Master's thesis
in the program Molecular Medicine
"Master of Science"

of the Faculty of Medicine
at the University of Regensburg

DEVELOPING A WEB APPLICATION FOR THE DYNAMIC EXPLORATION OF TUMOR HISTORIES

submitted by
Antonia Kaspar
from
Landshut

in the year of
2025

Dean:	Prof. Dr. Christian Wolff
First Advisor:	Prof. Dr. Rainer Spang
Second Advisor:	Prof. Dr. Till Rudack
Date of oral examination:	July 29th, 2025

Contents

Glossary

Angiogenesis the formation of new blood vessels from pre-existing blood vessels.

App Router routing system based on the `app/` directory.

Backend handles logic and data processing of a website or web application.

Base Rate the natural rate of occurrence for a genetic event.

Biopsy a sample of tissue taken from the body to examine, e.g., for cancer cells.

Cancer a disease in which abnormal cells grow uncontrollably and can spread to other parts of the body.

Client typically a web browser that sends a request to the server.

Client–Server Model a distribution model, e.g., for computer applications, that divides tasks or workloads between servers and clients.

Component a reusable piece of code, represents part of a user interface (e.g., a button).

Context a React feature that allows global sharing of data across components without prop drilling.

Count parameter, indicates the number of patients that share a specific history sequence up to a certain genetic event.

CPMs Cancer Progression Models, mathematical models with different approaches (e.g., deterministic, stochastic) used to reconstruct the sequence of genetic events that drive tumor development.

Cross-Sectional Data data collected from multiple individuals at one specific point in time.

CSV Comma-Separated Values, a data format for storing tabular data.

D3.js a JavaScript library for dynamic, interactive data visualizations.

Deployment the process of making a software application or website available for use on the internet.

Development Environment a collection of software and tools required for building and testing applications.

Development Server server environment, used to preview and test the application during the development process in real time.

DOM Document Object Model, object representation of HTML elements in a tree-like shape.

Driver Mutations mutations that actively contribute to tumor development.

Edge link or path between two genetic events displayed in the History Tree.

Event Handler a function that runs in response to user interaction (e.g., a click).

File-System Based Routing a routing method based on folder and file structure.

Framework a basic structural concept consisting of predefined classes and functions to support and simplify the development of software applications.

Frontend the part of a website or application that is visible to the user.

Fullstack the development process of both frontend and backend.

Genetic Events changes that affect the genetic material, e.g., mutations in the DNA sequence.

Hallmarks of Cancer biological traits that differentiate cancer cells from healthy cells.

History Trees MHN Patient History Trees, display the most likely genetic event history path in a tumor.

Hooks built-in functions in React that simplify state management, for example.

Inference an estimation, e.g., by MHN based on data and statistical modeling.

JSON JavaScript Object Notation, a data format used to store and share structured data.

JSX JavaScript XML, syntax extension for JavaScript, provides a HTML-like syntax.

Library a collection of pre-written functions that can be used to carry out particular tasks during the development process.

Markov Chain mathematical system.

Markup a language (e.g., HTML) that uses special characters (tags) to describe the structure and appearance of a document.

Metastasis the process by which cancer cells spread from the primary tumor to other parts of the body and form new tumors.

Method Chaining a programming pattern where multiple methods are called in a single line.

MHN Mutual Hazard Networks, a probabilistic CPM for estimating the most likely order of genetic events in tumors.

Multiplicative Effects interaction effects between genetic events.

Multistep Tumor Progression Model biological model that describes the development of cancer in four steps.

Next.js a web development framework that simplifies the process of building fast, interactive React applications.

Node.js a JavaScript runtime environment outside the browser.

npm Node Package Manager, manages and installs JavaScript dependencies.

Observation Effects the effect of the presence of a genetic event on the observation event, i.e., clinical detection of the tumor.

oMHN observation MHN, extension to the classic MHN.

Oncogenes genes that cause uncontrolled cell proliferation when activated through e.g., mutations.

Package Manager a tool that automates the process of installing, upgrading, configuring, and removing software packages.

Passenger Mutations mutations that have a neutral effect on tumor development.

Performance the speed at which a website loads and reacts.

Prop "property", data passed as an argument from a parent to its child component.

Prop Drilling the process of manually passing props through multiple layers.

React a JavaScript library for building user interfaces using components.

Rendering the process of converting code into viewable, interactive web content.

Runtime Environment the software infrastructure that supports the execution of programs.

Server a program or computer that responds to client requests.

shadcn/ui a UI component library with reusable React components.

State Management dynamic data storage, provides a mechanism for components to manage and update data.

Static Superset a language that extends another language by adding static features, e.g., type checking.

SVG Scalable Vector Graphics, used to define graphics directly in the browser.

Tailwind CSS a utility-first CSS framework for styling user interfaces.

Theta Matrix hazard matrix, contains interaction effects between genetic events and their base rates.

Toggle graphical control element, a switch that allows the user to activate or deactivate a function or setting.

Tumor Suppressor Genes genes that act as brakes in the cell cycle and lead to uncontrolled cell proliferation when inactivated.

Tumorigenesis the process of tumor development.

TypeScript a static superset of JavaScript, adds type annotation to the language.

UI User Interface, the graphical layout of an application, often used as a synonym for frontend.

URLs Uniform Resource Locator, the address or route of e.g., a website.

Utility-first web design approach, many small, reusable CSS classes are used for quick and efficient design directly in markup (e.g., HTML).

Virtual DOM a concept in which React saves a copy of the user interface in memory and only synchronizes the changes with the real DOM.

Web Application an interactive software accessed through a browser providing a service for the user.

Web Development the process of building, programming, and maintaining websites and web applications.

Website a collection of related web pages providing information.

Zusammenfassung

Die Entstehung von Krebs ist ein vielschichtiger Prozess. Er beruht auf einer Anhäufung verschiedener genetischer Ereignisse (wie Mutationen) in der Zell-DNA. Für die Krebsforschung ist es bislang eine große Herausforderung, die Reihenfolgen dieser genetischen Veränderungen nachzuvollziehen. Sogenannte Krebsprogressionsmodelle unterstützen mithilfe mathematischer Wahrscheinlichkeitsberechnungen dabei, Muster und Abfolgen in der Tumorentwicklung zu erkennen. Der Mutual Hazard Networks Algorithmus (MHN) ist ein besonders geeignetes Tool für die Modellierung der Krebsentwicklung, da er auch gegenseitige Einflüsse, wie beispielsweise hemmende oder fördernde Wirkungen zwischen den genetischen Ereignissen, einbezieht. Die aus den Patientendaten berechneten wahrscheinlichsten Reihenfolgen genetischer Ereignisse werden in Form von MHN Patient History Trees dargestellt. Da diese Verlaufsbäume sehr umfangreich und schwer zu interpretieren sein können, habe ich eine interaktive Anwendung entwickelt, die ihre Visualisierung vereinfacht. Mit diesem Werkzeug können Nutzer und Nutzerinnen die Bäume flexibel anpassen und eigenständig erkunden. Die Anwendung soll nicht nur Bioinformatiker:innen bei der Analyse unterstützen, sondern auch die Wissenschaftskommunikation verbessern, indem sie MHN-Ergebnisse für Forschende in der Onkologie sowie Kliniker und Klinikerinnen zugänglicher macht.

Abstract

The development of cancer is a complex process. It is based on an accumulation of various genetic events (such as mutations) in the DNA of cells. For cancer research, decoding the sequence of these genetic changes has so far been a significant challenge. Cancer Progression Models use mathematical probability calculations to help identify patterns and sequences in tumor development. The Mutual Hazard Networks (MHN) algorithm is a particularly suitable tool for modeling cancer development, as it also takes into account reciprocal influences, such as inhibitory or promoting effects between genetic events. The most probable sequences of genetic events calculated from patient data are presented in the form of MHN History Trees. Since these History Trees can be very extensive and difficult to interpret, I have developed an interactive web application that simplifies their visualization. With this tool, users can customize the trees and explore them independently. The application is not only intended to support bioinformaticians in their analysis, but also to improve science communication by making MHN results more accessible to oncology researchers and clinicians.

1 Introduction

1.1 Understanding the Biology of Cancer

With rising life expectancy and environmental risk factors, cancer has become one of the most pressing global health challenges of the 21st century [1]. As a result, there is an urgent demand for the development of more effective diagnostic tools and therapeutic approaches, which require a deeper understanding of the biological foundations of the disease.

Examining how normal cells become malignant is key to understanding why cancer is such a vigorous disease. At its core, cancer arises through *tumorigenesis*, which describes the complex process of tumor development by which genetic changes in the DNA of cells lead to uncontrolled cell growth. Whereas cancer is often diagnosed suddenly and unexpectedly, the transformation from normal to malignant cells typically unfolds over several years, often long before any clinical symptoms appear. During this time, cells gradually acquire numerous genetic and epigenetic alterations that collectively and silently contribute to tumor formation [2].

The accumulation of these alterations is largely driven by well-established risk factors, such as aging and exposure to carcinogens, like radiation or tobacco smoke. They increase the risk of cancer by causing epigenetic modifications, e.g., changes in DNA methylation, and triggering *genetic events*, e.g., somatic mutations in the DNA sequence. These include base substitutions, insertions, deletions, rearrangements, and copy number variations that can even result in the complete loss of specific genomic regions [3, 4].

One of the most dangerous consequences of these molecular alterations is the ability to invade tissue. Unlike benign tumors, cancer is defined by its ability to infiltrate healthy tissue and potentially disseminate to distant sites in the body. This process, known as *metastasis*, involves cancer cells breaking away from the primary tumor, entering the bloodstream or lymphatic system, and forming secondary tumors in distant organs. Metastasis is the leading cause of cancer-related deaths [5, 6].

However, acquiring these lethal capabilities requires cancer cells to develop far more than just uncontrolled cell growth. For instance, they must also evade immune surveillance, resist cell death, and establish their own blood supply through a process called *angiogenesis*, thereby ensuring a continuous source of nutrients and oxygen. These biological capabilities that distinguish malignant from normal cells are described as the *Hallmarks of Cancer* that were first introduced by Hanahan and Weinberg in 2000 and later revised and expanded in 2011 and 2022 [7, 8, 9]. Currently, the core Hallmarks

of Cancer include: sustained proliferative signaling, evasion of growth suppressors, resistance to programmed cell death, replicative immortality, induction of angiogenesis, activation of invasion and metastasis, altered cell metabolism, evasion of the immune system, and finally, unlocking phenotypic plasticity.

Cells acquire these capabilities through a systematic process of cellular transformation best described in the *multistep tumor progression model*, which outlines four key steps in the development of cancer in somatic cells: initiation, promotion, malignant transformation, and progression [10]. As cancer evolves through these stages, cancer cells accumulate numerous alterations. Most of them have a neutral impact on tumor development (*passenger mutations*) and only a few of them are *driver mutations*, which actively promote tumor development, for example by fostering cell proliferation or other malignant traits [11].

To understand how driver mutations contribute to cancer, it is essential to distinguish between different functional classes of cancer-related genes. *Oncogenes* drive cancer progression when mutated or overexpressed, stimulating uncontrolled cell growth and survival. *Tumor suppressor genes* regulate cell growth and can therefore promote the progression of cancer when they are inactivated [11].

While identifying individual genetic alterations is undoubtedly crucial for cancer research, fully understanding the complexity of the disease requires seeing the bigger picture. In addition to determining the specific genetic events that occur during tumor development, researchers must also examine the order in which they occur and how they interact with one another [2].

1.2 Modeling Tumor Progression

1.2.1 Cross-Sectional Data

In cancer research, the importance of determining the sequential occurrence of genetic events has long been recognized. For instance, Gerstung et al. found that *APC* mutations have the highest odds of occurring early in colorectal adenocarcinoma, followed by *KRAS*, loss of *17p* and *TP53*, and *SMAD4* [2]. This finding suggests that understanding the chronological order of genetic events and identifying patterns is crucial for guiding treatment strategies and improving therapeutic outcomes.

Yet, most clinical data only provide static snapshots of tumors. For instance, to determine whether cells are cancerous, a small piece of tissue, known as a *biopsy*, is often taken from a suspicious area and sent to a laboratory for analysis. The type of data derived from a large number of different patient samples is referred to as *cross-sectional-data*, as it consists of observations from many different individuals at one single point in time (*across* individuals). In such a dataset, each data point typically represents a single patient (biopsy) and contains information about, for example, the mutations that have occurred to date. However, it offers no insight into the order in which those mutations arose or how the tumor has evolved over time.

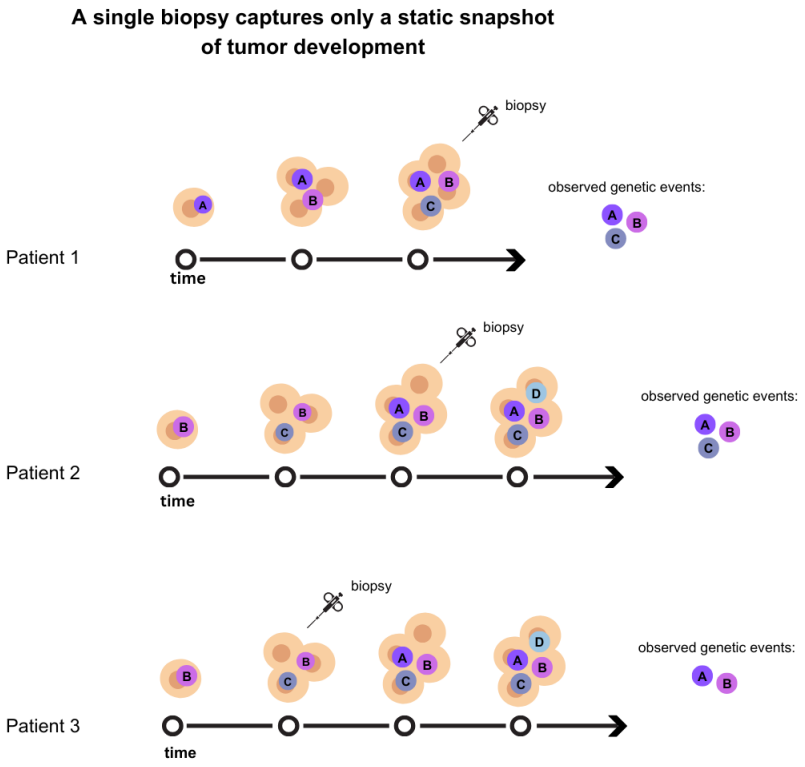


Figure 1.1: Simplified schematic representation of tumor progression over time and the limitations of biopsies in cross-sectional data. Each row represents a different possible progression state of genetic events (A, B, C, D) within a growing tumor. Since biopsies are taken at varying time points in the disease course and capture only a single snapshot of the tumor, it becomes difficult to reconstruct the true evolutionary path. Despite different trajectories, the resulting list of observed genetic events at the time of biopsy may appear identical.

To address this issue, *Cancer Progression Models* (CPMs) have been developed [12]. These models aim to reconstruct the sequence of genetic alterations that drive tumor development. There are different types of CPMs, differing in their mathematical approaches and assumptions. Roughly, CPMs can be divided into *deterministic* and *stochastic* models.

1.2.2 Mutual Hazard Networks

Deterministic CPMs model tumor progression as a fixed sequence of genetic events. An event can only arise if all previously defined "parent" events have already occurred. Stochastic CPMs, on the other hand, allow for probabilistic dependencies, where events with a non-zero probability can happen at any time [13].

Mutual Hazard Networks (MHN) are a machine learning algorithm and belong to the latter category of CPMs. MHN uses a stochastic approach and models tumor progression as a *Markov Chain*. Each event has a natural rate of occurrence (*base rate*), and the presence of other genetic events can increase or decrease that rate through *multiplicative effects* (interaction effects). These effects can be greater than one (promoting), less than one (inhibiting), and dependencies can even be cyclic [13]. For this reason,

MHN can generate more realistic and flexible cancer progression pathways, allowing for better insights into the underlying biology [14].

