ANALYZED: A Linux-based Disk Analyzer

Fall 22'

CSCE 345 / 3401: Operating Systems

Abdelaaty Rehab
Ibrahim Gohar
Amr Sallam

The American University in Cairo

ANALYZED: A Linux-based Disk Analyzer

Fall 22’

by

Abdelaaty Rehab
Ibrahim Gohar
Amr Sallam

Student Name	Student Number
Rehab	900204245
Gohar	900203321
Sallam	900196011

Instructor:	Dr. Amr ElKadi
Project Deadline:	3th December, 2022
School:	School of Sciences and Engineering, AUC
Department:	Computer Science and Engineering Department

Preface

Abdelaaty Rehab
Ibrahim Gohar
Amr Sallam
AUC, January 2023

This report represents a detailed manual of the presented application that is considered to be an open-source Linux disk analyzer. Disk analysers have become of great importance since computers are developing rapidly. Disk Space Analyzer's purpose is to help a user to analyze disk usage, find large files and remove unneeded "space eaters" – old unused files and folders that occupy significant space on the disk. Moreover, the lack of disk analysers for Linux-based systems make this application of great importance since Linux packages lack rigorous yet GUI-provided application. This application represents a humble trial to offer such opportunity for Linux-based systems users.

Summary

In this manual, you can find a detailed explanation of a trial to implement a $RUST$ Linux disk analyzer. The manual will take you in a trip throughout the application starting from how to use the program, passing by a flavour of how the front-end and the back-end of the application of the application were implemented, then the used frameworks and the design trade-offs and ending with the application of some Agile terminologies within this application.

Contents

1

Introduction

$RUST$ has become of the fastest growing programming languages these days. This is due to its rigorous implementation as well as its powerful analyser that tolerates no errors. That is why $RUST$ was chosen to be the base programming language for building this application because not only is $RUST$ secure but also it is very efficient in performance where it is considered to be one of the fastest programming languages available nowadays. As a result of this choice, a $RUST$ back-end framework was chosen so that it becomes integratable with the desired functions. $Tauri$, which is one of the most powerful $RUST$ back-end frameworks, has been chosen to perform such purpose. As for the front-end framework, there has been multiple front-end js framework but $VanillaJS$ served as the easiest and the most convenient one for such simple project. The next chapters of the manual will guide you through a journey across the program.

2

How to use?

2.1. Dependencies and required libraries

Make sure to install the following dependencies before running the application.

- Tauri
- jQuery
- rustup
- cargo

2.2. How to install and run?

```
git clone https://github.com/abdelmaksou/analyzed
cd analyzed
cargo tauri dev
```

Listing 2.1: Cloning and Buiding the application

2.3. How to use?

The application starts a screen in which the user can choose to write the path, choose it, or run the program at the home directory. [Shown in Figures 2.1 & 2.2] After that, the application displays both nested pie chart and tree view. All the program functionalities is further explained in Chapter **7**.

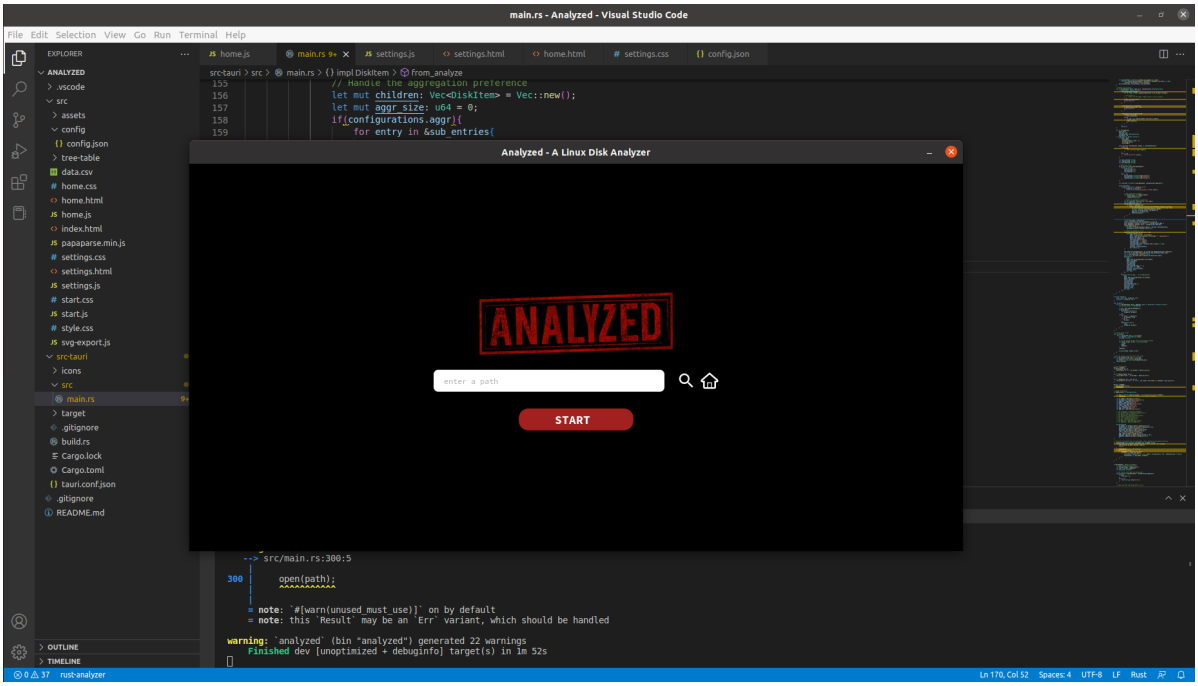

Figure 2.1: A start screen so the user can either enter the path, choose it with a file dialog screen, or run the app at the home screen.

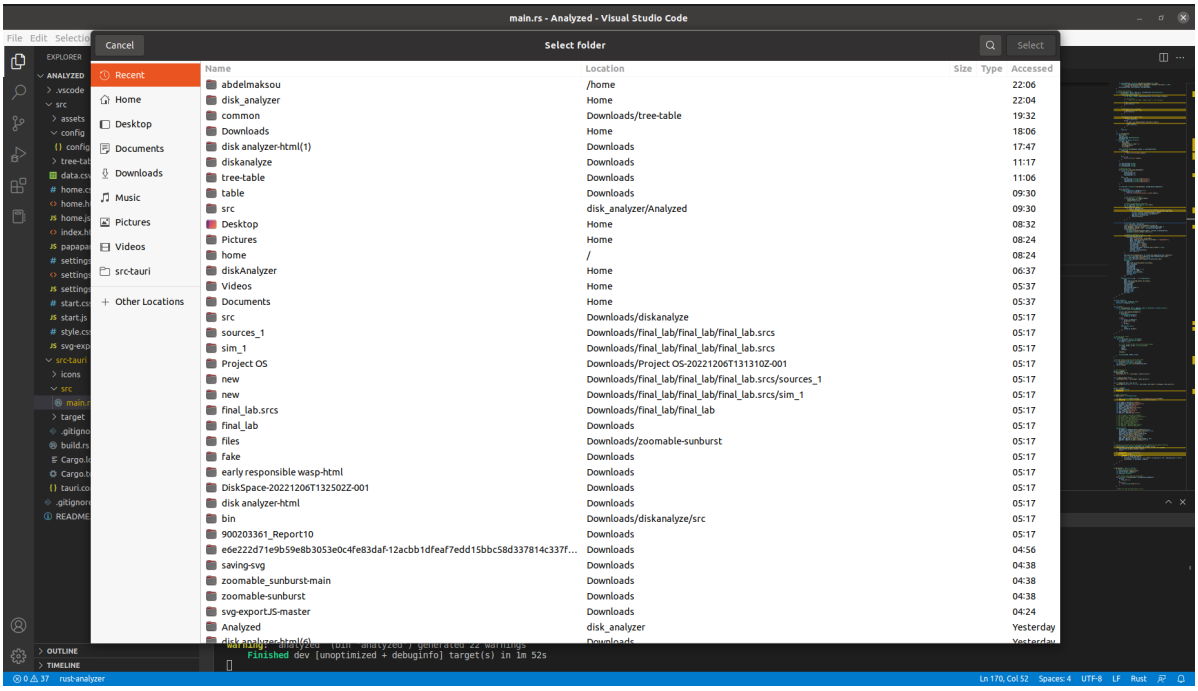

Figure 2.2: A Screenshot showing the prompt asking the user to choose a directory to start the scanning process from it.

