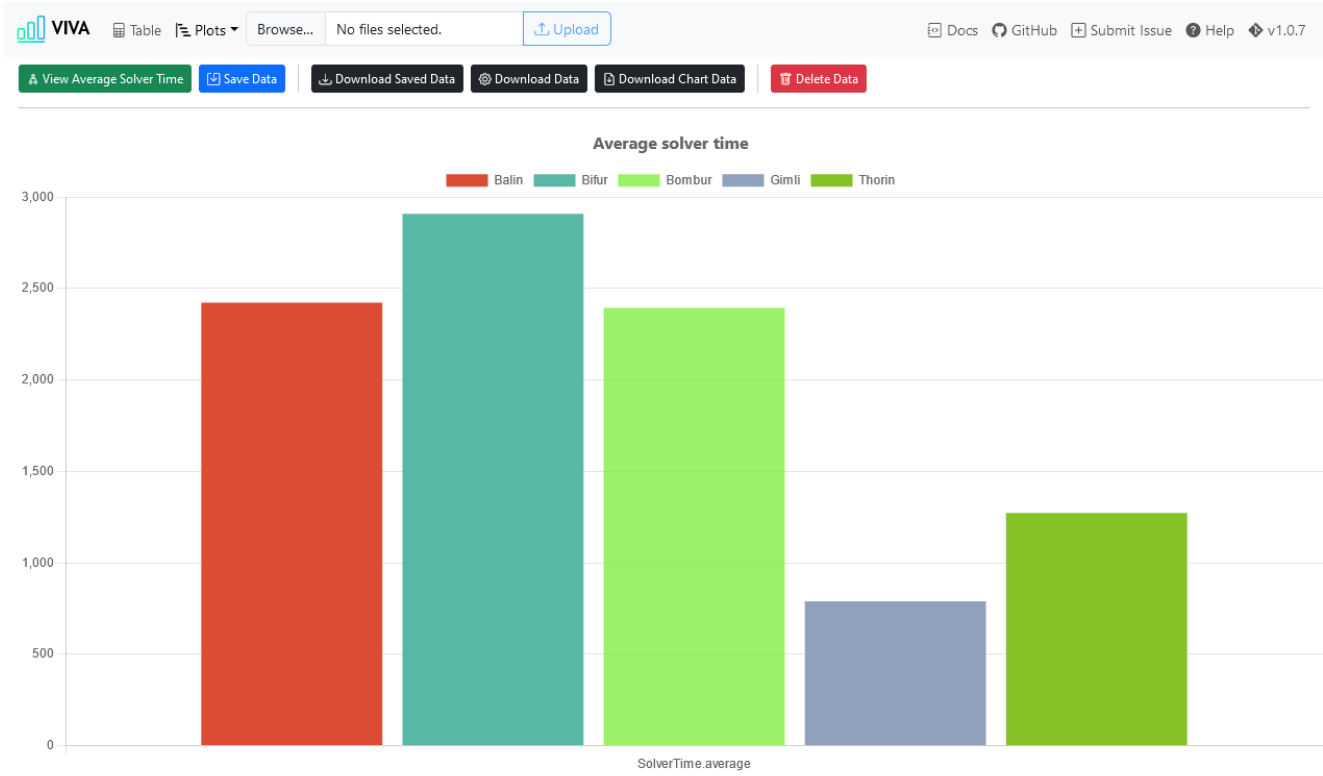


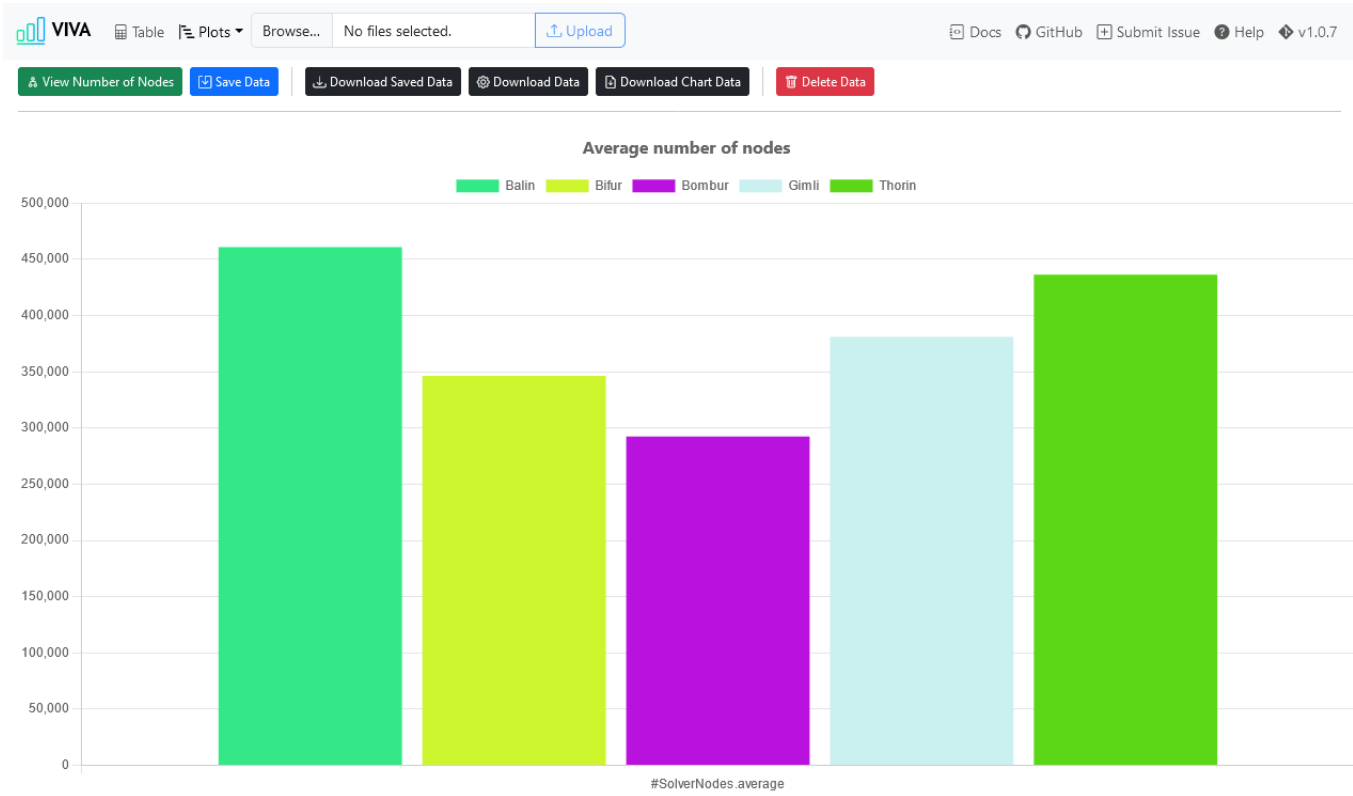
Figure C.2: Solver times per problem plot.



Solver	average	min	max	std	sum	percentile_10	percentile_25	percentile_50	percentile_75	percentile_90
Balin	2422.934	0.03	7614.11	1807.206	188988.8	29.357	214.645	3600.805	3604.72	3607.584
Bifur	2908.907	3.78	3659.06	1220.084	247257.1	490.88	2414.84	3600.01	3600.18	3602.216
Bombur	2394.604	5.21	13959.96	1932	208330.5	161.296	599.465	3369.19	3600.065	3600.988
Gimli	789.4134	0.73	3600.23	1246.125	68678.97	8.032	34.405	159.1	703.415	3600.03
Thorin	1273.31	3.49	3600.27	1405.026	110778	61.194	169.07	572.35	2626.845	3600

Average solver time statistics.

Figure C.3: Average solver time.



Solver	average	min	max	std	sum	percentile_10	percentile_25	percentile_50	percentile_75	percentile_90
Balin	460890	0	4875883	999438	35949420	1	372.75	24441.5	340276.3	1382808
Bifur	346410.1	0	5477335	970863.2	29444860	169.4	1678	17249	91658	1020535
Bombur	292347.1	0	7637922	1003292	25434200	286.6	3515.5	30640	167717.5	442150.2
Gimli	380887.7	0	8511016	1269020	33137230	60.4	1077.5	9039	80858.5	655578
Thorin	436415.8	1	7221264	1242145	37968180	207.8	1948.5	24221	324204	933520.4

Average number of nodes statistics.

Figure C.4: Average number of nodes plot.