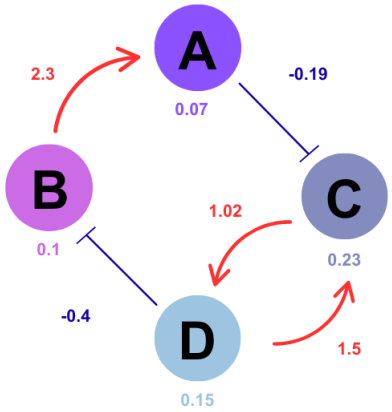


Figure 1.2: Simplified representation of the MHN algorithm. A, B, C, and D are genetic events that can occur in the process of cancer development. Each event is displayed with its natural rate of occurrence, e.g., A with a base rate of 0.07. The blue arrow represents inhibiting effects, and the red arrow represents promoting effects. Consequently, in the presence of B, the base rate of A is increased by $e^{2.3}$. The arrows between C and D represent cyclic dependencies in this illustration.

1.2.3 Theta Matrix

The algorithm learns how genetic events influence each other by estimating a *hazard matrix* also referred to as *Theta Matrix*, with interaction effects, and base rates from observed tumor data [14]. The recently introduced *oMHN* model, an extension of the classical MHN algorithm, also includes *observation effects*, which indicates the effect of a genetic event on the observation event, e.g., the point at which a tumor is clinically detected [14].

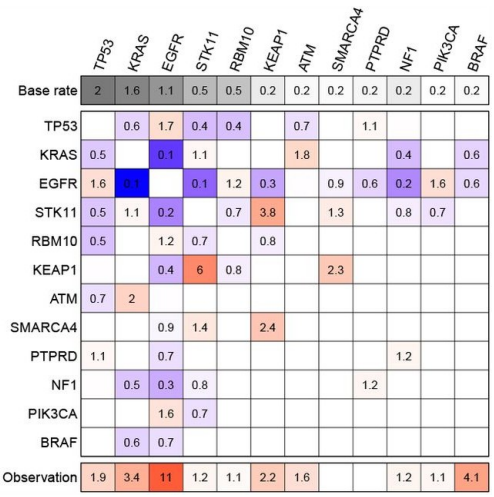


Figure 1.3: Example of a Theta Matrix on a colorectal cancer (COAD) dataset inferred by oMHN. The base rate indicates the rate at which a genetic event occurs on its own; the colored cells visualize multiplicative effects (promoting in red, inhibiting in blue); The values in the observation row equal the effect of a genetic event on the observation event, e.g., clinical detection. Blank cells equal a value of 0. All values are rounded to the first decimal (modified from Figure 4, Schill et al. [14]).

In a second step, MHN combines these parameters with genetic profiles from patient data to reconstruct individual History Trees, calculating the most likely evolutionary path for each tumor. For a detailed explanation of the mathematical principles underlying MHN, see Schill et al.[14, 13].

1.3 MHN Patient History Trees

MHN Patient History Trees visually represent the *inferred* (estimated) order of event occurrence based on the Theta Matrix.

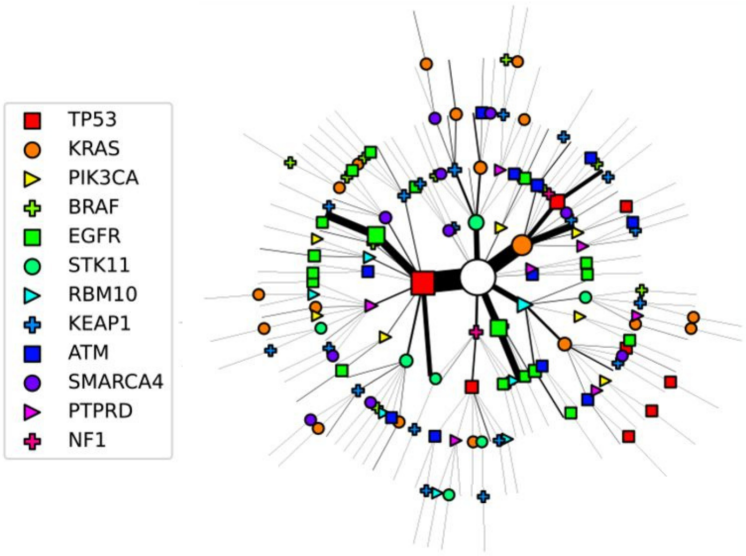


Figure 1.4: Example of a MHN Patient History Tree inferred by the oMHN model using the COAD dataset (modified from Figure 5, Schill et al. [14]).

The white circle in the center of the tree represents the root, which marks the common starting point for all tumor histories in the dataset. From there, each node represents a genetic event, placed in chronological order based on its most likely position of occurrence. The symbols and colors used for each genetic event are unique combinations, which help distinguish them from one another. The name of the genetic event can be taken from the legend on the left. Each branch of the tree represents an individual tumor evolution path, or "history", observed in the data. The thickness of each *edge* (path between two genetic events) corresponds to the number of patients who share that specific sequence of genetic events, in this work referred to as *count*. For improved readability, MHN Patient History Trees will hereafter be referred to as *History Trees*.

2 Motivation

Until now, the only way to illustrate History Trees has been by manually generating them. However, since they are static representations, even minor adjustments require the entire History Tree to be redrawn. This process is not only highly time-consuming and tedious, but it also demands knowledge in programming, which limits accessibility for researchers without a technical background.

Yet, the results inferred by the MHN algorithm are particularly relevant for oncologists. As mentioned before, they provide probabilistic insights into how tumors may evolve and support the interpretation of patient-specific mutation profiles. For MHN to be used effectively in clinical research and practice, History Trees must be presented in a way that is both accessible and intuitive for scientists and clinicians with different backgrounds.

To address this need, this work introduces an interactive web application that enables the visualization of the Theta Matrix, as well as the dynamic exploration of History Trees. By allowing clinicians and researchers to customize the History Tree visualization to their specific analytical needs without the need to manually redraw the tree each time, the tool supports both scientific analysis and interdisciplinary communication, making it easier for experts in various disciplines to collaborate effectively.

3 Technical Implementation

This chapter shortly describes the web application, provides an introduction to web development, and outlines the technical implementation of the app. Furthermore, it explains the technologies used in the development process, as well as how data is processed and visualized in the History Tree and Theta Matrix. It describes the structure and logic behind key components of the application. As this chapter is fairly extensive, the text box below provides an overview of the chapter's content.

Chapter Overview

- Web development fundamentals:
  - **3.2** Introduction to Web Development
- Overall architecture and technologies used:
  - **3.1** Short Description of the Web Application
  - **3.3** Technologies Used in the Project
  - **3.4** Project and Folder Structure
- Implementation details and logic:
  - **3.5** Setting up the Project
  - **3.6** Styling
  - **3.7** State Management and Context.tsx
  - **3.8** Visual Implementation with D3.js
- Challenges with D3.js:
  - **3.9** Challenges with D3.js Integration

3.1 Short Description of the Web Application

The MHN History Tree Web App is a web application designed to visualize the most likely evolutionary path of tumors inferred from patient data by the MHN algorithm.

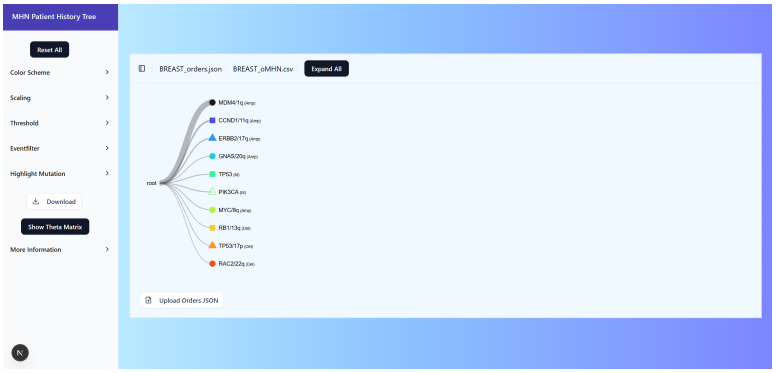


Figure 3.1: User interface of the MHN History Tree Web App

This tool is primarily intended for scientists and clinicians working with tumor data, regardless of their technical background. With this tool, users can examine how genetic events accumulate over time during the progression of tumors.

The application supports:

- importing data files
- visualization of the Theta Matrix
- interactive visualization of History Trees
- styling options (e.g., colors, edge scaling)
- filtering for parameters (event filtering, threshold)
- downloading the customized History Tree as *SVG* (Scalable Vector Graphics)

3.2 Introduction to Web Development

3.2.1 Definition

The term *web development* refers to the process of building, programming, and maintaining websites and web applications [15]. The main purpose of a *website* is to provide information. For example, the homepage of the *University of Regensburg* (`https://www.uni-regensburg.de/`) lists details about its faculties and departments, as well as informs about current news and upcoming events. A *web application*, on the other hand, is interactive software accessed through a browser (such as Chrome or Safari) that provides a specific service to the user. For instance, the social media platform *Instagram* allows users not only to view content, but also to interact with it, e.g., by commenting on posts [16].

3.2.2 Front and Backend Development

In the realm of web development, *frontend* and *backend development* can be distinguished. The frontend of a website refers to the parts that are visible and directly interacted with by the user, also known as the *user interface* or abbreviated as *UI*. The UI comprises elements such as buttons, menus, forms, and layouts. Technologies such as *HTML* (structure), *CSS* (styling), and *JavaScript* (interactivity) are typically used for frontend development [17]. In this work, user interface, UI, and frontend are used as synonyms.

The backend handles the underlying logic, for example, data processing. It ensures that user actions, such as submitting a form, are processed correctly and that data is stored and retrieved as needed. Typical programming languages for backend development are *Java*, *Python*, and JavaScript. *Fullstack development* is the generic term for the practice of both front and backend development [17, 15].

3.2.3 Document Object Model (DOM)

The *Document Object Model* (DOM) is an object representation of the HTML elements [18]. When a browser loads an HTML file, it converts the page's structure into a tree-like representation. The DOM allows, for example, JavaScript and other libraries to update the page content.

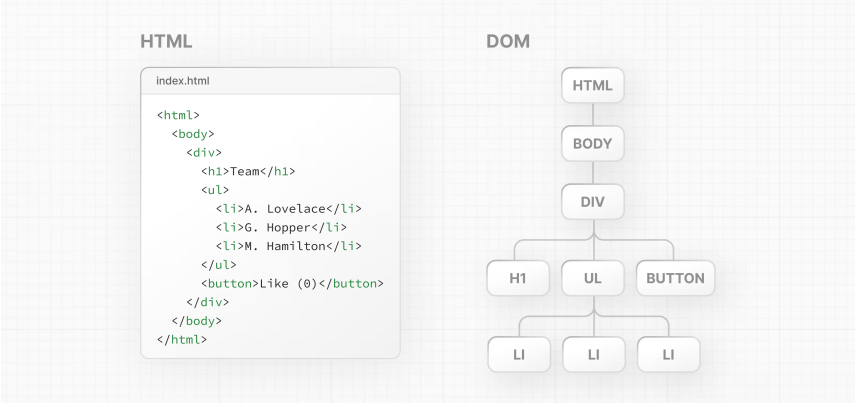


Figure 3.2: When a user visits a web page, the server returns an HTML file to the browser. The browser then reads the HTML and constructs the Document Object Model (DOM) (Figure adapted from Next.js documentation, Vercel [18]).

3.2.4 Client-Server Model

The front and backend communicate via the *client-server-model*. The *client* (typically a web browser) sends requests to the *server* (a program or computer), the server processes them and returns responses. For example, when a user visits the University of Regensburg's website, the browser (client) sends a request to the university's server, asking for the homepage. The server receives the request, processes it, and then sends back the data needed to display the website in the browser [19].

3.2.5 Rendering and Performance

A central element in this interaction is called *rendering*. Rendering describes the process of converting *markup* (e.g., HTML) into the visible layout that the user sees on the screen. Choosing where and how rendering happens can impact the *Performance* of a website, which refers to the speed at which a website loads and reacts. Web development requires a clear separation between the UI (client-side) and the backend (server-side), as well as an understanding of how data is rendered and delivered efficiently to ensure optimal performance.

3.3 Technologies Used in the Project

The development of this web application involved both *libraries* and *frameworks*, which are reusable pieces of code written by other developers to simplify software development. Although they serve a similar purpose, they differ in the level of control they leave to the developer. A library is a collection of pre-written functions that can be used to perform specific tasks, almost like a toolbox from which developers select the tools they need. A framework, in contrast, offers less flexibility, as it provides a predefined structure and calls the developer's code when required, thereby determining the overall flow and architecture of the application [20].

Table 3.1 provides an overview of the technologies used in the project and their respective purposes.

Table 3.1: Technology Overview

Technology	**Purpose**
`React`	builds the user interface and manages interaction logic
`Next.js`	provides structure, routing, and layout
`D3.js`	visualizes the History Tree and Theta Matrix
`shadcn/ui`	provides visually appealing, customizable UI components
`Tailwind CSS`	styles UI components and page layout
`TypeScript`	helps prevent errors and improves code quality
`Framer Motion`	adds animations and transitions to the interface

The main technologies used in the development of this web application represent the *React* framework *Next.js*, the data visualization library *D3.js*, the UI component library *shadcn/ui*, and the programming language *TypeScript*. The following section presents these tools and provides a brief explanation of why they were selected.

3.3.1 React

React (also known as React.js or ReactJS)(`https://react.dev/`) is a JavaScript library for creating user-friendly, interactive user interfaces. It was developed by *Facebook* (now Meta) and is one of the most widely used frontend tools in web development. React builds applications from small, reusable building blocks called *components*. Each component represents a specific part of the user interface in the application, for example, a button, a menu, or an entire page section. These components can be reused and combined in various ways to build the application.

3.3.2 Next.js

Next.js (`https://nextjs.org/`) is a modern React framework and simplifies the development of fullstack web applications by providing features such as automatic routing based on the file structure (3.5.3), different types of page rendering, and optimized performance by default. These features make it particularly suitable for building fast, dynamic, and interactive React applications.

3.3.3 D3.js

D3 (or D3.js) is a JavaScript library for visualizing data (`https://d3js.org/`). Its flexibility in authoring dynamic, data-driven graphics made it particularly interesting for the project. Further, it offers a wide variety of graph templates, two of which were used for this project (3.8).

3.3.4 shadcn/ui

To ensure a consistent and visually appealing user interface, shadcn/ui (`https://ui.shadcn.com/`) was used. The open-code UI library was created by a developer at the company *Vercel* called shadcn and is built on top of the component library *RadixUI* (`https://www.radix-ui.com/`) by *WorkOs* and the styling tool *Tailwind CSS* (`https://tailwindcss.com/`). A key advantage of shadcn/ui is that it provides access to the top layer of the component code, enabling complete control to customize and extend the components to specific project needs or even to build an entirely custom UI [21].

3.3.5 TypeScript

TypeScript is a so-called *static superset* of JavaScript, meaning it is JavaScript, but adds type annotations to the language, for example, by declaring that a variable must be a number. This helps catch mistakes early in the development process, before they lead to runtime errors (when the code is executed) or application crashes.

For example, TypeScript will report an error when assigning a string (e.g., `"Hello"`) to a function that expects a number:

Code Snippets 3.1: Example Code: TypeScript

```
let variable: number = 5
variable = "Hello"
//Error: Type 'string' is not assignable to type 'number'
```

JavaScript will not, as function parameters and variables do not have predefined types.

TypeScript was used consistently across all files in this project. However, using it alongside D3.js demonstrated significant challenges (see 3.9.3).

3.4 Project and Folder Structure

During the automatic setup of the project (3.5), Next.js creates a basic folder structure.