3

Back-end Implementation

In this chapter, we will discuss the implementation done in the back-end and how the data that are displayed in the program front-end is calculated and formatted for proper display.

3.1. Data Structures

- The main data structure is a struct ”DiskItem” which represents a node in the hierarchical tree of the file-system.
- Struct ”DiskItem” has the corresponding metadata of every scanned folder/file:
  - `Name:` A string representing the name of the file/folder.
  - `Disk-size:` A $u64$ representing the size of the corresponding file/folder in bytes.
  - `depth:`A $u64$ representing the level in the hierarchy tree.
  - `Last-accessed:` A string representing the last access time/date to the file/folder.
  - `Last-modified:` A string representing the last modification time/date to the file/folder.
  - `Creation-time:` A string representing the creation time/date of the file/folder.
  - `num-contained-items:` An Option<u64> attribute that represents the number of items contained in the corresponding folder.
  - `Fraction:` An Option $< f64 >$ attribute that represents the size percentage of the corresponding file/folder out of the parent folder.
  - `Children:` An $Option < Vec < DiskItem >>$ attribute that holds the structs of all children of the corresponding folder.
  - `Err:` An $Option < String >$ attribute that holds an error message, in case of failures.
- Another struct ”Configurations” is used to keep track of the currently defined preferences for the user. This struct has the following attributes:
  - `no_hidden` A flag to indicate whether to include hidden files or not.
  - `apparent` A flag to indicate whether to show the apparent size or the actual size of each item.
  - `no_empty_dir` A flag to indicate whether to include empty folders or not.
  - `depth` A flag to indicate whether a specific depth is to be considered or not.
  - `max_depth` A $u64$ to hold the desired depth.
  - `regex` A flag to indicate whether there will be some regex patterns to exclude.
  - `re` A string holding the actual regex patterns.
  - `aggr` A flag to indicate whether to aggregate files in a pseudo-directory or not.
  - `aggr_max` A $u64$ holding the maximum file size to be aggregated.
- An enum ”FileInfo” is used to hold the information of whether the specified path is a file or a folder, because they will be dealt with differently.
- Struct ”ScriptOut” is used to hold the results of running a script, with the following attributes:
  - `Code:` An $i32$ attribute to hold the return status.
  - `Output:` A string attribute to hold the output of running the script in the shell.

  - `Error`: A string attribute to hold a message of any error happens.

- All $Option <_>$ attributes are optional, meaning that some are for folders and some are for files, like children, which are for folders only. This feature is used to be able to use the same struct for both files and folders.

3.2. Scanning Algorithm

The main algorithm is the one used for scanning the file-system starting from the specified path, then passing back the head of the tree representing the scanned file-system nodes. The algorithm goes through the following steps:

- Specify the root path, from which to start the file-system scanning.
- Check whether the root path corresponds to a file or a folder. If it is for a file, a message stating that scanning starting from a file is not possible.
- The path is then passed to the scanning function to start the file-system scanning.
- For both files and folder, the following attributes are extracted from $path.metadata()$:
  - `name`
  - `last_accessed` (being “Unknown” if an error happens)
  - `last_modified` (being “Unknown” if an error happens)
  - `creation_time` (being “Unknown” if an error happens)
- In case the current path is the path of a file, the following is invoked
  - `num_contained_item = NULL`
  - `children-item = NULL`
  - `Err = NULL`
- In case the current path is the path of a directory, the following is invoked
  - The first level sub files/folders are read using `fs::read_dir()`, then attached to a vector.
  - For every entry in the previous vector, the scanning function is called recursively. This is the same as doing DFS.
  - Because we are doing DFS, the tree will be filled bottom-up, meaning that we can calculate the total size of each folder recursively by summing the sizes of the first level sub files/directories.
  - `num_contained_items` is calculated based on the decision of using DFS, as the previous point. It is calculated by summing `num_contained_items` of the sub-directories and the number of sub-directories themselves.
  - Finally, the vector of children, specified previously, is assigned to the struct attribute `children`.
- For each recursive call to the scanning function, the ”Configuration” struct is used to apply the user-specified preferences, like excluding specific files/folder and stop at certain depth.
- For the file aggregation feature, the algorithms is as follows:
  - Check if the ”aggr” flag is set. If so, continue.
  - Get all first level children of the current folder.
  - Loop over them, then if the current entry is a file, check if its size is smaller than or equal the max aggregate file size. If so, push it to ”children” vector and add to the aggregated total size so far.
  - Check that the children vector is not empty. If so, make a new ”DiskItem” called ”<aggregated>”, assign ”children” vector to its ”children” attribute, then push this new ”DiskItem” to the ”children” vector of the current folder.
- After the scanning function returns, the tree is converted to a `JSON` object for communication with the front-end.

3.3. Used Crated

The following list includes the most prominent crates that were imported and used in the application's back-end.

- `std::path::PathBuf` Used to construct all paths to be scanned.
- `std::error::Error` Used to catch and identify errors.
- `serde::Serialize` Used to be able to construct the `JSON` object from the complex tree.
- `std::fs` Used to read the paths and metadata of the first level folders/files of the specified path.
- `chrono::DataTime` Used to represent the access, modification, and creation date/time.
- `run_script::ScriptOptions` Used to run CLI scripts, collect the return value, output, and error, if there exists.
- `opener::open` Used to open the specified path using the default program on the system.
- `std::fs::File` Used to read and write to a file.
- `regex::Regex` Used to generate regex patterns to be used in the exclusion feature.
- `serde_json::Value` Used to dynamically parse JSON string to the corresponding object. This is used in the configuration part.

4

Front-end Implementation

This chapter discusses the main design approaches followed during the implementation of the application's graphical user interface.

4.1. Vanilla JS

Vanilla JS represents the main front-end framework upon which the application resides. Despite being less efficient than other available JavaScript frameworks like $Vue.js$ and yew, Vanilla represents a simple alternative for each of them since the syntax is mainly dependent on basic JavaScript functions which are well-documented on $FirefoxMDN$. The source directory of the application's front-end includes the following sub-directories:

4.1.1. HTML

This sub-directory includes the mark-up files of the application's different screens written down using HTML.

4.1.2. CSS

This sub-directory includes different styling sheets for the HTML files linked using $< style >$ tag.

4.1.3. JS

This sub-directory includes different $.js$ files that are responsible for manipulating the DOM as well as invoking functions from the application's back-end.