Solver	average	min	max	std	sum	percentile_10	percentile_25	percentile_50	percentile_75	percentile_90
Balin	8425960	0	50761710	12724880	657224900	22523.6	209851	2576107	12524500	25756720
Bifur	18068930	38564	132721100	21960850	1535859000	892791.8	4773869	11596900	20506180	47421130
Bombur	12092100	0	90886270	16473820	1052013000	79051.4	729250	6941510	15790320	39041490
Gimli	8495599	0	161394200	22326580	739117100	29921.2	98903.5	689981	3708141	17550330
Thorin	7667601	0	122138000	17395660	667081300	54424.2	193368.5	1295042	5091299	28023820

Average number of iterations statistics.

Figure C.5: Average number of iterations plot.

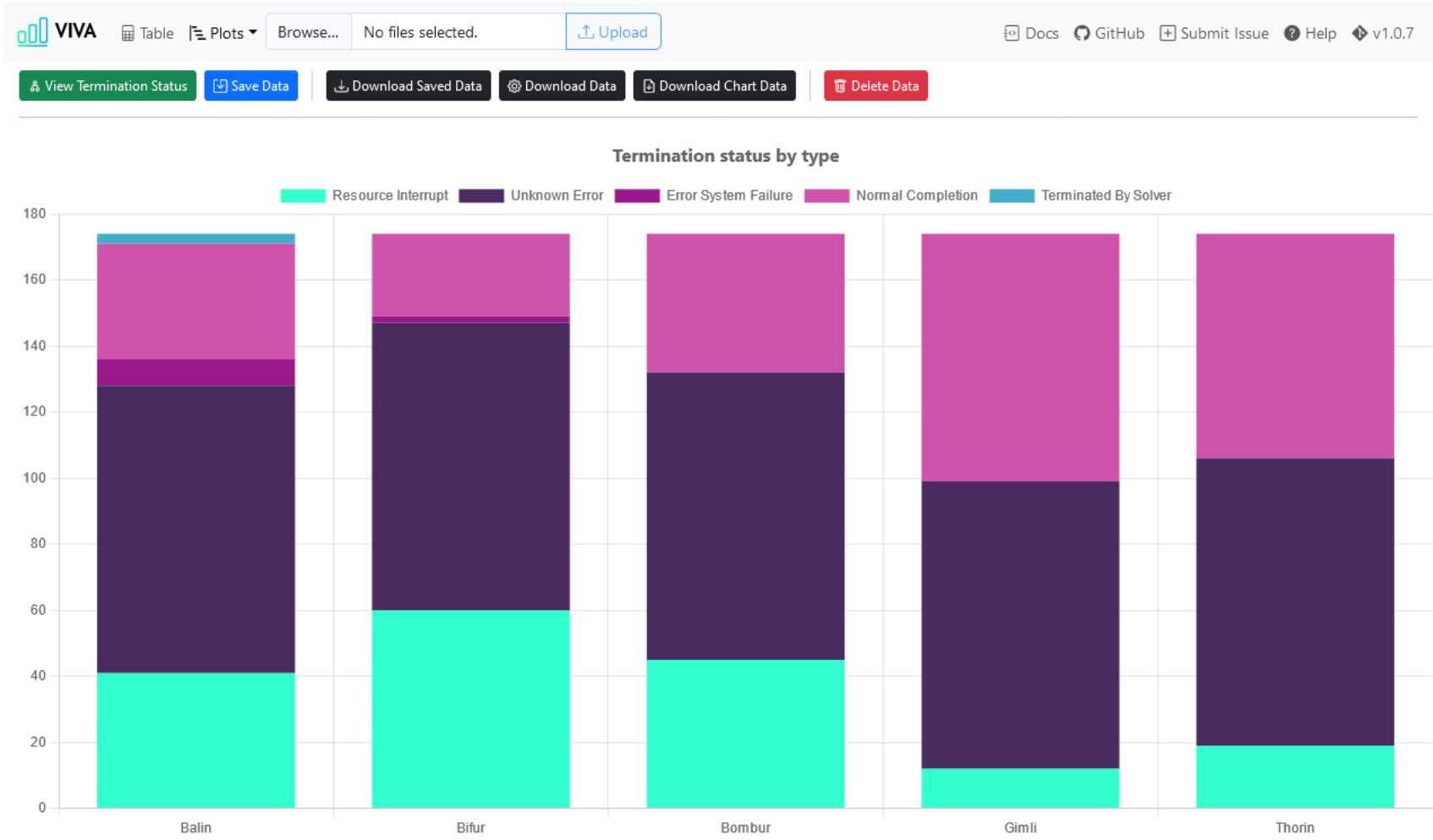


Figure C.6: Termination status per solver plot.