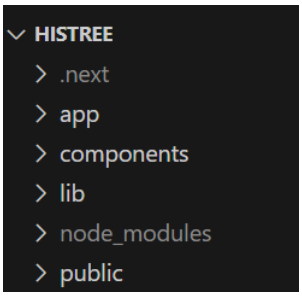


Figure 3.3: Top-level folder structure in the History Tree Project

The top-level folders include:

- `.next` folder: stores the compiled application, including static files, server-side rendered pages, and metadata
- `app` folder: represents the *App Router* (3.5.3)
- `lib` folder: intended for reusable logic or utility functions
- `node_modules` folder: contains all external dependencies and libraries installed
- `public` folder: stores static assets such as images, fonts, or files

The `components` folder is not part of the initial setup. It was added to organize UI and feature components of the History Tree application. The following figure provides a simplified overview of the project architecture and the technologies used in each part.

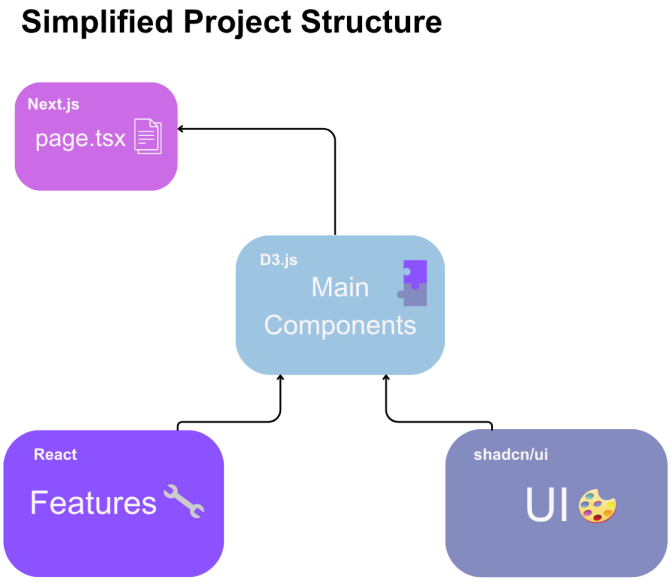


Figure 3.4: Simplified overview of the project structure and technologies used. Each label indicates the primary technology responsible for that part of the application.

The core of the application represents the so-called “main components”, which include the elemental functionalities of the project, mainly developed with D3.js: the interactive History Tree, the Theta Matrix, and the `sidebar`. The `sidebar` component combines reusable UI elements (`UI`), built with shadcn/ui, and functional logic (`Features`)(e.g., event filtering or threshold). The main components are rendered using Next.js.

The diagram below provides a more detailed overview of the internal folder hierarchy and its contents at the end of the development process, highlighting the most relevant elements of the project.

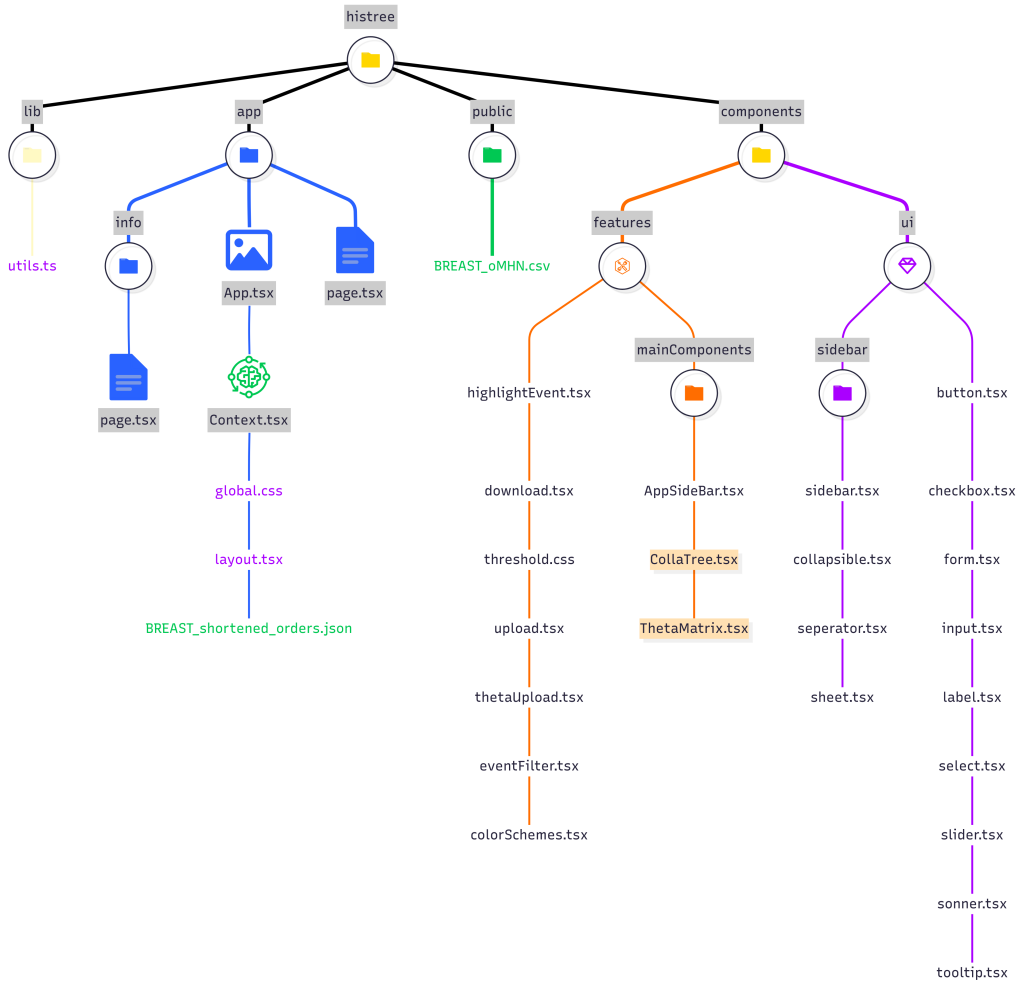


Figure 3.5: Overview of the folder structure at the end of the development process. The diagram highlights how the different parts of the application (UI components, logic, data, etc.) are structured and connected.

The top-level folder `histree` contains the entire project. Each folder inside of it serves a specific purpose: The `components` folder separates the UI building blocks (button, checkbox, etc.) from the `features` (core application logic like upload, download, etc.). The `public` folder stores data such as the default template dataset for the Theta Matrix, while the `lib` folder contains helper functions (`utils.ts`) required by the shadcn/ui elements.

All these parts come together in the `app` folder, which connects data, logic, and interface, and renders the complete graphical layout, which is then displayed on the screen.

3.5 Setting up the Project

3.5.1 Installation and Configuration

To set up a Next.js project, a suitable *development environment* with *Node.js* (`https://nodejs.org/en`) must first be established. A development environment includes the necessary software tools and runtime components required to create, build, and test the application. Node.js is required for a Next.js project as it allows the execution of JavaScript code outside the browser.

A Next.js project is initialized using a single command `npx create-next-app@latest` . During the setup process, the user is prompted to choose various options, such as whether to enable TypeScript support or use Tailwind CSS for styling in the project. Based on these selections, the corresponding configuration files and type definitions are added automatically. Alternatively, a Next.js application can also be set up manually by installing the necessary packages and creating the required structure from scratch.

3.5.2 TypeScript and ESLint Support

As mentioned above, Next.js comes with built-in TypeScript support, which is enabled when the `.tsx` file extension (e.g., `component.tsx`) is used. Additionally, Next.js includes an *ESLint* feature, which automatically installs the necessary packages and configures the proper settings when a new project is created. Both of these tools enhance code quality and readability by identifying errors during the development process. In this project, the default configurations for TypeScript and ESLint provided by Next.js were used.

3.5.3 File-System Based Routing

In Next.js 13, the modern routing system via the App Router was introduced, which uses the structure of the `/app` directory to define the *URLs* (routes) of a web application. Inside the main `app` folder, each subfolder represents a route segment, which is mapped to a corresponding segment in the URL path. For example, creating a `page.tsx` file in the `info` folder results in the URL `@/info` . A page will not be accessible in the browser unless there is a `page.tsx` file inside the folder. This approach is called *file-system based routing* because the way folders and files are organized directly determines which pages exist and how they are accessed [22].

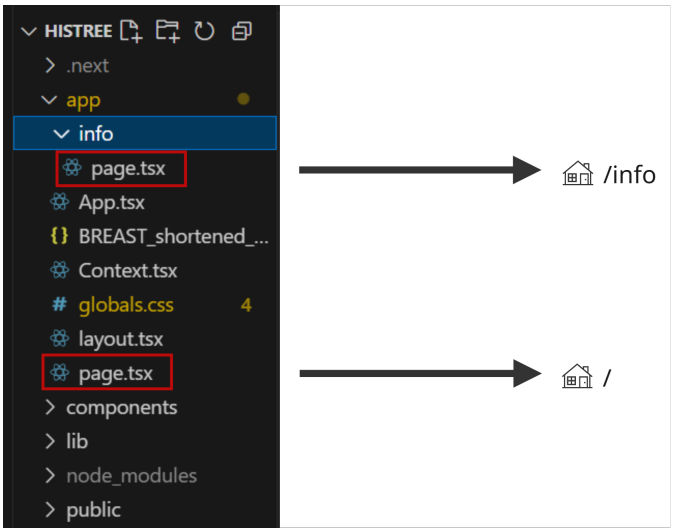


Figure 3.6: File-system based routing in the History Tree Project. The folder and file structure automatically creates two routes: `app/page.tsx` becomes `/` , and `app/info/page.tsx` becomes `/info` .

3.5.4 Start of the Development Server

To test and preview the application during the development process, developers use a so-called *development server*. It can be started with the command `pnpm run dev` in the terminal. The application is then available for viewing at `http://localhost:3000` in any web browser.

3.6 Styling

A combination of elements shapes the design of the project's frontend. In React and Next.js, *JSX* is the default syntax to describe UI components. It enables developers to write HTML-like code directly in JavaScript, which improves readability and structure.

3.6.1 Tailwind CSS

The standard language in web development to describe how elements should be displayed on the screen is called CSS (Cascading Style Sheets). The History Tree Project uses the CSS framework Tailwind CSS, which follows a *utility-first* approach. Styling is applied through small, single-purpose classes written directly in the markup.

Code Snippets 3.2: JSX Markup: Styling a Download Button Using Tailwind CSS

```
<Button variant="outline" onClick={downloadSVG} className="transition
    delay-150 duration-300 ease-in-out hover:-translate-y-1 hover:scale
    -110 hover:bg-indigo-800 text-slate-700 hover:text-white">
    <DownloadIcon className="mr-2 h-4 w-4" />
    Download
    </Button>
```

These utility classes, for instance `className="bg-indigo-800"`, which sets the background color to the Tailwind CSS color "indigo-800", represent specific style rules (properties) and can be freely combined. General styling rules that apply across the entire project are stored in the `global.css` file, which is automatically included at the root of the project during setup and contains the Tailwind CSS base styles.

3.6.2 Layout.tsx

In Next.js projects, `layout.tsx` is required for the overall page layout. It defines global page elements, such as headers or navigation components, that apply to all routes across the project. Since this project only contains two routes, each with its own `page.tsx`, `layout.tsx` was not actively used.

3.6.3 UI Components by shadcn/ui

The UI components were built using a combination of Tailwind CSS and shadcn/ui as demonstrated with the `Slider` UI element in the code below.

Code Snippets 3.3: JSX Markup: Customizing the shadcn/ui Slider Component

```
<Slider
   className="w-[60%]"
   value={[scalingFactor]}
   min={1}
   max={7}
   step={0.5}
   disabled={!scalingEnabled}
   onValueChange={([v]) => {
   if (scalingEnabled) setScalingFactor(v);
   }}
    />
```

The `Slider` component is used to control edge scaling in the visualization [23]. It was imported into another component and integrated via `<Slider{...}/>`. The `Slider` was then customized by defining parameters such as its minimum and maximum values, step size, and width (`w-[60%]`, which sets the width to 60%).

In some cases, UI components were also combined to build customized frontend elements.

3.7 State Management and Context.tsx

Components often need to change what is displayed on the screen in response to user interaction. For example, when the user adjusts the `Slider`, the component needs to remember and update the current value. This "memory" of components can be achieved through *State Management* [24].

For this, React provides built-in functions called *Hooks*, such as `useState`, that make it easy for components to store and change data.

Code Snippets 3.4: Example Code: State Management scalingFactor

```
const [scalingFactor, setScalingFactor] = useState<number>(1);
```

In this example, the line of code does two things: it defines the state variable `scalingFactor` with a default value of 1, and provides an update function `setScalingFactor`. When a user interacts with the `Slider` (drags it to change the value), an event handler (a function that responds to a user action, e.g., `onValueChange` in 3.6.3) calls `setScalingFactor({newValue})`. This updates the internal state of the value.

For the stroke of the edges to change in the History Tree, the value of `scalingFactor` must be passed to `CollaTree.tsx`, which is achieved by a mechanism called a *prop* (short for "property"). Props are used to pass data from a parent component to a child component. If a prop changes its value, an automatic re-render of all components that use it is triggered.

Code Snippets 3.5: Example Code: Passing Props in React

```
function Welcome(props) {
  return <h1>Hallo {props.name}</h1>;
}
```

While this approach to handling data works well for isolated components, it becomes difficult and inefficient in larger applications where multiple components require access to the same data. For instance, the names of the genetic events in the dataset are relevant for different features as well as the main components.

To avoid manually passing props through multiple layers (also known as *prop drilling*), React *Context* can be used [25]. The `Context.tsx` file defines a global state container, making information available to all components in the app. This shared state architecture ensures consistent updates across the entire application, significantly simplifying the development process, especially when combining different technologies with React, such as D3.js.

3.8 Visual Implementation with D3.js

Both the History Tree and the Theta Matrix posed highly specific visualization requirements. While the general direction of the visualizations was clear, the exact layout, structure, and interaction design had to be developed during the project. Therefore, a data visualization tool was needed that allowed for experimentation and offered maximum flexibility to find custom solutions without technical limitations.

D3 provides exactly these requirements. It was chosen for the project because it offers exceptional flexibility and almost unlimited possibilities for creating visually appealing, highly customized data visualizations. D3 can be paired with web frameworks such as React, which was essential for enabling interactive features that allow users to modify the visualized graph in real-time, and allows direct control over every element of the visual output by binding data to DOM elements.

3.8.1 Templates and Visualization

3.8.1.1 History Tree

For the implementation of the History Tree, the code of the *Collapsible Tree* template from the online platform for interactive data visualization *Observable* (`https://observablehq.com/`) was copied and then customized to meet the specific requirements of visualizing History Trees in Next.js [26].

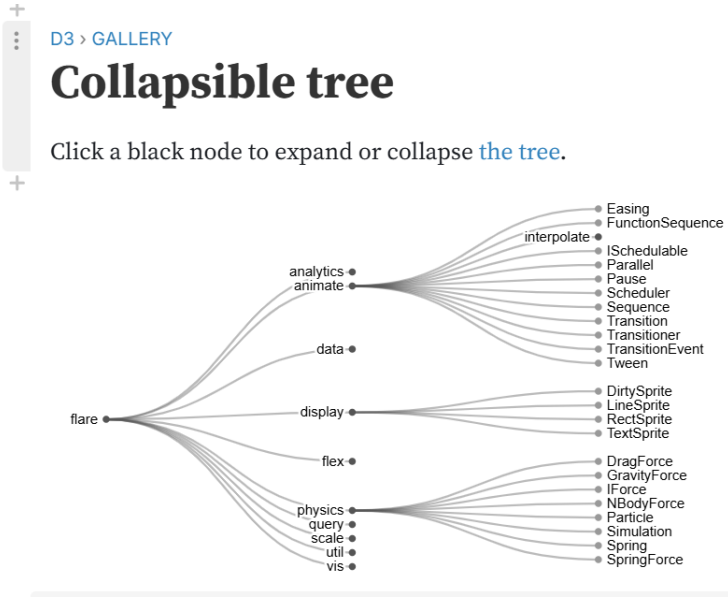


Figure 3.7: Collapsible Tree template at Observable by Mike Bostock [26]

The template was chosen for this project because its collapsible structure enables the possibility of expanding or collapsing genetic events, making it easier to navigate larger datasets while preserving an overview of all tumor progression paths. Further, the template also provided integrated animations, which contributed to a particularly appealing hierarchical data representation.