4.2. Widgets

Looking at the main screen of the application, we can see that we have the following main widgets:

- **Tree View:** A $jQuery$ plug-in that is ported to the application and customize to display the information received from the $RUST$ back-end. The tree view port occupies the left hand side of the application's main UI.
- **Nested Ring Chart:** A $d3.js$ block that is ported to the application's main UI to represent the statistics of the current directory. The chart offers responsiveness as well as interactivity where clicking a chart segment takes the user to the corresponding directory represented by the invoked segment of the chart representation.
- **Top Navigation Menu Bar:** A navigation menu bar that contains the functionalities offered by the application sorted based on their category.

5

Integration

Tauri is used to integrate the front-end with the RUST back-end. Tauri provides a simple yet powerful command system for calling Rust functions from your web app. Commands can accept arguments and return values. They can also return errors and be async. The command starts with

```
#[tauri::command]
```

and an invoke handler with the name of the function is added to the main rust file.

```
.invoke_handler(tauri::generate_handler![command])
```

In the same time, the JS file contanins an invoke request

```
import { invoke } from '@tauri-apps/api/tauri'
const invoke = window.__TAURI__.invoke

// Invoke the command
invoke('command')
```

and the JSON text is parsed by this command.

```
JSON.parse('JSON text')
```

The RUST back-end serializes the file system content into JSON object. Then it gets edited into two different structures that used to run both the pie chart and the tree view.
This JS function is used to edit the JSON object so it can run in the pie chart:

```
function edit_for_pie(obj) {
  obj.value = obj.disk_size;
  delete obj.disk_size;
  if(obj.children != null) {
    if (obj.children.length === 0) {
      obj.children = null;
    } else {
      for(let i = 0 ; i < obj.children.length ; i++){
        edit_for_pie(obj.children[i])
      }
    }
  }
}
```

This one for editing it for the treetable view:

```
function edit_for_tree(obj) {
  obj.value = obj.name;
  obj.data = obj.children;
  delete obj.name;
  delete obj.children;
  if(obj.data != null) {
    if (obj.data.length === 0) {
      obj.data = null;
    } else {
      for(let i = 0 ; i < obj.data.length ; i++){
```

```
          edit_for_tree(obj.data[i])
        }
      }
    }
  }
}
```

6

Used Programming Languages and Frameworks

In this chapter, we discuss the different front-end and back-end frameworks that are used within the application and why they were chosen over other available frameworks.

6.1. Frameworks and UI Kits

There were a plenty of front-end frameworks that were available and ready to use. The following subsections represent a survey of which frameworks were tried and which were actually chosen.

6.1.1. Vue.js

Vue.js represents one of the most performant and versatile framework for building web user interfaces. It is approachable where it easily builds on the top of standard HTML, CSS and JavaScript with intuitive API. It is efficient where it overtakes most of the available frameworks in terms of speed and performance. This is because it is reactive having a compiler-optimized rendering system that rarely requires manual optimization. It also groups the simplicity of a JS library with the power of a JS framework. The greatest advantage of Vue.js is that it is one of the best documented front-end JS frameworks. The drawback we faced with Vue.js was related to signal and slot linking. We had a bunch of errors while invoking the back-end functions the thing which had us think about an easier alternative.

6.1.2. Yew

Yew represents one of the most abstract front-end applications. It is component-based as it features a component-based framework which makes it easy to create interactive UIs. It has two major features over any other front-end frameworks which are HTML macros and Server Side Rendering. Yew is the only framework that offers interactive HTML declaration using pure $RUST$ syntax. What makes Yew the fastest available framework is the conditional rendering. Yew's server side rendering offers conditional rendering states which re-render front-end elements based on certain states. This offers a tremendous performance advantage over other frameworks that depend on invocations and signals which suffer from performance overhead. The biggest disadvantage of this framework is its hardness and lack of documentation the things which makes using it without having enough experience nearly impossible.

6.1.3. React.js

React is the most famous JavaScript framework nowadays that is provided with a rigorous documentation as well as a tremendous community the thing which makes solving errors during application development easier than any other framework. The drawback that React suffers from is the performance overhead with respect to other frameworks. It nearly offers no advantage over using pure JS especially when developing small applications that are more dependent on their back-end more than their front-end. That's what has made using React somehow useless in this application since it is mainly

dependent on $RUST$ in pre-computing the disk analysis results and using only the front-end in adding some user-friendly graphical interface to those functionalities.

6.1.4. Tauri

During our search for a RUST back-end framework, we found Tauri. Getting more information about it had us know that Tauri is a popular framework for creating incredibly small, lightning-quick binaries for all popular desktop operating systems. For the creation of their user interface, developers can incorporate any front-end framework that compiles to HTML, JavaScript, and CSS. The application's back-end is a rust-sourced binary with an API that the front-end can use to communicate. Tauri offers a bunch of advantages like App storage, App tray, Core plugin system, Desktop bundler, GitHub action, Native notifications, Scoped file-system, Self-updater and Sidecar. Thus, we decided to use Tauri framework incorporated with Vanilla JS for out application implementation.

6.1.5. Vanilla JS

After reviewing a bunch of the available frameworks, we agreed that using Vanilla JS is the best available option for such small projects since most of the functionalities implemented are mainly depended on the application's back-end the thing which makes powering the application with some sophisticated technology a hustle that is not worth going through.

6.2. Languages

6.2.1. HTML

The `Hyper Text Markup Language` represents the god father of building websites. It is one of the most basic yet most powerful markup languages used not any in web development but in cross-platform development as well.

6.2.2. CSS

The `Cascaded Style Sheets` represent the most fundamental method to statically and dynamically style HTML files for visual appeal.

6.2.3. JS

The greatest JavaScript which is a text-based programming language that allows the creation interactive elements on the DOM. JS is also used for invoking functions that are used for message and data passing between the front-end and the back-end.

6.2.4. Rust

Rust is the main programming language used in the whole application. It is the main utility upon which the application depends in processing data in the back-end and communicating with the front-end passing this data and receiving invocations through Tauri. Rust is chosen as it is solid and fast providing both security and performance advantages.

7

Application Functionalities

The aim of this chapter is to provide a user manual for the functionalities available in the application and how to use them.

7.1. Nested Ring Chart

Directories and files in the specified path are displayed in a nested ring chart view which makes it easier to see the relative sizes and contents of the directories. The nested ring chart have four levels in maximum that could be displayed. Hovering over a section of the chart displays the name of the item and its size. Once one is clicked in the chart, the tree view is updated accordingly expanding all the directories down to the selected one. [Shown in Figure 7.1 and Figure 7.2]