The nodes were replaced with a customized, unique color-symbol combination for each genetic event, replicating the style of the initial History Trees. The colors were dynamically generated based on the number of genetic events using the `d3.quantize method` in combination with color schemes from the `d3-scale-chromatic module` . To assign the individual color-symbol combination to genetic events, as well as to scale the edge thickness based on the patient count, the method `d3.scaleLinear` was used.

Code Snippets 3.6: D3 Code: Mapping Symbols to Genetic Events with d3.scaleOrdinal

```
const symbols = [d3.symbolCircle, d3.symbolSquare, d3.symbolTriangle];
const shapeScale = d3.scaleOrdinal<string, d3.SymbolType>()
    .domain(finalOrder)
    .range(symbols)
```

Node labels were adjusted to the corresponding genetic event names. Existing animations from the original template have been adapted, and additional features, such as `Collapse All` and `Download SVG` , have been implemented.

3.8.1.2 Theta Matrix

The Theta Matrix was implemented, adapting the *Heat Map with Tooltip* template from the *D3.js Graph Gallery* (`https://d3-graph-gallery.com/`) [27]. Similar to Observable, the gallery offers an extensive collection of well-documented, reproducible, and editable D3 chart examples.

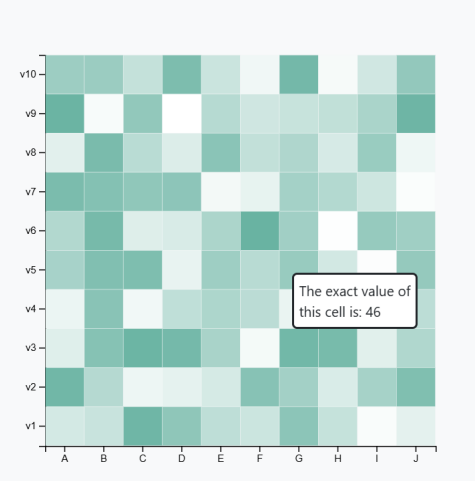


Figure 3.8: Heat Map with Tooltip template by Yan Holtz [27]

In contrast to the History Tree, which required a hierarchical structure, the Theta Matrix represents pairwise relationships that are best displayed in a grid format. As heatmaps are specifically designed to visualize the strength and patterns of such pairwise interactions, this template was selected.

An additional row was added below and above the matrix for the base rate and the observation effects. Colors were adjusted using the `d3.scaleLinear` method to reflect the intensity of values. Labels were adapted to display the names of the corresponding genetic events. The integrated tooltip was used to illustrate the genetic event names in their reading direction, improving the interpretability of the represented multiplicative effects.

To keep the user interface tidy and avoid visual overload, the Theta Matrix is not displayed by default when the app is opened. Instead, it can be revealed via a *toggle* effect implemented through a button outside of D3 code in the `sidebar` .

Code Snippets 3.7: JSX Markup: Toggle Function for Displaying the Theta Matrix

```
        <div className="ml-10 mt-3">
          <Button className="transition hover:-translate-y-1" onClick={() =>
              setShowMatrix(!showMatrix)}>
            {showMatrix ? "Hide Theta Matrix" : "Show Theta Matrix"}
          </Button>
        </div>
}
```

To improve the user experience, the toggle effect is accompanied by a smooth animation created with the *Framer Motion* library for React (`https://motion.dev/`).

3.8.2 Data Management

Data management refers to how the application handles and organizes the input data used in the visualizations. This includes uploading the data, transforming it into the required formats, and ensuring it is correctly passed to the appropriate components within the application.

3.8.2.1 History Tree

In this project, data for the History Tree is loaded from static *JSON* (JavaScript Object Notation) files - a text format used for storing and transporting structured data.

The files have a hierarchical structure in which each genetic event is represented as a node, containing the following properties:

- `name`: the name of the event (e.g., *TP53 (M)*)
- `count`: the number of patients that share the exact same genetic event history
- `children`: a list of subsequent genetic events

Users can upload JSON files via a `file input` field. When a user selects a file, the application uses the browser's `FileReader API` to read its contents.

Once uploaded, the JSON content is converted into a JavaScript object and passed to the application's internal state via the `onUpload` function. The resulting data is then forwarded to the tree component as a prop (`treedata`), which triggers a re-render and redraws the History Tree with the new, uploaded data.

For the initial render, a template breast cancer dataset is stored locally in the `app` folder and loaded by default.

Figure 3.9: Excerpt of the template breast cancer JSON dataset

3.8.2.2 Theta Matrix

Similar to the JSON upload, users can upload the corresponding Theta Matrix in *CSV* (Comma-Separated Values) format using a `file input` field.

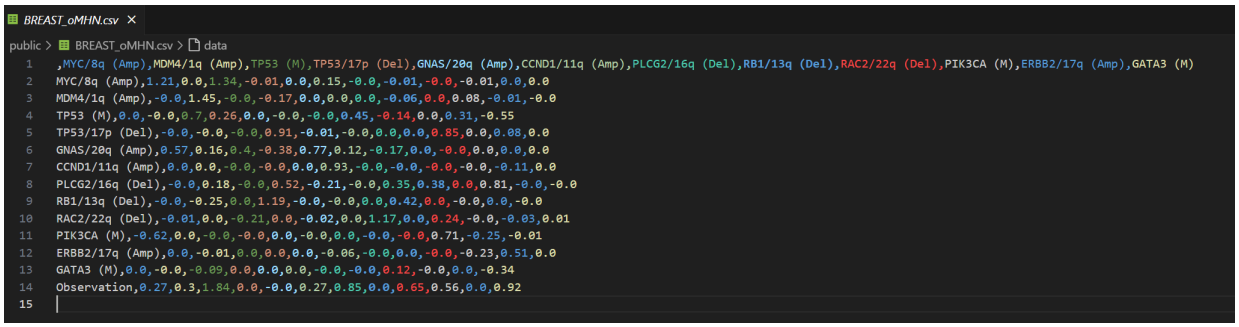


Figure 3.10: Default breast cancer CSV file in the History Tree Project

When the user selects a file, its content is read directly in the browser. The D3.js helper function `csvParseRows()` processes the table, extracts the relevant information, and converts it into a structured format. Each value describes the interaction between two genetic events: a source event (`row`), a target event (`column`), and a numerical value (`value`) that equals the multiplicative effect.

With the `onThetaUpload` function, the processed data is then passed to the application's internal state and forwarded to the Theta Matrix visualization, which is automatically updated to reflect the uploaded matrix.

The respective CSV file for the breast cancer orders is stored in the `public` folder and loaded by default.

3.9 Challenges with D3.js Integration

3.9.1 Syntax

Working with D3 proved to be more challenging than initially expected. While D3 is based on JavaScript, its syntax and structure differ significantly from plain JavaScript. The example below demonstrates this difference by showing how to create a circle using both approaches.

In plain JavaScript, the process involves multiple steps: first creating an SVG element, then creating a circle element, assigning attributes to it, and finally appending the circle to the SVG.

Code Snippets 3.8: Example Code: Creating a Circle with JavaScript

```
const svg = document.createElement("svg");
const circle = document.createElement("circle");
circle.setAttribute("fill", "red");
svg.appendChild(circle);
```

D3, in contrast, uses a *method chaining* syntax. Instead of executing each step separately, elements are selected, created, and modified in one command chain. The example below selects the HTML body element in the DOM, appends an SVG element, creates a circle, and assigns visual attributes in one flow.

Code Snippets 3.9: Example Code: Creating a Circle with D3.js

```
d3.select("body")
  .append("svg")
  .append("circle")
  .attr("fill", "red");
```

D3 code tends to be shorter and more compact than plain JavaScript. However, this can make it harder to read and understand because the chained syntax gives little information about what is happening step by step. This becomes particularly challenging when working with more complex data structures, such as the hierarchical format of the History Tree, where the logic behind each step is not immediately obvious.

3.9.2 Lack of Tutorials

This challenge of understanding D3 code was closely linked to another major difficulty during the development process: the lack of modern tutorials and documentation, especially for combining D3 with React and TypeScript.

At the beginning of the project, I completed several tutorials to familiarize myself with the required frameworks, libraries, and languages. While excellent resources exist for Next.js, React, JavaScript, HTML, and CSS, such as the official Next.js tutorial [28] or online platforms like *W3Schools* (`https://www.w3schools.com/`), learning D3 was much more difficult.

Many of the online resources were either too complex, limited to very basic examples, or simply listed available D3.js methods without offering practical exercises or

comprehensible examples for working with it, including the documentation of D3 itself [29]. Eventually, a D3 tutorial on *freeCodeCamp* proved helpful. Yet, it did not cover hierarchical and dynamic graphs, such as the Collapsible Tree, either [30].

As a result, understanding and adapting the Collapsible Tree template for use in this project required considerable time and substantial effort.

3.9.3 Compatibility Challenges with React and TypeScript

3.9.3.1 D3.js and TypeScript

Another challenge in working with D3.js was its limited compatibility with TypeScript. While type definitions in D3 do exist (`@types/d3`), they are large and complex, often resulting in type errors when chaining methods or working with dynamic data. This issue is well-known in the developer community and is frequently discussed in forums (e.g., [31]). The most common workaround for this incompatibility is to partially bypass TypeScript checks (for instance, setting the type to `as any`) or isolating D3 logic from the rest of the application.

In this project, TypeScript was consistently used throughout the entire application, including in the D3 logic. Although no runtime errors were encountered, some type-related errors remained unresolved by the end of development and required certain TypeScript rules to be relaxed.

3.9.3.2 DOM Control Conflicts with React

JavaScript libraries like React or D3.js manipulate the DOM to dynamically update the frontend without reloading the entire page as previously described in 3.2. However, they differ substantially in how they manipulate the DOM. React manages the DOM through a *virtual DOM* system, which is an internal copy of the actual DOM that React uses to efficiently determine which changes need to be applied. When the state of a component changes, React updates the virtual DOM first, compares it to the previous version, and then applies only the differences to the real DOM. D3, by contrast, updates DOM elements directly (3.9.1).

This becomes problematic when both libraries try to control the same elements, because layout or style definitions applied through React's JSX (e.g., via className or Tailwind CSS) do not affect DOM elements that were created and styled by D3. This issue occurred in the History Tree visualization. Since the History Tree SVG is dynamically generated and styled by D3, React cannot directly influence its appearance via JSX-defined class names. Attempts to apply styling using Tailwind CSS in `page.tsx` had no visible effect on the SVG. It was difficult to understand why certain layout or style changes were not working as expected, which cost a significant amount of time and effort to troubleshoot. This particular problem was partly resolved by removing as much D3-specific styling from the SVG as possible, allowing React to control its appearance better.

To avoid further issues, D3 logic was isolated within React components. The D3 rendering was triggered using the `useEffect` hook, and DOM access was handled through `useRef` . This setup ensured that D3 only modified the DOM once React had

rendered the component, allowing data from the React state (including Context) to be passed into the D3 logic.

Later in the development process, I came across the `react-d3-tree` library [32], which aims to simplify exactly this issue by offering a tree visualization explicitly built for React using D3 and TypeScript. At that point, however, the main parts of the application had already been implemented. Still, its existence underlines that integrating D3 into React using TypeScript is a well-known challenge in the development community.

4 User Guide

While the previous chapter focused on the technical background and the development process of the web application, this section provides a practical guide for users. It explains how to download the project, start the application, upload data, and use the built-in features to customize and explore the History Tree as well as interpret the Theta Matrix. The goal is to enable users, regardless of their technical background, to independently and effectively use the application and gain meaningful insights from their dataset.

4.1 Requirements

To run the MHN History Tree Web Application locally, the following software components must be installed on the system:

- Node.js (version 18 or later), available at (`https://nodejs.org/en/download`)
- npm (included in Node.js download) - package manager
- Modern web browser (for example, Google Chrome or Mozilla Firefox)

Node.js provides the *runtime environment* necessary for executing the programming language JavaScript outside of the browser, which was used to build this application. *npm*, which comes bundled with Node.js, is a *package manager* responsible for installing, updating, and managing the project's dependencies. These are prebuilt components, such as libraries and modules, that the application requires to work correctly. More information about the technologies used in the development process can be found in section 3.3.

In addition to the established runtime environment, these input files can be uploaded for visualization:

- JSON file (orders): This file contains the most likely reconstructed tumor history paths for individual patients. It is required to display the History Tree.
- CSV file (rates): This file contains the multiplicative effects between genetic events, as well as base rates and observation effects. It is used to visualize the Theta Matrix.

4.2 Installation and Start of the Project

4.2.1 Downloading the Project

Once all system requirements have been met, the web application can be downloaded with the following steps:

1. Visit `https://github.com/Antonia-gthb/historytree`
2. Click on the green button and select "Download ZIP"
3. Unpack the ZIP file (right-click ⇒ "Extract All")

4.2.2 Starting the Project

1. A terminal must be opened in the **unpacked project folder** by right-clicking on the folder and selecting "Open in Terminal".

   As an alternative, a terminal can be opened manually and navigated to the project folder using the `cd` command:

   >_ Windows: Click on the ⊞ button or the 🔍 symbol in the taskbar and search for "Terminal".

   🖿 For example: `cd C:\Users\admin\Downloads\historytree-main\historytree`

   >_ macOS: Open the Launchpad and enter "Terminal" in the search field.

   🖿 For example: `cd ~/Downloads/historytree-main/historytree`

2. The app is located in the subordinate directory "historytree", which can be reached by using the `cd` command as shown above
3. The required dependencies are installed by entering and executing:

   `npm install`

   This step is only necessary during the initial setup.

4. The application is compiled with:

   `npm run build`

   This step is only necessary during the initial setup.

5. The application is started in production mode using the command:

   `npm start`

6. Once the server is running, the application can be accessed in a web browser of choice by copying this URL into the address bar:

   `http://localhost:3000`

7. The application can be exited by pressing:

   `strg + c` or `control + c` ⇒ yes

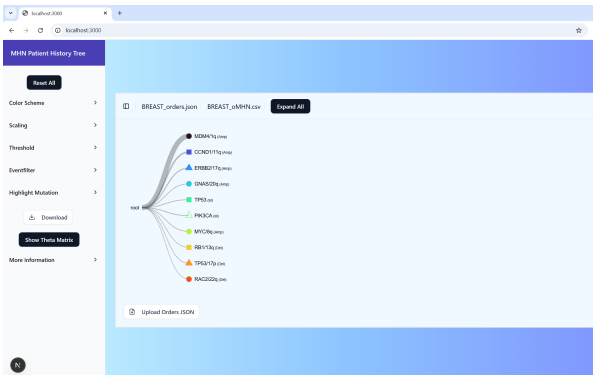


Figure 4.1: Initial user interface after starting the app locally with `http://localhost:3000`.

4.3 Example Workflow with Breast Cancer Data

To demonstrate how to use the web application, the breast cancer dataset, which originally stems from the work of Schill et al. [14] on oMHN, is used to present an example workflow in the app.

4.3.1 Breast Cancer Background

Over the past few decades, biomedical research has made remarkable progress concerning the screening, diagnosis, and treatment of breast cancer patients. Due to improved early detection methods, such as mammography, as well as advanced treatment options, the number of deaths from breast cancer has been decreasing for years [33].