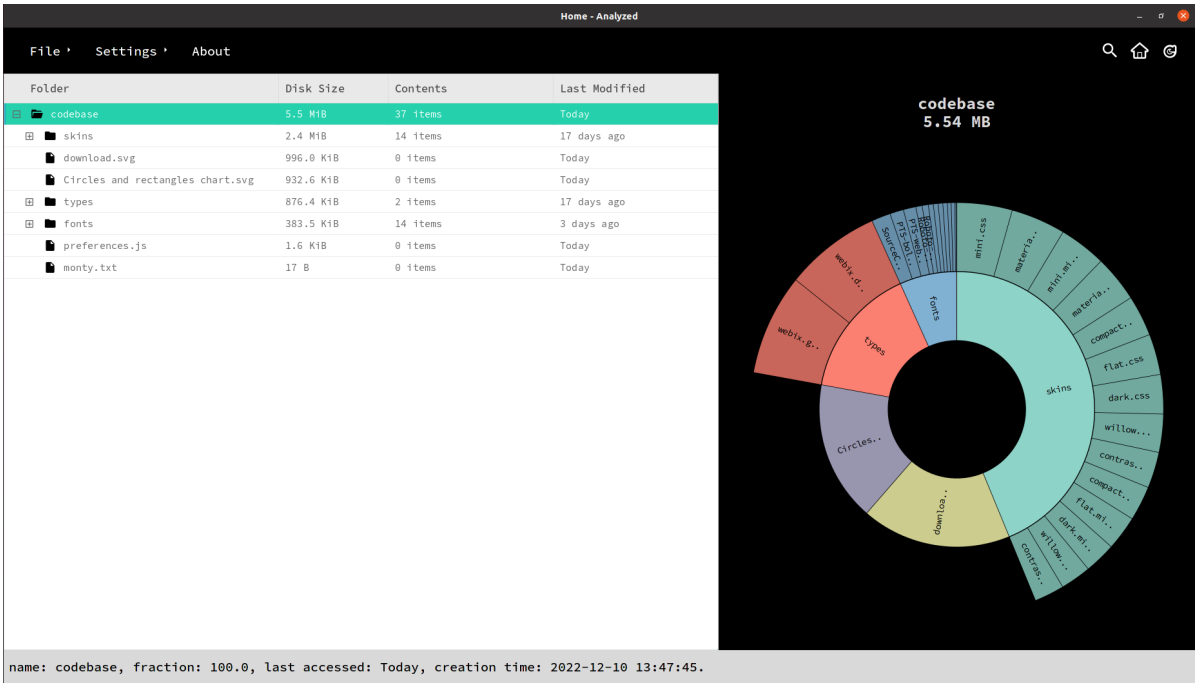

Figure 7.1: A Screenshot from the Program selecting ”codebase” directory as the root, and hovering over any directory displaying its metadata with the relative and the rest of the metadata at the bottom.

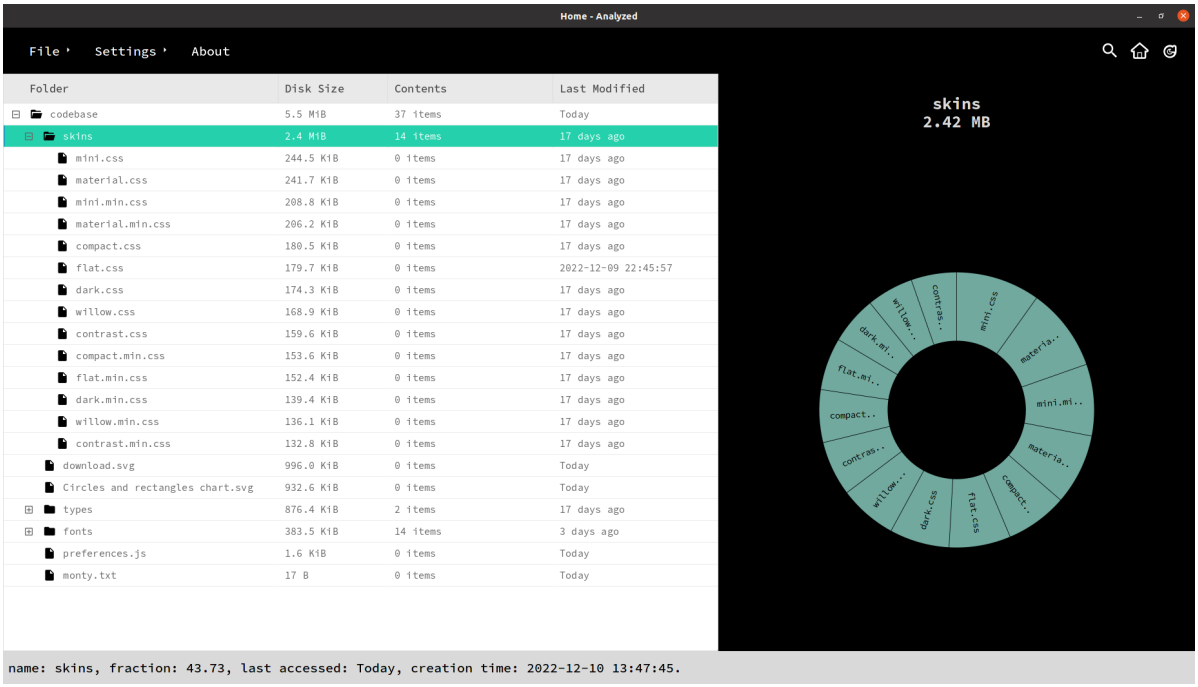

Figure 7.2: A Screenshot from the Program after clicking on ”skins” directory, making it the root of the ring chart, and updating the tree view.

7.2. File System Tree

Directories and files in the specified path are, in the same time, displayed in a tree-like view which makes it easier to access the files and read their metadata as a whole. Once one is selected in the tree, there is two things that follow. First, the metadata is displayed in the bottom right part in the screen. Second, the Pie Chart is updated accordingly selecting this directory as a root. [Shown in Figure 7.1 and Figure 7.2] Moreover, there is a Context Menu that is displayed by clicking right; user can open, delete, or view the full metadata from it. [Shown in Figure 7.3]

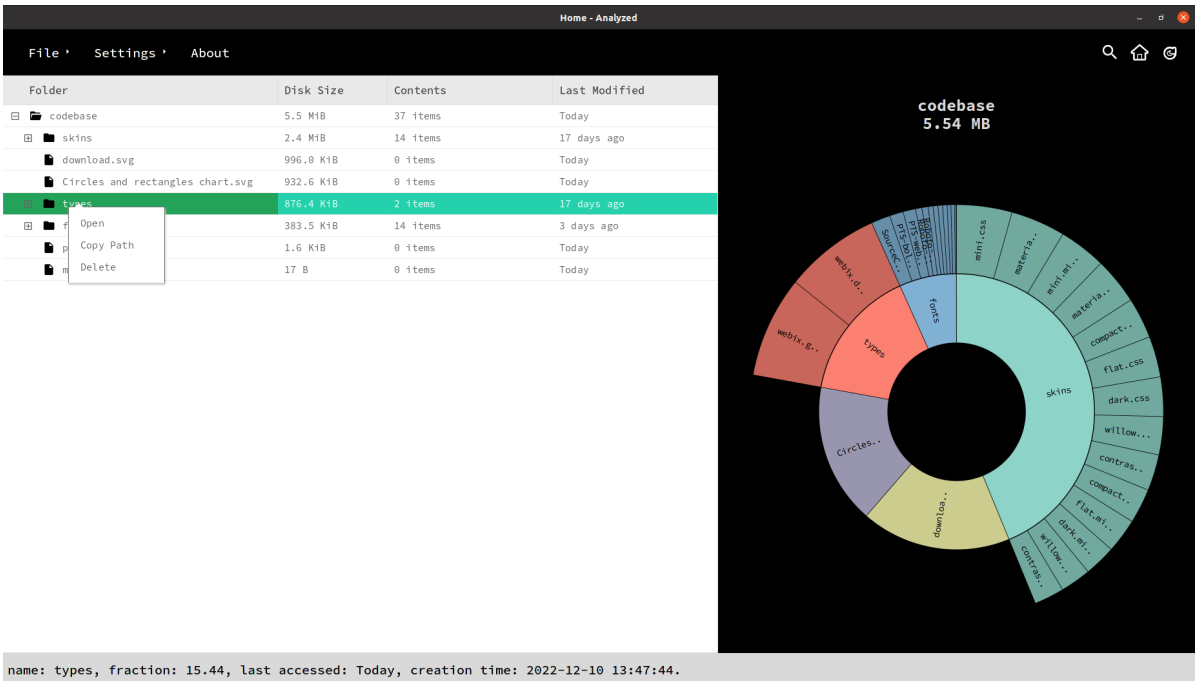

Figure 7.3: A Screenshot from the Program clicking right on ”types” directory in the tree view showing the context menu.