Nevertheless, breast cancer remains a significant health concern. Approximately one in eight women will develop breast cancer during her lifetime [33]. According to the Federal Statistical Office, it is still the fourth most common cause of death among women [34].

Early detection plays a crucial role in improving prognosis and increasing the chances of recovery. Understanding the sequence of genetic events that occur during tumor evolution can provide valuable insights into early tumorigenesis, potential therapeutic targets, and personalized treatment strategies. The MHN History Tree Web App supports this goal by visualizing reconstructed genetic event sequences, thereby facilitating the identification of relevant patterns in breast cancer research.

4.3.2 Uploading Input Files

As previously mentioned, a JSON file is needed to visualize the History Tree. Optionally, the corresponding CSV file can also be uploaded to support the interpretation of the History Tree. Both files are generated in a 2-step process by the MHN algorithm and are typically provided by bioinformaticians.

In this example, the breast cancer dataset is used to demonstrate the application's functionality (`BREAST_orders.json` and `BREAST_oMHN.csv`). In a regular workflow, users can upload their files via the respective upload buttons. Upon clicking, a window opens, allowing users to select the desired file from their local system.

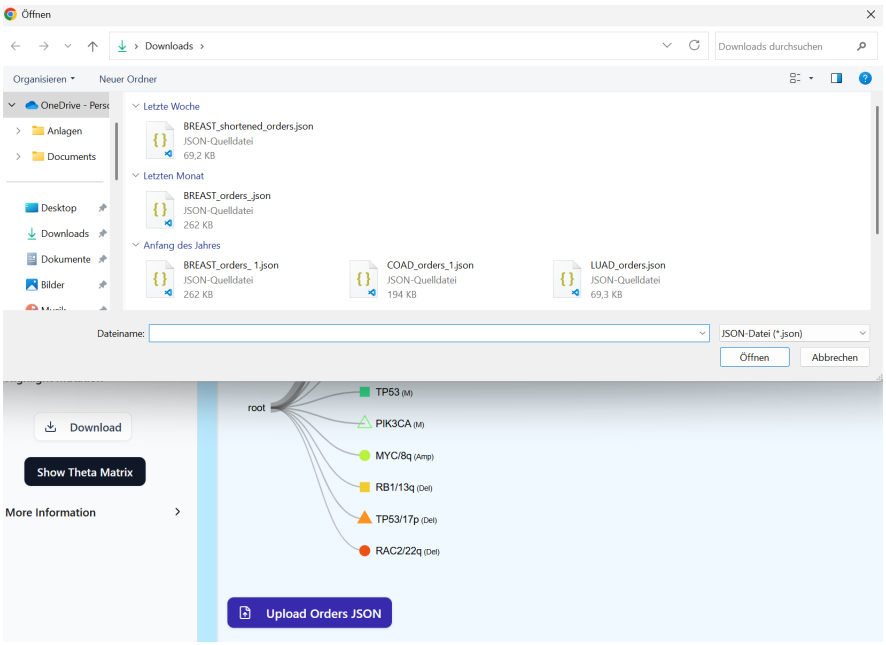


Figure 4.2: User interface after clicking on JSON upload button and opening a file selection dialog.

Both files can be uploaded independently. Once selected, their file names appear above the History Tree visualization to ensure that users always know which data they are currently viewing (see Figure 4.4). The upload button for the Theta Matrix is revealed by clicking on the `Show Theta Matrix` button.

Figure 4.3: Upload buttons for the JSON and CSV file in the user interface after revealing the Theta Matrix by clicking the `Show Theta Matrix` button.

The following section provides an overview of the user interface and explains its main components.

4.3.3 User Interface

After successfully starting the application in the browser and uploading a JSON file, the web application automatically renders the corresponding History Tree, presenting an intuitive interface consisting of two main sections: the sidebar on the left, highlighted in yellow, and the visualization panel on the right, highlighted in red.

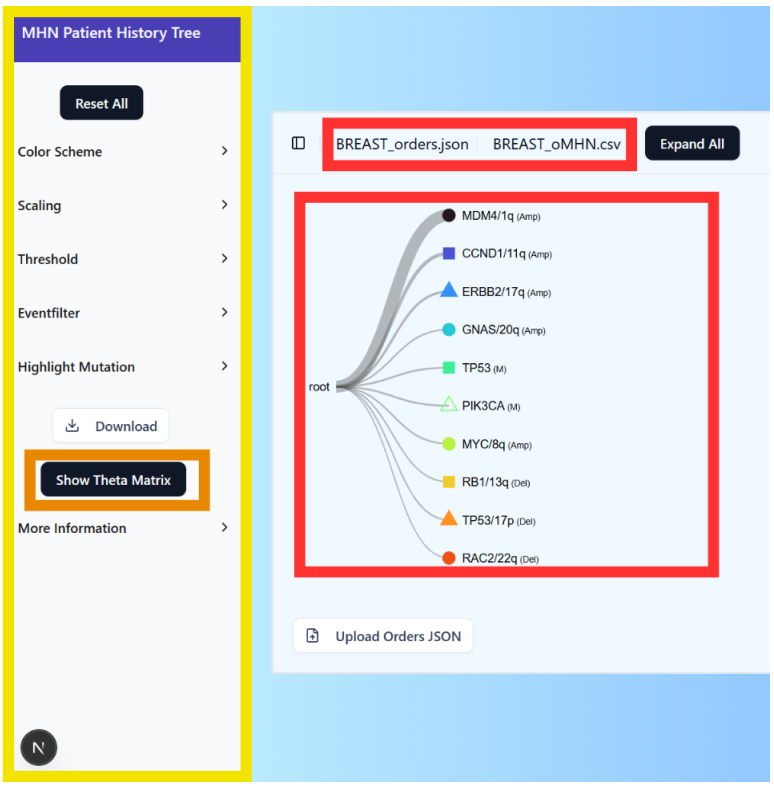


Figure 4.4: User interface view after uploading the input file(s). The sidebar (yellow) contains several customization features; the main panel (red) displays the History Tree and the names of the uploaded files above. The button highlighted in orange reveals the Theta Matrix.

The sidebar enables users to adjust various parameters, including the color scheme, threshold values, edge scaling, and event filtering. Clicking the `Reset` button at the top of the sidebar sets all features back to their default values. Additionally, the current History Tree view can be downloaded using the `Download` button, and the Theta Matrix (accessible via the orange-highlighted button) can be toggled on or off.

The sidebar also includes a collapsible section labeled “More Information”, which provides access to the following resources:

- a link to the subpage `/info`, which contains the user guide
- a link to the PDF version of this master’s thesis
- and a link to a *Miro* board (`https://miro.com/`) in the online workspace, providing an overview of the project structure and code, which was developed in the course of this project

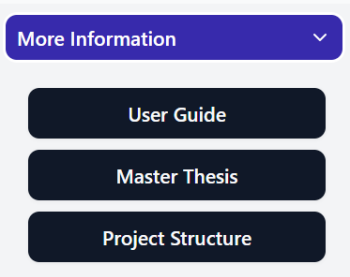


Figure 4.5: Collapsible section "More Information" in the sidebar, containing links to additional resources such as a user guide or the project structure on Miro.

The visualization panel represents the main part of the user interface, displaying the interactive History Tree. The name of the uploaded file (handled by clicking on the `Upload Orders JSON` button) is shown above. The `Expand All` button fully expands the History Tree (see Figure 4.7).

4.3.4 History Tree View

i Important for interpretation

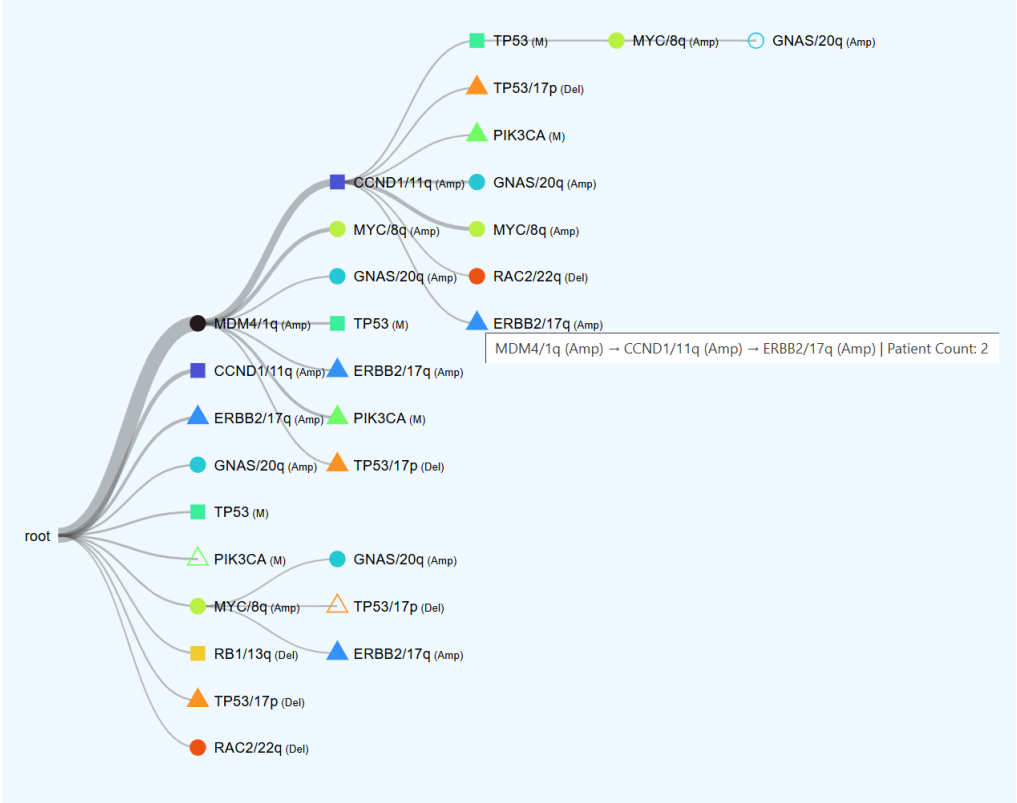


Figure 4.6: User interface after expanding nodes in the History Tree. Filled symbols indicate parent nodes with children, while unfilled symbols represent terminal nodes with no children. A tooltip with event information, including patient count and sequence up to the selected event, appears on hover.

Each node corresponds to a genetic event, distinguishable by a unique combination of color and symbol. The letter in brackets describes the type of genetic event. "Del"

stands for *deletion*, "M" for *point mutation*, and "Amp" for *amplification*. Filled symbols indicate that a node has children (further nodes), while hollow symbols (stroke only) represent terminal nodes with no children. By clicking on a node, the subordinate node(s) is/are expanded (or collapsed), accompanied by an animation. A tooltip (a small, interactive pop-up element that appears when hovering over a node) with information about the patient count and the genetic event trajectory appears on hover, as pointed out in the Figure above. The link between two genetic events (referred to as *edge*) is determined by the patient count, providing insight into how many patients share identical tumor histories.

4.3.5 History Tree Features

In addition to the features displayed in the History Tree, users can access further interactive elements, primarily located in the sidebar, which are listed and described in the following subsections.

4.3.5.1 Expand All

The `Expand All` button expands all nodes within the History Tree, providing a comprehensive overview of the complete structure.

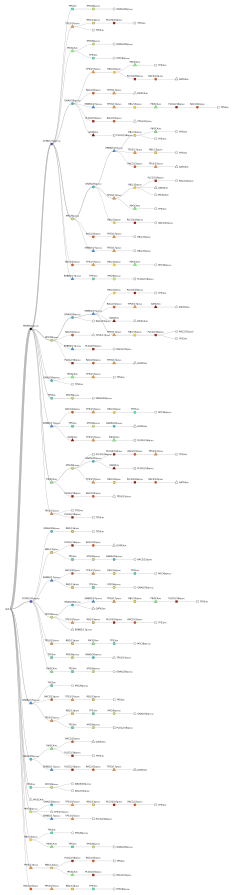


Figure 4.7: Fully expanded History Tree after clicking the `Expand All` button. The tree displays all nodes and branches in the dataset.

4.3.5.2 Coloring

In the first collapsible section of the sidebar, a custom color scheme can be selected. Overall, there are six different color schemes to choose from. “Turbo” is set by default.

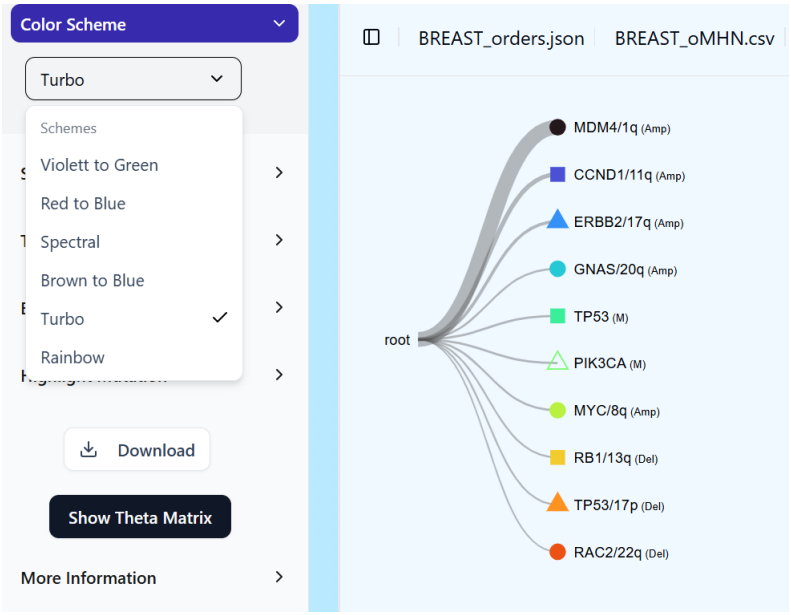


Figure 4.8: User interface after opening the color scheme selection menu in the sidebar. “Turbo” is selected by default.

4.3.5.3 Edge Scaling

i Important for interpretation

The edge scaling slider linearly maps the stroke width to the patient count. In other words, the more patients share a specific tumor sequence in their tumor history, the thicker the corresponding edge is rendered. When the slider is deactivated through deselecting the checkbox, all edges are rendered with equal thickness. Reactivating the checkbox sets the slider back to its default value.

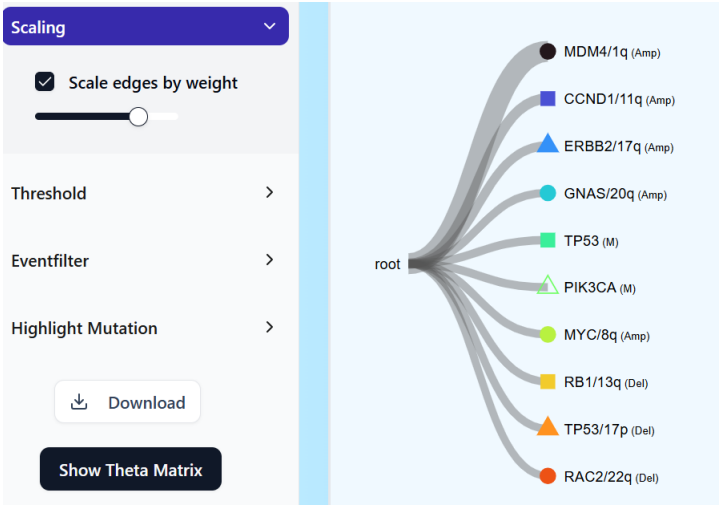


Figure 4.9: User interface after adjusting the slider for edge scaling. By deselecting the checkbox, all edges are displayed with the same stroke width.

4.3.5.4 Threshold

ℹ Important for interpretation

The threshold input field allows users to filter data based on a minimum patient count.

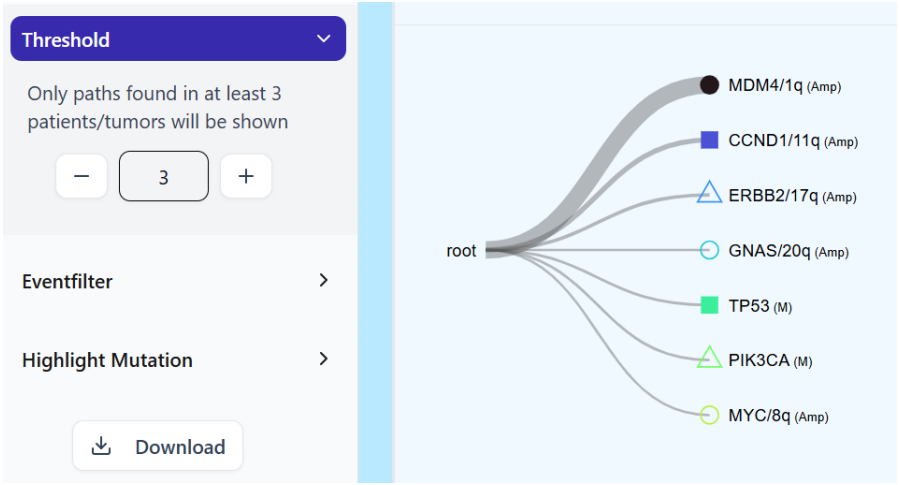


Figure 4.10: User interface after filtering out edges with counts below the input value "3".

For example, setting the threshold to "3" leads to filtering out genetic events with a patient count of "2" or less. Therefore, only paths found in **at least** three patients or tumors are displayed as illustrated in the Figure above.

4.3.5.5 Event Filtering

ℹ Important for interpretation

By default, all genetic events are selected and displayed. Using the event filter, specific genetic events can be selected or deselected for visualization. The desired changes can be submitted via the provided `Submit` button.

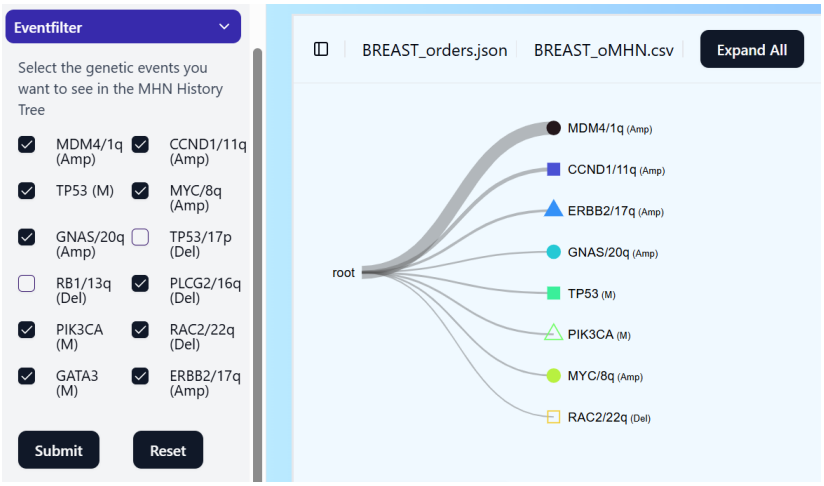


Figure 4.11: User interface after deselecting *TP53/17p* and *RB1/13q*. The History Tree does not display these genetic events.

The `Reset` button restores the default setting (full selection).

4.3.5.6 Highlight Paths

ℹ Important for interpretation

The "Highlight Paths" feature allows users to select a specific genetic event and highlight all paths in the tree that include it. However, **only the paths currently expanded in the History Tree view are taken into account**. If additional nodes are expanded afterwards, the feature must be executed again to ensure that all relevant edges are highlighted.

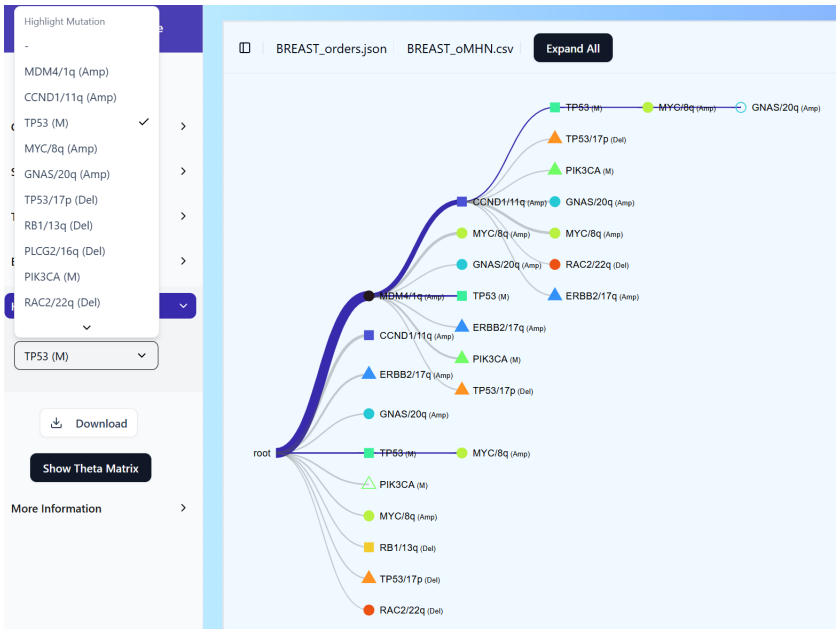


Figure 4.12: User interface after highlighting paths containing *TP53 (M)* in the current History Tree view.

Additionally, individual paths can be highlighted by clicking directly on an edge within the tree. When a path is selected in this way, all preceding nodes are automatically highlighted as well to provide full context.

4.3.6 Theta Matrix

ℹ Important for interpretation

As mentioned before, the Theta Matrix is revealed by clicking on the `Show Theta Matrix` button. The matrix shows how genetic events influence each other, comprising both promoting and inhibiting effects (multiplicative effects). The Theta Matrix also includes base rates, which indicate the frequency at which an event occurs independently of other genetic events. The observation column represents the rate at which a tumor is clinically detected, based on the preceding events.

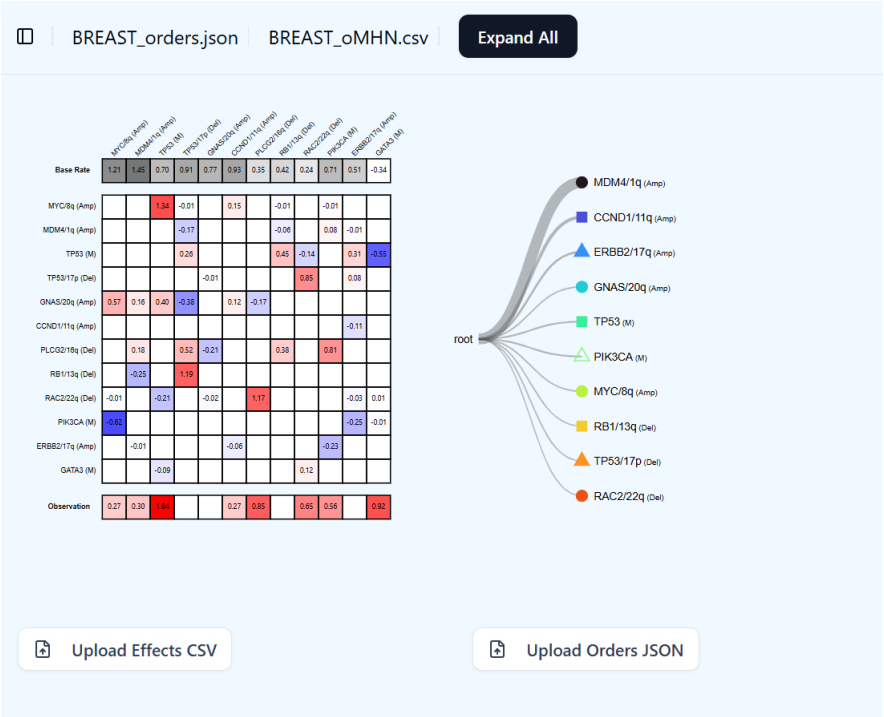


Figure 4.13: User interface after toggling the Theta Matrix. Rows and columns represent genetic events. Cell colors encode the strength and direction of multiplicative effects. Red for promoting effects, blue for inhibiting effects. The value zero leads to a blank cell. Base rates represent the rate at which an event occurs independently, while the observation column indicates the rate at which a tumor is clinically detected, based on the preceding events.

The Theta Matrix uses a color gradient to visualize the effect values: Blue for negative values below 0 (inhibiting effects), red for positive values above 0 (promoting effects). Blank cells represent the value 0. The intensity of the color reflects the effect size. The top row visualizes the base rates, and the bottom row shows the observation effects. The deviating values (e.g., promoting effects > 0, not > 1 as described in the introduction) can be attributed to the use of the natural logarithm on the values. Further details on the mathematical background can be found in the paper by Schill et al. [14].

i Important for interpretation

The matrix is read from column to row, i.e., the value in a cell represents the influence of the event in the column on the event in the row. The correct interpretation is supported by the tooltip.

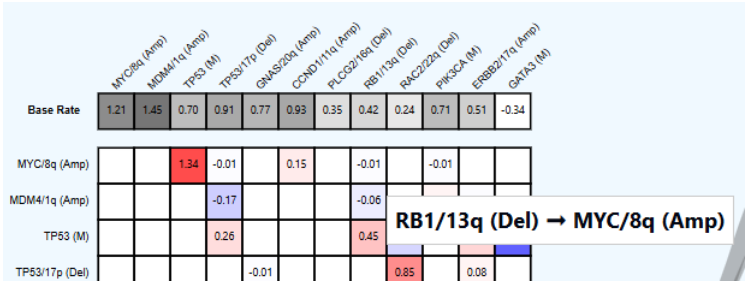


Figure 4.14: Theta Matrix and tooltip showing the multiplicative effect of "RB1/13q" on "MYC/8q", which equals 0.45.

5 Discussion

This section critically reflects on methodological and technical aspects of the project implementation and design. It highlights the achievements and limitations inherent to the web application, with particular emphasis on the risk of misinterpretation arising from unguided exploration of the History Trees and the Theta Matrix. Furthermore, it discusses the limitations of depending on cross-sectional data and finally outlines potential improvements and future development steps.

5.1 Achievements

The development process followed clearly defined goals, allowing for steady and structured progress. An initial design draft (5.1), outlining the envisioned features and layout, served as a guiding reference throughout the project.

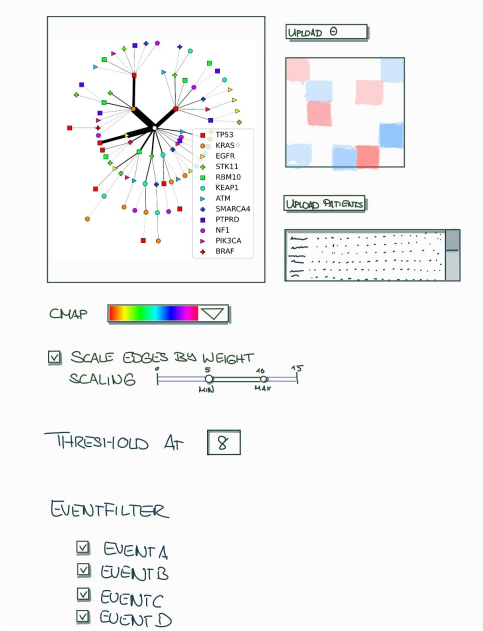


Figure 5.1: Initial drawing of the MHN Web Application at the beginning of the development process.

Since the majority of the envisioned features, encompassing threshold, event filtering, color scheme, and edge scaling, were implemented as intended, the final web application closely reflects the original draft. The core functionality and objective of the project—the upload and visualization of MHN-inferred tumor progression paths in the

form of History Trees—was successfully achieved. Further, the integration and upload of the Theta Matrix were also completed.

Beyond the planned features in the draft, several additional enhancements were realized to expand the flexibility in the customization process. These include:

- Displaying a tooltip in the History Tree with path overview and the patient count
- Integrating a sidebar and toggle function for the Theta
- Highlighting paths by clicking on them or by selecting an event in the sidebar
- Adjusting node styling (only stroke when no children)
- Adding `Expand All` and `Reset` buttons
- Including a user guide/information section
- Implementing a toggle effect for the Theta Matrix with an animation to prevent visual overload

While this variety of features already offers valuable support and extensive customization options for users, some functional aspects have not been realized or require improvement.

5.2 Open Points in Development

One of these aspects is the automatic generation of the History Tree through uploading the Theta Matrix and patient data. To obtain the required JSON file with the inferred orders, two steps involving the MHN algorithm are needed. In the first step, MHN estimates the Theta Matrix. In the second step, the JSON file is generated. To enhance the workflow and lighten the workload of bioinformaticians, the web application should allow users to upload tumor data, as well as the corresponding Theta Matrix, and automatically infer the most likely tumor progression paths for the dataset. For this, implementing a Python backend is needed, which can be approached by, e.g., using the node module `child_process` (`https://nodejs.org/api/child_process.html`).

Furthermore, the application can currently only be accessed by downloading and running it locally on the computer. Although instructions for installation are provided in the README file in the GitHub repository, users must first understand and execute several technical steps to start the application, which creates additional barriers to using the app. Therefore, it is essential to make the web application publicly available online (*deployment*).

Another notable shortcoming in the web app is the lack of detailed error feedback. Currently, users receive little to no information about the cause of an error, let alone how to resolve it independently. For example, in the development process, repeated issues were encountered during dataset uploads, where the input file was not displayed correctly. Yet, when a user encounters the same error, there is no error message explaining whether the problem stems from the data structure or an internal application failure. For users with limited or no experience with IT, such errors might lead to confusion and frustration. Therefore, testing for errors and implementing error handling in a user-friendly way is required.

However, beyond the technical aspects, it is also essential to evaluate the constraints of the current application in terms of its informative value.

5.3 Limitations in the Informative Value

5.3.1 Risk of Misinterpretation

The web application serves as a scientific tool for exploring and interpreting complex MHN data independently. Nevertheless, for a correct interpretation of the information contained in the Theta Matrix and the History Tree, a solid understanding of the underlying mathematical principles is necessary.

For instance, the multiplicative effects in the Theta Matrix might be mistakenly interpreted as indicating direct causal relationships between genetic events. In reality, these values reflect the promoting or inhibiting effects on the base rate of the influenced event, rather than equaling the probability of an event occurring. In general, the Theta Matrix can be challenging to comprehend as a whole and utilize effectively as a supplement to the History Tree without a sufficient theoretical background.

Further, the parameter "count", which serves as the basis for the edge scaling and threshold, is prone to misinterpretation. While it might be assumed that "count" indicates the overall frequency of a genetic event in the dataset, it rather represents the number of patients who share the same genetic event history up to a particular event.

Although explanations have been provided in the user guide and on the web page, there remains a risk that, without a clear understanding of the mathematical background of the parameters, users may unintentionally misinterpret their findings, potentially leading to incorrect biological conclusions.

5.3.2 Focusing on the Most Likely Path

A further constraint of the current project implementation is that only the most likely progression path for each tumor can be visualized. While a compact dataset simplifies the workflow and keeps the History Tree view clear, it also excludes alternative paths that may contain valuable biological insights and could be equally relevant in unraveling patterns in the order of genetic events.

5.3.3 Data Derived from Biopsies

MHN is a powerful and flexible framework for modeling tumor progression, as it captures relevant dependencies between genetic events. For instance, in the work of Schill et al. in 2019, the inferred progression paths suggested an association between *IDH1 (M)* and *TP53 (M)* mutations in a glioblastoma dataset [13]. This dependency was observed and supported by independent data [35], demonstrating the model's potential to uncover biologically meaningful interactions.