7.3. Analyzing Preferences

After choosing ”Preferences” from the ”Settings”, a new window opens, displaying the all the preferences that can be configured so the user experience is customized as much as we can. [Shown in Figure 7.4]

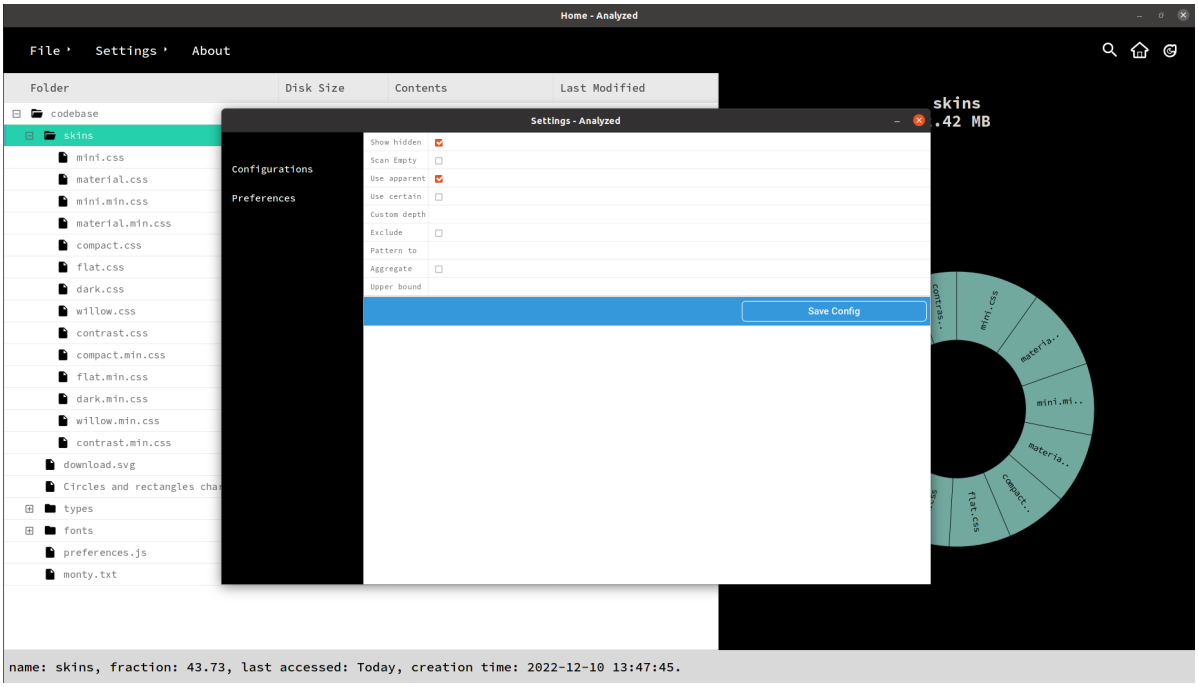

Figure 7.4: A Screenshot from ”Preferences”, displaying the full options that can be configured.

7.4. Grouping files in a pseudo-directory

When scanning the file-system hierarchy, sub-files being accompanied by sub-folders, in the same parent folder, are grouped together in a pseudo-directory for better visualization. [Shown in Figure 7.5]

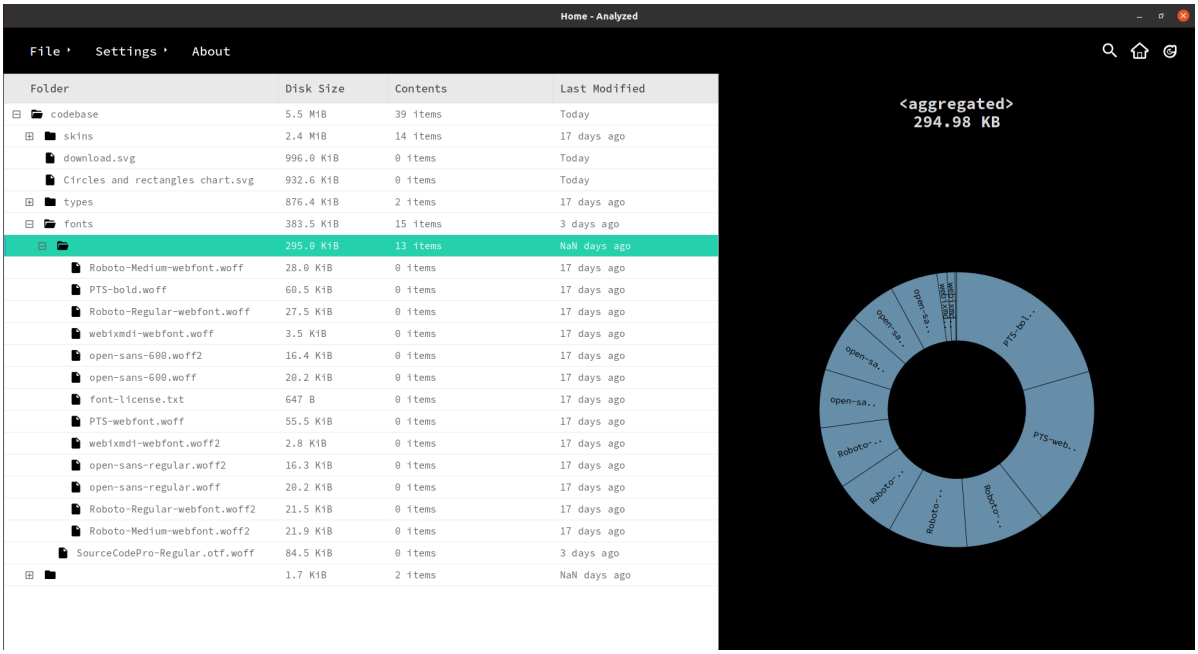

Figure 7.5: A Screenshot from ”fonts” directory, now aggregated is on in the ”preferences” sheet, showing a pseudo-folder called ”<Aggregated>”.

7.5. User-defined controls

User-defined controls are provided in the tool, with default clean-up options for not experienced user. Those options can be accessed from the top navigation bar in ”Settings” tab. [Shown in Figures 7.6 & 7.7] User-defined means that the user can specify what exactly the option does, by a CLI instruction. Also, the user can modify the default options and expand their functionality as far as they need. Those functionalities can be added in the ”Add New Configuration” option, found in ”Settings” tab. [Shown in Figures 7.6 & 7.7]

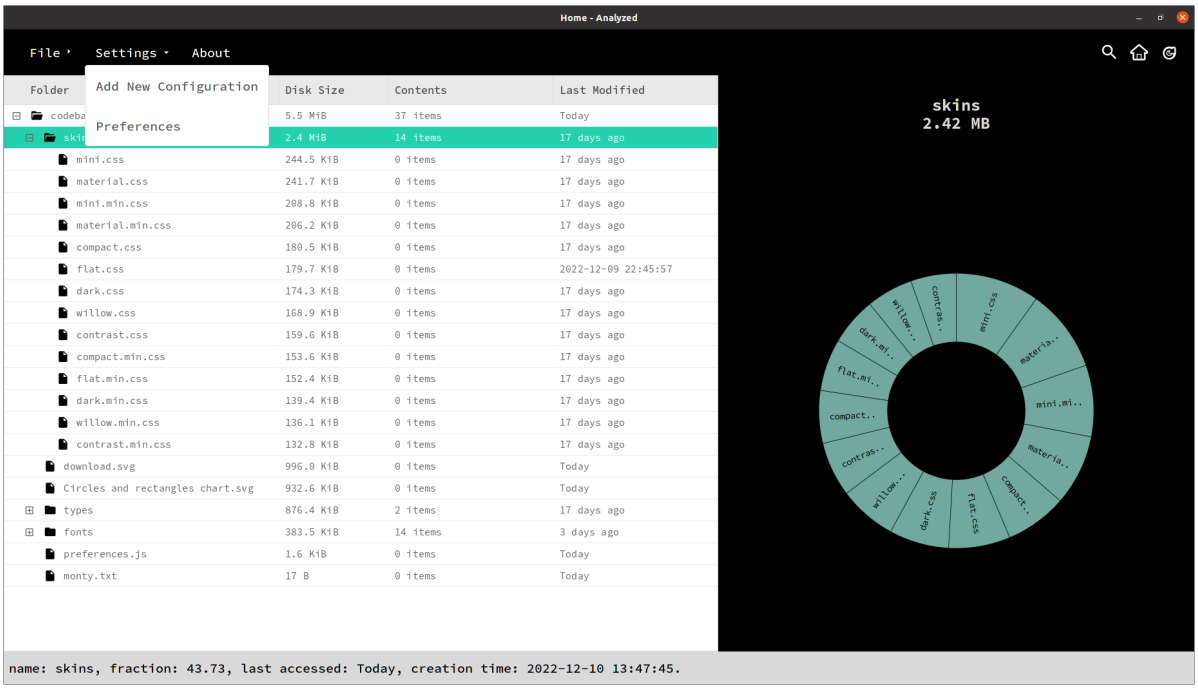

Figure 7.6: A Screenshot showing the contents of ”Settings” tab in the top navigation bar.

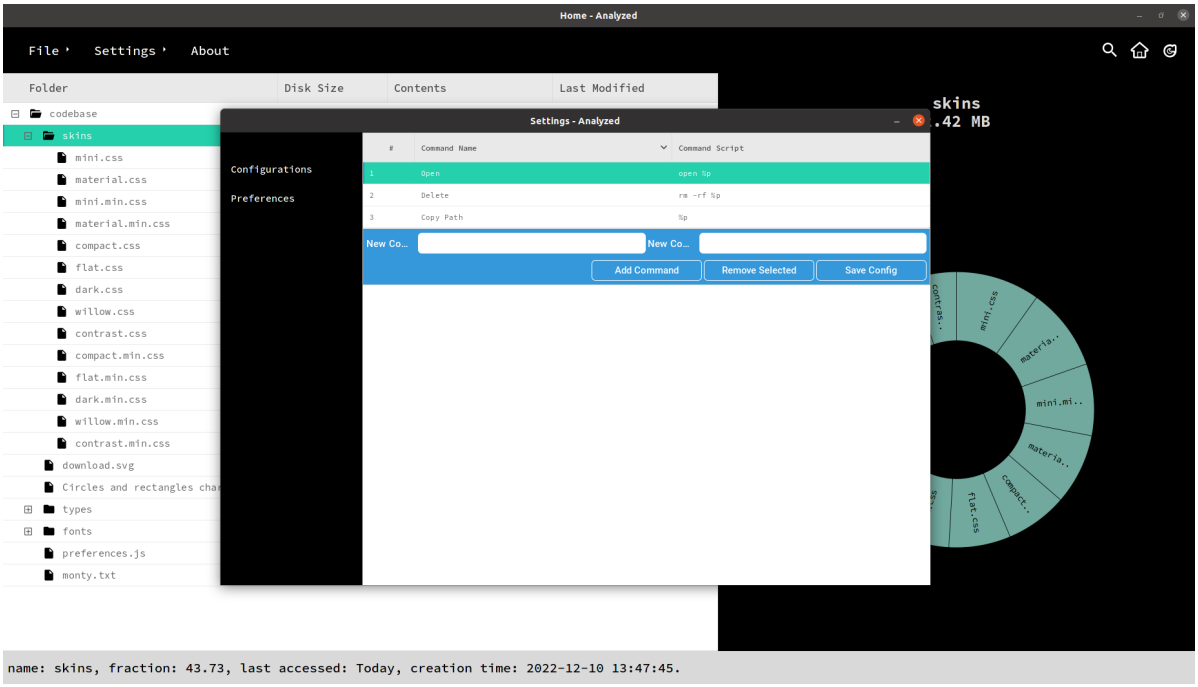

Figure 7.7: A Screenshot showing the contents of ”Configurations” tab in the top navigation bar.

7.6. Files and directory exclusion

The tool provides the utility of excluding specific files or directories from the file-system scanning using pattern techniques, such as regex.

7.7. Retrieve the largest N files

The user can retrieve the largest N files among the whole scanned file-system portion.

7.8. Scan new directory

The user can choose to scan a new directory in another location. This can be done by choosing ”Open New Directory” option in the ”File” tab that is in the top navigation bar. [Shown in Figure 7.8] After that the user will be prompted with the same screen that got displayed in the beginning, so they get to choose a new directory.