However, a common limitation among CPMs is their reliance on cross-sectional data derived from biopsies. MHN infers the most likely sequence of genetic events from static snapshots of tumors taken from different patients at unknown and varying stages of

tumor development. As a result, the reconstructed progression paths are statistical estimates rather than direct observations of tumor evolution [13].

Ultimately, the quality of information obtained by the application depends on a clear communication of what the data represent and where their limitations lie. Users should be aware that the visualizations are based on statistical models and cross-sectional data, rather than direct observations of tumor evolution, when interpreting the data in the context of the application.

Taking these listed constraints into account, the final section provides an outlook on how the web application can be improved to ensure technical accessibility and science communication.

5.4 Outlook

There are several ways to improve the usability of the app, reduce technical barriers, and increase the informative value.

Essential next steps for improving the usability of the web application include implementing error feedback, deploying the application online, and automating the generation of MHN History Trees. To reduce the risk of misinterpretation, dynamic explanatory elements could be integrated into the user interface, guiding users through the visualization and its results. As a quick and effective measure, the user guide could be extended with a section that explains the mathematical foundations, model limitations, and common mistakes in interpretations using simplified terms, tailored to users without a strong statistical and/or mathematical background.

Another meaningful step could be considering the option of visualizing not only the most likely path but also displaying alternative paths with a slightly lower likelihood. As tumors can evolve along multiple possible trajectories, adding alternative progression paths or perhaps even future progression paths to the dataset could support clinicians in evaluating different scenarios of tumor development and lead to making better-informed decisions.

With these improvements and a continuous integration into clinical workflows, this web application has the potential to facilitate interdisciplinary collaborations between scientists and clinicians, advancing our understanding of cancer evolution together.

6 Conclusion

In this work, an interactive web application was developed to enable interactive data exploration and analysis, thereby fostering interdisciplinary collaboration among bioinformaticians, clinicians, and researchers. The tool enables users to visualize tumor progression paths inferred by the Mutual Hazard Networks (MHN) algorithm, as well as the corresponding Theta Matrix. A variety of features support users in independently analyzing and interpreting the visualizations.

These interactive features, including complete tree expansion, path highlighting, color customization, setting a threshold, event filtering, and edge scaling, were developed using modern web technologies such as Next.js, D3.js, React, and TypeScript. While the combination of these tools represented significant obstacles at times due to compatibility challenges or the lack of tutorials, overall, the development process was successful.

Further necessary development steps include integrating an automated generation of History Trees based on uploaded patient profiles and the Theta Matrix, improving error handling, and deploying the application publicly to enhance both accessibility and usability simultaneously.

Yet, a significant risk of misinterpretation remains due to confounding factors in cross-sectional biopsy data and the lack of an accessible explanation of the mathematical foundations underlying the MHN model. Addressing this limitation is crucial to ensure that users without a mathematical background can accurately interpret the results.

In summary, by lowering the technical barrier to interacting with bioinformatic results, the application represents a first step toward making CPMs more accessible and practically valuable for clinicians and oncologists. Therefore, this thesis bridges the gap between computational modeling and practical application, offering both a technically solid and scientifically meaningful contribution to the interpretation of tumor progression data.

7 References

[1] Rebecca L. Siegel et al. "Cancer statistics, 2025". In: *CA: A Cancer Journal for Clinicians* 75.1 (2025), pp. 10–45. DOI: https://doi.org/10.3322/caac.21871. URL: https://acsjournals.onlinelibrary.wiley.com/doi/abs/10.3322/caac.21871.

[2] Moritz Gerstung et al. "The evolutionary history of 2,658 cancers". In: *Nature* 578.7793 (2020), pp. 122–128. URL: https://doi.org/10.1038/s41586-019-1907-7.

[3] Shaosen Zhang et al. "Tumor initiation and early tumorigenesis: molecular mechanisms and interventional targets". In: *Signal Transduction and Targeted Therapy* 9.1 (2024), p. 149. URL: https://doi.org/10.1038/s41392-024-01848-7.

[4] Michael R. Stratton, Peter J. Campbell, and P. Andrew Futreal. "The cancer genome". In: *Nature* 458.7239 (2009), pp. 719–724. URL: https://doi.org/10.1038/nature07943.

[5] Jawad Fares et al. "Molecular principles of metastasis: a hallmark of cancer revisited". In: *Signal transduction and targeted therapy* 5.1 (2020), p. 28. URL: https://doi.org/10.1038/s41392-020-0134-x.

[6] David W. Kufe et al., eds. *Holland-Frei Cancer Medicine 6*. Vol. 6. BC Decker, 2003. URL: https://www-ncbi-nlm-nih-gov.translate.goog/books/NBK12354/?_x_tr_sl=en&_x_tr_tl=de&_x_tr_hl=de&_x_tr_pto=rq.

[7] Douglas Hanahan and Robert A Weinberg. "The Hallmarks of Cancer". In: *Cell* 100.1 (2000), pp. 57–70. URL: https://doi.org/10.1016/S0092-8674(00)81683-9.

[8] Douglas Hanahan and Robert A. Weinberg. "Hallmarks of Cancer: The Next Generation". In: *Cell* 144.5 (2011). URL: https://doi.org/10.1016/j.cell.2011.02.013.

[9] Douglas Hanahan. "Hallmarks Of Cancer: New Dimensions". In: *Cancer Discovery* 12.1 (2022), pp. 31–46. URL: https://doi.org/10.1158/2159-8290.CD-21-1059.

[10] A. Weston and C. C. Harris. "Multistage Carcinogenesis". In: *Holland-Frei Cancer Medicine*. Ed. by David W. Kufe et al. 6th ed. Hamilton (ON): BC Decker, 2003. URL: https://www.ncbi.nlm.nih.gov/books/NBK13982/.

[11] Daria Ostroverkhova, Teresa M. Przytycka, and Anna R. Panchenko. "Cancer driver mutations: predictions and reality". In: *Trends in Molecular Medicine* 29.7 (2023), pp. 554–566. URL: https://www.sciencedirect.com/science/article/pii/S1471491423000679.

[12] Ramon Diaz-Uriarte. "Cancer progression models and fitness landscapes: a many-to-many relationship". In: *Bioinformatics* 34.5 (2017), pp. 836–844. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btx663.

[13] Rudolf Schill et al. "Modelling cancer progression using Mutual Hazard Networks". In: *Bioinformatics* 36.1 (2019). URL: https://doi.org/10.1093/bioinformatics/btz513.

[14] Rudolf Schill et al. “Overcoming Observation Bias for Cancer Progression Modeling”. In: *bioRxiv* (2023). DOI: 10.1101/2023.12.03.569824. URL: https://www.biorxiv.org/content/early/2023/12/05/2023.12.03.569824.

[15] GeeksforGeeks. *Web Development - GeeksforGeeks*. Accessed July 18, 2025. 2023. URL: https://www.geeksforgeeks.org/web-tech/web-development/.

[16] GeeksforGeeks. *Difference between Web Application and Website*. Accessed July 22, 2025. 2021. URL: https://www.geeksforgeeks.org/websites-apps/difference-between-web-application-and-website/.

[17] ComputerScience.org Editors. *Frontend vs. Backend: What's the Difference?* Accessed July 8, 2025. 2023. URL: https://www.computerscience.org/bootcamps/resources/frontend-vs-backend/.

[18] Vercel. *Rendering: Client and Server Components*. Accessed July 8, 2025. 2024. URL: https://nextjs.org/learn/react-foundations/rendering-ui.

[19] simpleclub. *Client-Server-Modell einfach erklärt*. Accessed July 8, 2025. 2023. URL: https://simpleclub.com/lessons/informatik-client-server-modell.

[20] freeCodeCamp. *The Difference Between a Framework and a Library*. Accessed July 19, 2025. 2021. URL: https://www.freecodecamp.org/news/the-difference-between-a-framework-and-a-library-bd133054023f/.

[21] shadcn and Vercel. *Introduction – Shadcn UI: Re-usable components built using Radix UI and Tailwind CSS*. Accessed July 9, 2025. 2025. URL: https://ui.shadcn.com/docs.

[22] Vercel. *Next.js 13*. Accessed July 9, 2025. 2022. URL: https://nextjs.org/blog/next-13.

[23] shadcn. *Slider – shadcn/ui*. Accessed July 9, 2025. 2023. URL: https://ui.shadcn.com/docs/components/slider.

[24] React. *State – React*. Accessed July 9, 2025. 2024. URL: https://react.dev/learn/state-a-components-memory.

[25] React Documentation. *Passing Data Deeply with Context – React Docs*. Accessed July 10, 2025. 2023. URL: https://react.dev/learn/passing-data-deeply-with-context.

[26] Mike Bostock. *Collapsible Tree*. Accessed July 10, 2025. 2023. URL: https://observablehq.com/@d3/collapsible-tree.

[27] Yan Holtz. *Heatmap with tooltip in d3.js*. Accessed July 9, 2025. 2024. URL: https://d3-graph-gallery.com/graph/heatmap_tooltip.html.

[28] Vercel. *Next.js Learn Tutorial*. Accessed July 10, 2025. 2024. URL: https://nextjs.org/learn?utm_source=next-site&utm_medium=homepage-cta&utm_campaign=home.

[29] Mike Bostock. *D3.js — Getting Started*. Accessed July 10, 2025. 2024. URL: https://d3js.org/getting-started.

[30] freeCodeCamp. *Data Visualization with D3*. Accessed July 10, 2025. 2024. URL: https://www.freecodecamp.org/learn/data-visualization/.

[31] Reddit. *Good docs/samples for using D3 in TypeScript?* Accessed July 10, 2025. 2022. URL: https://www.reddit.com/r/d3js/comments/wpay6u/good_docssamples_for_using_d3_in_typescript/.

[32] bkrem. *react-d3-tree*. Accessed July 7, 2025. 2024. URL: https://www.npmjs.com/package/react-d3-tree.

[33] Robert Koch Institute and Association of Population-based Cancer Registries in Germany, eds. *Cancer in Germany 2019/2020*. 14th ed. Berlin: Robert Koch In-

stitute, 2024. URL: https://www.krebsdaten.de/Krebs/DE/Content/Publikationen/Krebs_in_Deutschland/krebs_in_deutschland_node.html.

[34] Statistisches Bundesamt (Destatis). *Leading causes of death among females in Germany*. Accessed July 13, 2025. 2025. URL: https://www.destatis.de/EN/Themes/Society-Environment/Health/_Graphic/_Interactive/causes-death-leading-female.html.

[35] T. Watanabe et al. "IDH1 mutations are early events in the development of astrocytomas and oligodendrogliomas". In: *American Journal of Pathology* 174.4 (2009), pp. 1149–1153. DOI: 10.2353/ajpath.2009.080958.

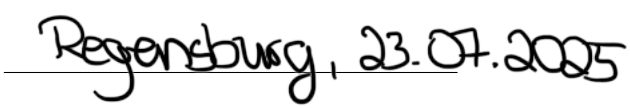
Declaration of Authorship

I hereby certify that I have written this thesis independently and have not used any sources or aids other than those specified. I have correctly cited all passages that I have quoted or paraphrased. I have not yet submitted the thesis in the same or a similar form for any other course or examination authority.

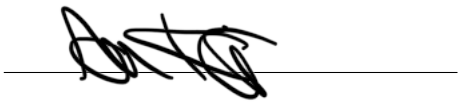
I have used digital tools, including artificial intelligence, in the following ways:

- ChatGPT was used to support the writing process (e.g., improving reading flow and structural guidance) and to assist in resolving technical issues, such as TypeScript errors
- Grammarly was used to check and improve language quality
- Canva Pro was used to design illustrations

The content and final responsibility for this thesis remain entirely my own.



Place, Date



Antonia Kaspar

Acknowledgements

I want to express my sincere gratitude to my supervisor, Yanren Linda Hu, for her exceptional support, insightful guidance, and unwavering patience throughout the development and writing of this thesis. Her expertise and encouragement were invaluable at every stage of the project. My appreciation also goes to Michael Huttner, whose input, especially regarding Next.js, significantly simplified the development process. Furthermore, I want to thank Prof. Dr. Rainer Spang from the Chair of Statistical Bioinformatics for welcoming me into his research group and giving me the opportunity to write my Master's thesis within the framework of this project. I would also like to sincerely thank Prof. Dr. Till Rudack from the Chair of Structural Bioinformatics for kindly agreeing to serve as the second referee of this thesis.

List of Figures

Code Snippets

A CollaTree.tsx

The complete source code is accessible on GitHub:
https://github.com/Antonia-gthb/historytree

```
1
2  import * as d3 from "d3";
3  import { useRef, useEffect } from "react";
4  import useGlobalContext from "@/app/Context";
5  import { cSchemes } from "../colorSchemes";
6
7  type MyNode = d3.HierarchyPointNode<TreeNode> & {
8    x0: number;
9    y0: number;
10   _children?: MyNode[];
11   color?: string;
12 }
13
14 export type TreeNode = {
15   name: string;
16   children?: TreeNode[];
17   originalName?: string, // ? means: property does not have to be present in the
         JSON data set and is optional!
18   value?: number;
19   count?: number;
20   _children?: TreeNode[]; // _children for storing the collapsed nodes
21   color?: string;
22 };
23
24 interface CollaTreeProps {
25   treedata: TreeNode;
26   width: number;
27 }
28
29 {/* The following code creates the SVG and contains the visualization logic for the
       MHN History Tree */ }
30
31 export default function CollaTree({
32   treedata,
33   width, //width from SVG
34 }: CollaTreeProps) {
35   const {
36     selectedSchemeName,
37     isExpanded,
38     threshold,
```

```
    scalingFactor,
    selectedMutations,
    highlightMutation,
    geneticEventsName,
    setGeneticEventsName,
    setSelectedMutations,
    setHighlightMutation,
  } = useGlobalContext();

  const svgRef = useRef<SVGSVGElement | null>(null);
  const colorScaleRef = useRef<d3.ScaleOrdinal<string, string, never> | null>(null);
  const nodeSelectionRef = useRef<d3.Selection<SVGGElement, MyNode, SVGGElement,
      unknown> | null>(null);
  const selectedLinksRef = useRef<d3.Selection<SVGPathElement, d3.HierarchyLink<
      MyNode>, SVGGElement, unknown> | null>(null);
  const mutationNamesRef = useRef<string[] | null>(null);
  const factorRef = useRef<number>(scalingFactor);
  const maxCountRef = useRef<number>(1);
  const highlightedLinkRef = useRef<d3.HierarchyLink<MyNode> | null>(null);
  const rootRef = useRef<MyNode | null>(null);
  const highlightRef = useRef<string>(highlightMutation);
  const finalOrderRef = useRef<string[]>([]);



  //the refs above are used to save data, so that the tree does not re-render every
      time a prop changes //the code for the useEffects is found at the end

  //function to name each node individually; otherwise, there were problems with the
       animation
  function numberNodes(node: TreeNode, parentName = "", mutationNames: string[] =
      []) {

    const orig = node.originalName ?? node.name;
    node.originalName = orig;
    mutationNames.push(orig);

    if (parentName) {
      node.name = `${parentName}_${orig}`;
    }

    if (node.children) {
      node.children.forEach((child, index) =>
        numberNodes(child, `${node.name}_${index + 1}`, mutationNames)
      );
    }
  }

  //function for filtering the data for the event filter. Only the mutations that
      are selected are filtered out.
  function filterTreeData(tree: TreeNode, selectedMutations: string[]): TreeNode |
      null {
    const isActive = (name: string | undefined) => {
      if (!name) return false;
      return selectedMutations.includes(name);
    };

    const originalName = tree.originalName || tree.name;
```

```
    if (!isActive(originalName)) {
      return null;
    }

    const filteredChildren = tree.children
      ?.map(child => filterTreeData(child, selectedMutations))
      .filter(child => child !== null) as TreeNode[] | undefined;

    return {
      ...tree,
      children: filteredChildren?.length ? filteredChildren : undefined,
    };
  }


  //function for filtering when a threshold is set
  function filterByThreshold(node: TreeNode, thr: number): TreeNode | null {
    if ((node.count ?? 0) < thr) return null;

    const children = node.children
      ?.map(child => filterByThreshold(child, thr))
      .filter((c): c is TreeNode => c !== null);

    return {
      ...node,
      children: children && children.length > 0 ? children : undefined,
    };
  }

  //function to save all of the names of the genetic events in an array
  function collectAllNames(node: TreeNode, out: string[] = []): string[] {
    const name = node.originalName ?? node.name;
    if (name !== "root") out.push(name);
    if (node.children) {
      node.children.forEach(child => collectAllNames(child, out));
    }
    const allNames = [...new Set(out)]
    return allNames;
  }

  //useEffect for the MHN Tree

  useEffect(() => {
    if (!svgRef.current) return;

    const margin = { top: 20, right: 20, bottom: 20, left: 40 }; //the margin around
        the tree
    const dx = 35; //distance between the nodes

    // if-loop to update mutation names
    if (!mutationNamesRef.current) {
      const mutationNames: string[] = [];
      numberNodes(treedata, "", mutationNames); // collect mutation names
      mutationNamesRef.current = mutationNames;
      setSelectedMutations(mutationNames);
    }

    //if the array of selectedMutations equals zero and the geneticEventsName array
```

```
        is bigger than zero, update the selectedMutations array to the
        mutationNamesRef
147     if (selectedMutations.length === 0 && geneticEventsName.length > 0) {
148       setSelectedMutations(mutationNamesRef.current);
149       return;
150     }
151
152     //here the filtered data is defined
153     const filteredData = selectedMutations && selectedMutations.length > 0
154       ? filterTreeData(treedata, selectedMutations) ?? { name: "No mutations
            selected", children: [] }
155       : treedata;
156     const filteredWithThreshold = threshold
157       ? filterByThreshold(filteredData, threshold) ?? { name: "No genetic events
            with this threshold available", children: [] }
158       : filteredData;
159
160     const allGeneticEvents = collectAllNames(treedata);
161     setGeneticEventsName(allGeneticEvents)
162
163     //this data is then used to create the root and the tree
164     const root = d3.hierarchy(filteredWithThreshold) as MyNode;
165     root.sum(d => d.count || 0);
166     rootRef.current = root;
167     const dy = (width - margin.right - margin.left) / (1 + root.height); //
        determines the position for the node texts
168     const tree = d3.tree<TreeNode>().nodeSize([dx, dy]);
169     const diagonal = d3.linkHorizontal<MyNode, MyNode>().x(d => d.y).y(d => d.x); //
        important for the edges!
170
171     d3.select(svgRef.current).selectAll("*").remove(); //SVG is removed and created
        again when re-rendering the tree
172
173     //creating the SVG
174     const svg = d3.select<SVGSVGElement, unknown>(svgRef.current)
175       .attr("viewBox", [-margin.left, -margin.top, width, dx])
176       .attr("width", "100%")
177       .style("height", "auto")
178       .style("font", "12px sans-serif") // adapt font size
179       .style("user-select", "none")
180       .attr("id", "histree-chart")
181
182     //creates link group and styles them
183     const gLink = svg.append("g")
184       .attr("fill", "none")
185       .attr("stroke-opacity", 0.4)
186       .attr("stroke", "555")
187       .attr("stroke-width", "1.5")
188
189     //creates node group
190     const gNode = svg.append("g")
191       .attr("cursor", "pointer")
192       .attr("pointer-events", "all");
193
194
195     //function for highlighting the paths, calculation of the ancestors of the node
        so that the entire path up to the node is colored purple
196     function highlightPath(link: d3.HierarchyLink<MyNode> | null, active: boolean) {
```

```
197       gLink.selectAll<SVGPathElement, d3.HierarchyLink<MyNode>>("path")
198         .attr("stroke", d => {
199           if (!active || !link) return "#555";
200           const ancestors = new Set(link.target.ancestors());
201           return ancestors.has(d.target) ? "#372aac" : "#555";
202         })
203         .attr("stroke-opacity", d => {
204           if (!active || !link) return 0.4;
205           const ancestors = new Set(link.target.ancestors());
206           return ancestors.has(d.target) ? 1 : 0.4;
207         });
208     }
209
210     {/*UPDATE FUNCTION */ }
211
212     // this function is called again if the dependencies treedata, isExpanded,
          selectedMutations, or threshold change
213     //then the tree is re-rendered
214     //the function is also called at node expansion or collapse without a re-render
215
216     function update(source: MyNode) {
217       const duration = 1200; //animation length
218       const nodes = root.descendants().reverse(); //fetches the nodes, upside down,
            so that we start at the parent nodes
219       const links = root.links() as unknown as d3.HierarchyLink<MyNode>[]; //array
            of all links
220
221       tree(root); //creates the layout
222
223       const firstOrders = root
224         .descendants()
225         .filter(d => d.depth === 1)
226         .map(d => d.data.originalName)
227         .filter((n): n is string => typeof n === "string" && n !== "root");
228
229       const finalOrder = [
230         ...new Set([
231           ...firstOrders,
232           ...allGeneticEvents
233         ])
234       ];
235
236       let left = root;
237       let right = root;
238
239
240       //which node is furthest up/down? The height is then calculated from this. x
            and y are reversed, as the tree runs horizontally
241       root.eachBefore(node => {
242         if (node.x !== undefined && node.x < (left.x ?? Infinity)) left = node;
243         if (node.x !== undefined && node.x > (right.x ?? -Infinity)) right = node;
244       });
245
246       const height = right.x - left.x + margin.top + margin.bottom; //calculates the
             height of the tree
247
248       //update nodes by calling the group, selecting all nodes, and connecting them
            with the data (name and depth, which is important for the symbols)
```

```
249       const node = gNode.selectAll<SVGGElement, MyNode>("g").data(nodes, d => d.data
              .name + "-" + d.depth);
250
251       //defining the transition
252       const transition: d3.Transition<SVGSVGElement, unknown, null, undefined> = svg
              .transition()
253         .duration(duration)
254         .attr("height", height)
255         .attr("viewBox", `${-margin.left} ${left.x - margin.top} ${width} ${height
                }`) //dynamic calculation of the view box
256
257
258       //calculating new nodes
259       const nodeEnter = node.enter().append<SVGGElement>("g")
260         .attr("transform", () => `translate(${source.y0},${source.x0})`)
261         .attr("fill-opacity", 0)
262         .attr("stroke-opacity", 0)
263         .on("click", (_, d) => {
264           if (d._children || d.children) {
265
266             root.eachBefore((node) => {
267               node.x0 = node.x;
268               node.y0 = node.y;
269             })
270
271             d.children = d.children ? undefined : d._children;
272             update(d);   //updates nodes
273           }
274         });
275
276       {/* COLOR SCHEME */ }
277       const schemeFn = cSchemes.find(s => s.name === selectedSchemeName)!.fn;
278
279       finalOrderRef.current = finalOrder; // the correct order of the events leading
              to the spectrum implemented by d3 chromatic on the first render
280
281       colorScaleRef.current = d3.scaleOrdinal<string, string>() //assign colors to
              events names
282         .domain(finalOrder)
283         .range(d3.quantize(schemeFn, allGeneticEvents.length));
284
285       //Adding symbols
286       const symbols = [d3.symbolCircle, d3.symbolSquare, d3.symbolTriangle];
287       const shapeScale = d3.scaleOrdinal<string, d3.SymbolType>()
288         .domain(finalOrder)
289         .range(symbols);
290
291       nodeEnter.append("path")
292         .attr("d", d => {
293           if (d.depth === 0) return "";
294           const type = shapeScale(d.data.originalName!);
295           return d3.symbol().type(type).size(100)();
296         })
297         .attr("fill", d => {
298           const isLeaf = !d.children && !d._children;
299           return isLeaf
300             ? "none"
301             : colorScaleRef.current!(d.data.originalName!);       // fetches the
```

```
            specified color, if no children are there, no color is assigned (
            leaves only have the stroke)
      })
      .attr("stroke", d => colorScaleRef.current!(d.data.originalName!)) // Stroke
          also from d.data.color


    const textEnter = nodeEnter.append("text") //determines the text position
        dynamically and keeps the root at its position at all times
      .attr("text-anchor", d => {
        if (d.depth === 0) return "start";
        return Array.isArray(d.children) && d.children.length > 0 ? "middle" : "
            start";
      })
      .attr("x", d => {
        if (d.depth === 0) return -25;
        return Array.isArray(d.children) && d.children.length > 0 ? 0 : 12;
      })
      .attr("y", d => {
        if (d.depth === 0) return 4;
        return Array.isArray(d.children) && d.children.length > 0 ? -12 : 4;
      });

    textEnter.each(function (d) {
      const full = d.data.originalName || d.data.name;
      const [main, rest] = full.split(" (");
      const text = d3.select(this);

      text.append("tspan")
        .text(main)
        .style("font-size", "11px");

      if (rest) {
        text.append("tspan")
          .text(` (${rest})`)
          .style("font-size", "8px");
      }
    });

    //everything is merged here and linked to the transition
    nodeEnter.merge(node).transition().duration(duration)
      .attr("transform", d => `translate(${d.y},${d.x})`)
      .attr("fill-opacity", 1)
      .attr("stroke-opacity", 1);


    //old nodes disappear
    node.exit().transition(transition as any).remove()
      .attr("transform", () => `translate(${source.y},${source.x})`)
      .attr("fill-opacity", 0)
      .attr("stroke-opacity", 0);

    nodeSelectionRef.current = nodeEnter.merge(node);

    // updating links
    const link = gLink.selectAll<SVGPathElement, d3.HierarchyLink<MyNode>>("path")
      .data(links, d => d.target.data.name + "-" + d.target.depth);

```

```
355       //create new links
356       const linkEnter = link.enter().append("path")
357         .attr("cursor", "pointer")
358         .attr("fill", "none")
359         .attr("stroke", "#555")
360         .attr("stroke-opacity", 0.4)
361         .attr("d", d => {
362           const o = {
363             x: (d.source as any).x0 ?? d.source.x,
364             y: (d.source as any).y0 ?? d.source.y
365           };
366           return (diagonal as any)({ source: o, target: o });
367         });
368
369       //tooltip for clicking to highlight
370       linkEnter.append("title")
371         .text("Highlight this path");
372
373       //merging links
374       const linkAll = linkEnter.merge(link);
375
376       //a link can be highlighted in purple by clicking on it
377       linkAll.on("click", (_, d) => {
378         if (highlightedLinkRef.current === d) {
379           highlightedLinkRef.current = null;
380           highlightPath(null, false);
381           setHighlightMutation("");
382         } else {
383           highlightedLinkRef.current = d;
384           highlightPath(d, true);
385           setHighlightMutation("");
386         }
387       });
388
389       {/* SCALING */ }
390
391       //code for mapping the stroke width linearly to the count
392       const counts = root.descendants().filter(d => d.data.originalName !== root.
              data.originalName).map(d => d.data.count ?? 0); //all counts except for
              root
393       const maxCount = counts.length > 0 ? Math.max(...counts) : 1; //which count is
               the highest?
394       const defaultWidth = 1.5; //normally always 1.5 px as default size, or when
              scaling is off
395       maxCountRef.current = maxCount;
396       const defaultMax = 10 + factorRef.current; //the width must not be greater
397       const baseScale = factorRef.current === 0
398         ? () => defaultWidth
399         : d3.scaleLinear<number>()
400           .domain([0, maxCount])
401           .range([factorRef.current, defaultMax]); //stroke width is distributed over
                   the current factor that we get from the page and the maximum width,
                  depending on count
402
403       //link transition
404       linkAll.transition(transition as any)
405         .attr("d", diagonal as any)
406         .attr("stroke-width", d => baseScale(d.target.data.count as any ?? 0));
```

```
407
408       //old links disappear
409       link.exit().transition(transition as any).remove()
410         .attr("d", () => {
411           const o = { x: source.x, y: source.y };
412           return (diagonal as any)({ source: o, target: o });
413         })
414
415       selectedLinksRef.current = linkAll;
416
417       //tooltip for path overview
418       nodeEnter.append("title")
419         .text(d => {
420           const count = d.data.count ?? 0;
421           const ancestorNames = d.ancestors()
422             .reverse() // start at root
423             .map(a => a.data.originalName || a.data.name)
424             .filter(name => name !== "root")
425             .join("    ");
426
427           return `${ancestorNames} | Patient Count: ${count}`;
428         });
429
430       //save old
431       root.eachBefore(d => {
432         d.x0 = d.x;
433         d.y0 = d.y;
434       });
435     }
436
437     //initialize tree
438     root.x0 = 0;
439     root.y0 = 0;
440     if (isExpanded) {
441       root.descendants().forEach(d => {
442         d._children = d.children;
443         d.children = d._children;
444       });
445     } else {
446       root.descendants().forEach(d => {
447         d._children = d.children;
448         if (d.depth && d.data.name.length !== 1) d.children = undefined; //on the
                first render, only the first depth will be displayed
449       });
450     }
451
452     update(root);
453
454   }, [treedata, selectedMutations, threshold, isExpanded]);
455
456
457   //in the following lines, there are several useEffects so that the tree does not
        have to be re-rendered each time a dependency changes
458
459   {/* USEEFFECT COLOR SCHEME */ }
460
461   useEffect(() => {
462     const scale = colorScaleRef.current;
```

```
    const node = nodeSelectionRef.current!
    const domain = finalOrderRef.current;

    if (!scale || !node) return;

    const fn = cSchemes.find(s => s.name === selectedSchemeName)!.fn;
    // generate as many colors as in the domain length
    const newColors = d3.quantize(fn, domain.length);

    scale.range(newColors);

    node
      .selectAll<SVGPathElement, MyNode>("path")
      .transition()
      .duration(600)
      .attr("fill", d =>
        !d.children && !d._children
          ? "none"
          : scale(d.data.originalName || d.data.name)
      )
      .attr("stroke", d =>
        scale(d.data.originalName || d.data.name)
      );
  }, [selectedSchemeName]);


  {/* USEEFFECT HIGHLIGHT MUTATION*/ }

  useEffect(() => {
    highlightRef.current = highlightMutation;
    const selLinks = selectedLinksRef.current;
    const root = rootRef.current;

    if (!selLinks || !root || !highlightMutation) return;

    const matches = root
      .descendants()
      .filter(d => (d.data.originalName || d.data.name) === highlightMutation);

    const highlightSet = new Set<MyNode>();

    matches.forEach((m) => {
      m.ancestors().forEach(a => highlightSet.add(a as MyNode));
      highlightSet.add(m as MyNode);
      m.descendants().forEach(d => highlightSet.add(d as MyNode));
    });

    selLinks
      .transition()
      .duration(100)
      .attr("stroke", (linkData) => {
        const source = linkData.source as any;
        const target = linkData.target as any;
        return highlightSet.has(source) && highlightSet.has(target)
          ? "#372aac"
          : "#555";
      })
      .attr("stroke-opacity", (linkData) => {
```

```
        const source = linkData.source as any;
        const target = linkData.target as any;
        return highlightSet.has(source) && highlightSet.has(target)
          ? 1
          : 0.3;
      });
  }, [highlightMutation]);


  {/* USEEFFECT SCALING */ }
  useEffect(() => {
    if (!selectedLinksRef.current) return;

    factorRef.current = scalingFactor;

    const defaultMax = 10 + factorRef.current;
    const defaultWidth = 1.5;
    const baseScale = factorRef.current === 0
      ? () => defaultWidth
      : d3.scaleLinear<number>()
        .domain([0, maxCountRef.current])
        .range([factorRef.current, defaultMax]);

    selectedLinksRef.current
      .attr("stroke-width", d => baseScale(d.target.data.count as any ?? 0));
  }, [scalingFactor]);

  return <svg ref={svgRef}></svg>;
}
```