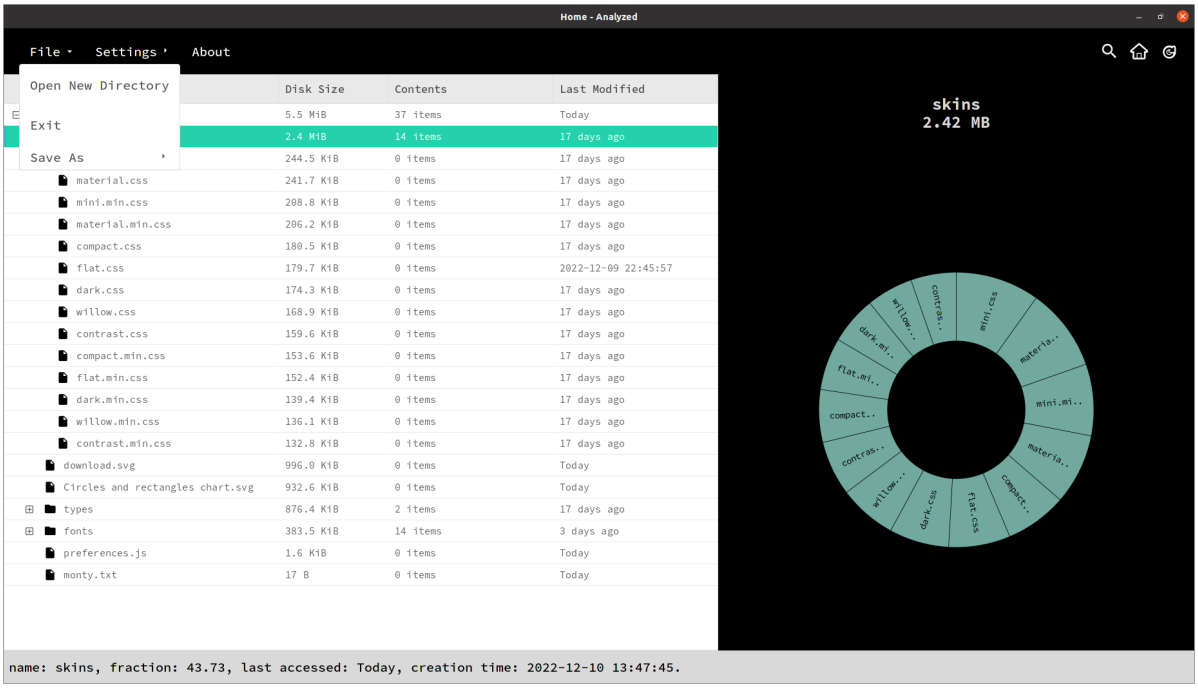

Figure 7.8: A Screenshot showing the contents of ”File” tab in the top navigation bar.

7.9. Save Charts

The user can choose to save either the tree map or the pie chart from ”Files” tab in specified location he chooses. [Shown in Figure 7.9]

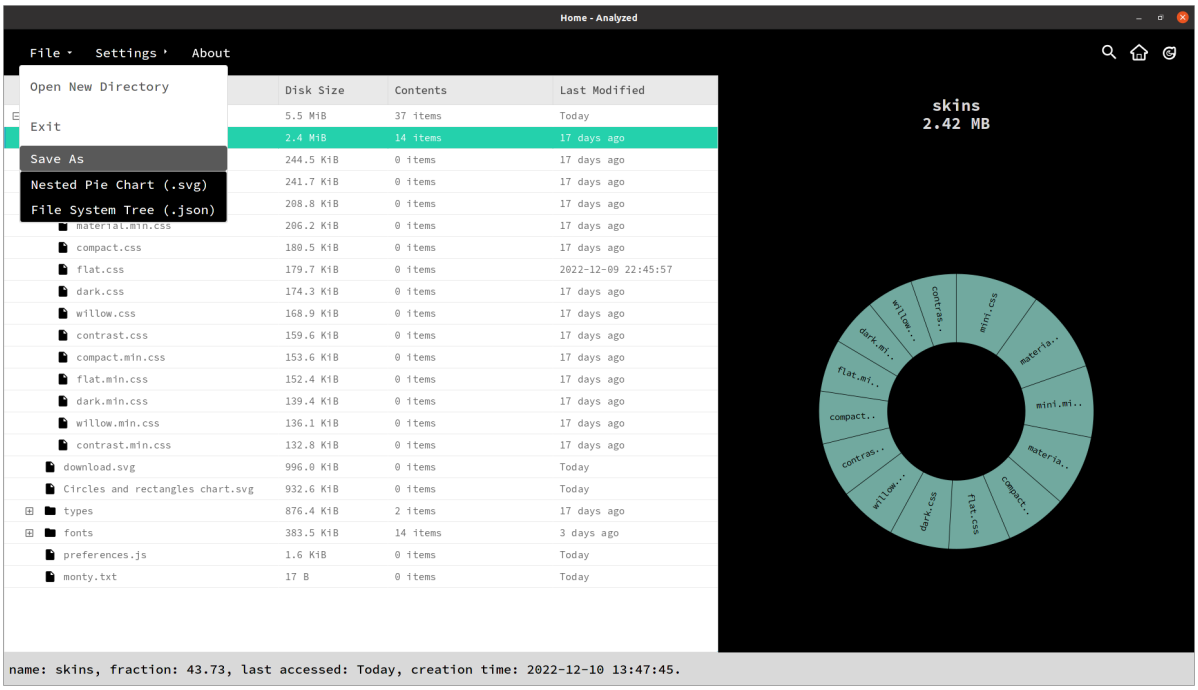

Figure 7.9: A Screenshot showing the contents of ”Save As” option in ”File” tab in the top navigation bar.

8

Application Development Cycle

Finally, we present how the Agile concepts were applied to this application by showing and highlighting the main cycles of the application development lifetime.

8.1. Planning and Design

We started by brainstorming to specify the design requirements, divide work among us, and agree on some rules to follow during the development cycle. An initial visualization of the desk analyzer was discussed and the main features to include were written down. An intensive research about the available front-end frameworks, rust crates, and any external APIs was done to identify the list of available alternatives.

- Using tree as the main data structure for the back-end.
- Using `JSON` in communication between back-end and front-end.
- Starting with already existent open source tools.

8.2. Implementation

Every member started implementing his part using the agreed on alternatives. Continuous integration was emphasized for better and fast results, such as using GitHub. When issues were encountered, or a used alternative found to be not suitable for our design requirements, a meeting was conducted to discuss new alternatives. The cycle continues till the assigned features for every member is completed.

8.3. Testing

Testing goes in parallel with implementation, but in a litter manner. However, after the Implementation has completed, intensive testing was conducted by scanning varying size filesystems, then observing for any issue to be fixed.

9

Testing and Benchmarks

9.1. Testing

The application analysis output were put in a `JSON` file and tested against the output of `dirstat_rs`. The results were shown to be approximately the same which validates the output of the implemented application to a good extent.

9.2. Benchmarks

One of the most important features of a disk analyser is to make sure that it will use minimal memory and mainly depend on read/write operations to be performed on the target hard disk. That is what we wanted to test in our application and that is what had us use minimal front-end framework along with a fast solid back-end framework like Tauri Rust.

9.2.1. Testing Mechanisms

The benchmark tests were designed upon stress-testing the CPU and the memory whilst running the application for a quite significant time. The results were collected and compared to those obtained from stress-testing `baobab` which is the GNOME disk analyser. The implemented program was found to be at least 150% faster. This is due to the fast back-end framework that depends on using Tauri along with RUST which provides a big performance advantage over `baobab`'s C++ implementation. The following figures represent the results obtained by stress-testing against the two applications for solid 5 minutes giving both a chunk of 5 terabytes of data to scan and analyse.

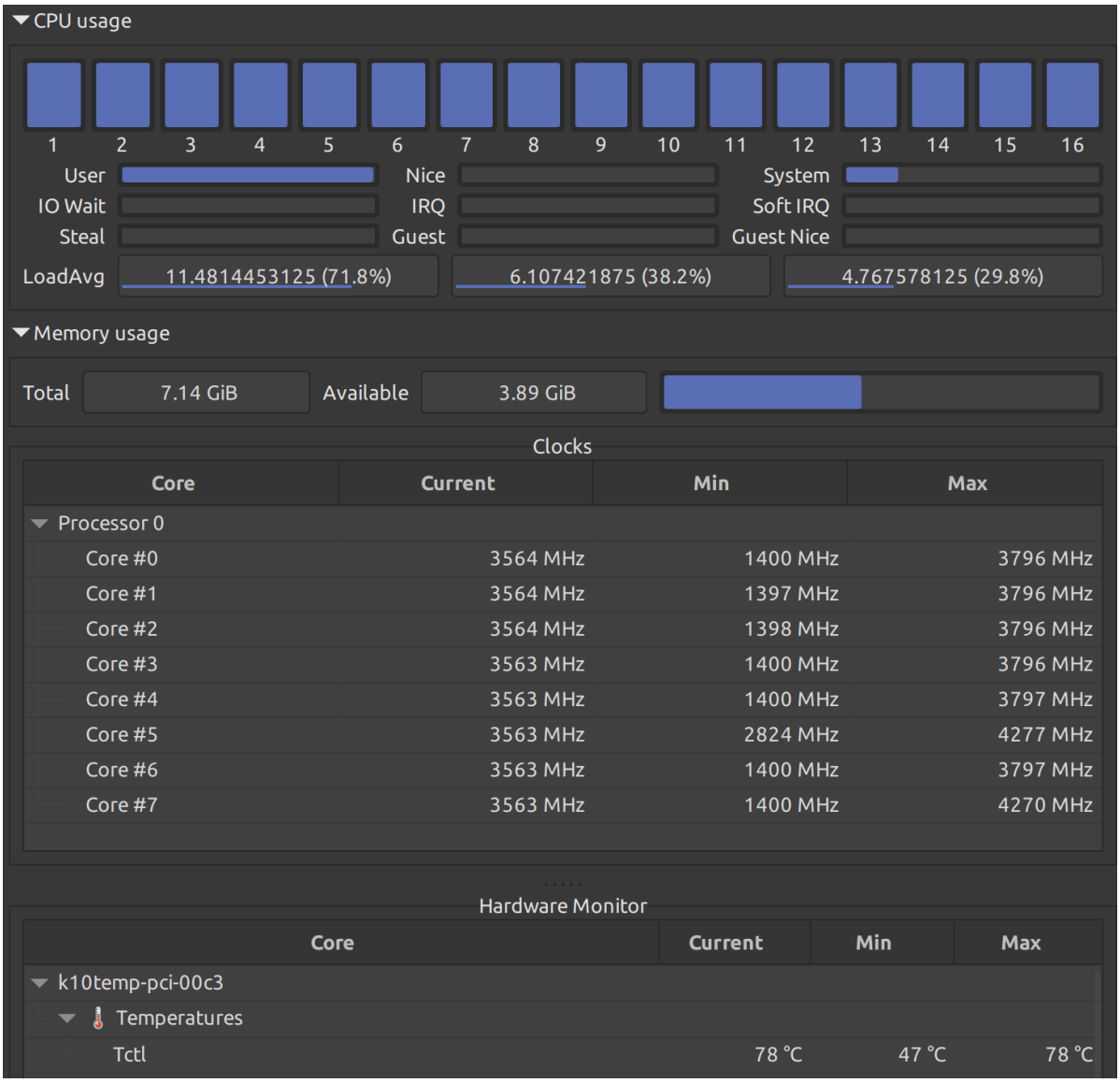

Figure 9.1: Benchmark results testing against `baobab` for 5 minutes.

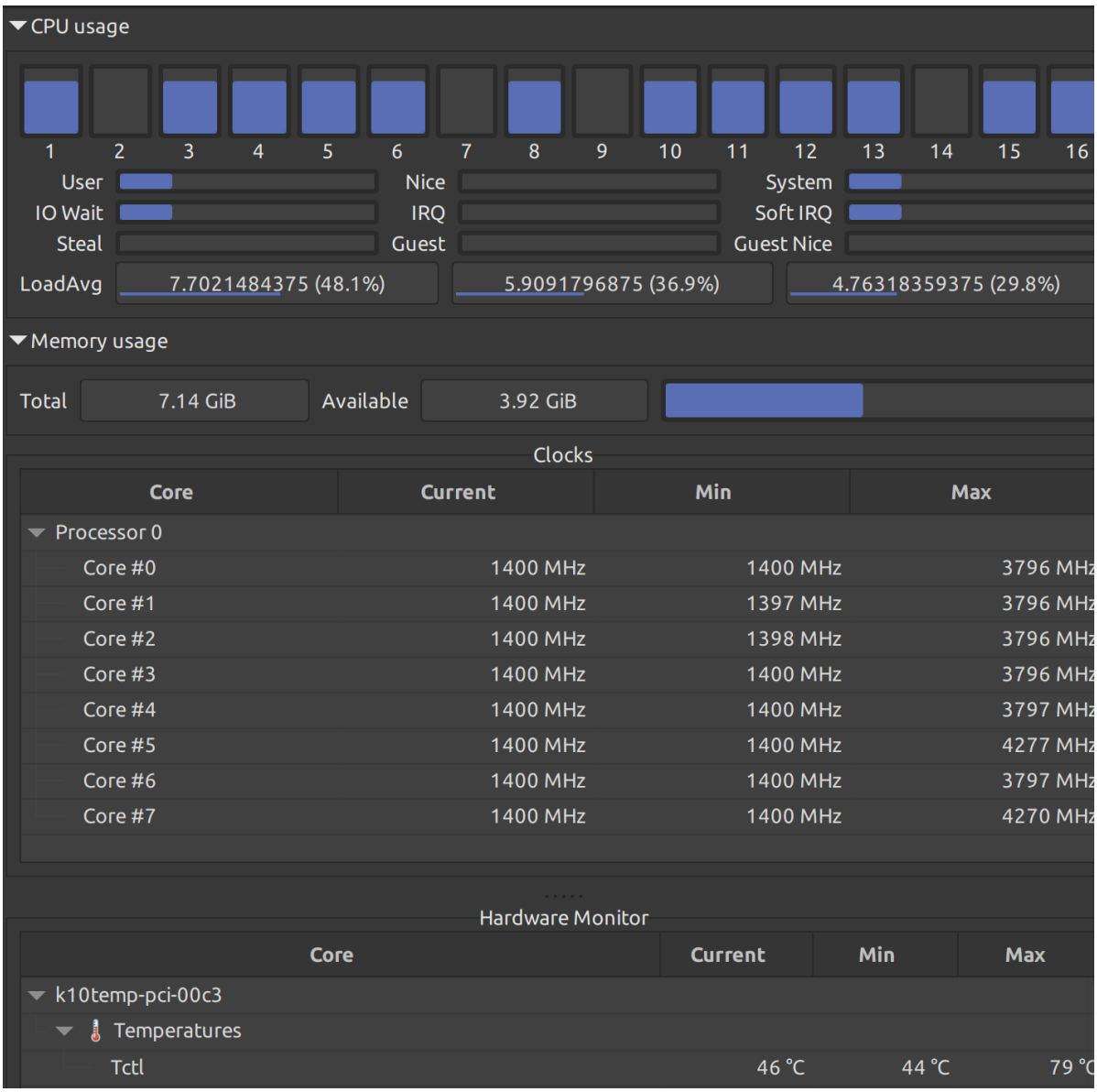

Figure 9.2: Benchmark results testing against `ANALIZED` for 5 minutes.

10

Conclusion

This manual represented a full review of the implemented application considering all its technical and design aspects as well. The user is expected to read the manual before using the program in order to understand the underlying operation, the design approach as well as the complete set of functionalities offered by the application. The application produces results that have been tested to be correct by testing it against already implemented for Linux-based systems.

A

Source Code

The application is mainly intended to be an open-source application. The source codes are on the application's main repository available on GitHub.

B

Task Division

Table B.1: Distribution of the workload

Task	Student Name(s)
Tree View & Configurations	Rehab
Nested Ring Chart & Preferences	Gohar
Directory Analyzing functions	Sallam
Tauri Integration	Rehab & Gohar & Sallam