HOCHSCHULE RUHR WEST
UNIVERSITY OF APPLIED SCIENCES

Hochschule Ruhr West | Gesundheits- und Medizintechnologien | 10011512

Evaluierung von Graphbibliotheken zum Visualisieren eines Wissensgraphen

Bachelorarbeit

Alperen Dagli
10011512

Mülheim an der Ruhr, Juli 2022

Erstprüfer/in: Prof. Dr. Fatih Gedikli

Zweitprüfer/in: Noah Janzen

Kurzfassung

In dieser Arbeit wird die Thematik Datenvisualisierung von Wissensgraphen behandelt. Durch den massiven Datenbestand in der heutigen Zeit und der immer stärker werdenden Digitalisierung ist die Notwendigkeit gestiegen, aus Daten effizient Wissen generieren zu können. Folglich wurde repräsentativ eine Graphbibliothek ausgewählt, um die Daten zu visualisieren. Umgesetzt wurde die Programmierung durch ein JavaScript Framework. Die Daten einer Wissensdatenbank wurden vom Betreuer bereitgestellt. Das übergeordnete Ziel der Arbeit ist eine verbesserte Darstellung von Wissensdaten in Form eines Wissensgraphen. Sinn und Zweck dahinter ist es, Beziehungen zwischen Daten verbessert erkennen zu können und gleichzeitig Informationen zu gewinnen, die auf den ersten Blick nicht sofort erkennbar sind.

Abstract

The following thesis deals with the topic of data visualization regarding knowledge graphs. Because of the massive amount of data and expanding digitalization, the world offers nowadays, the need to efficiently harvest knowledge out of it rises. Therefore, one chose a library for generating knowledge graphs to visualize given data. A knowledge database with proper data was given in advance. Overall goal of this thesis is to properly visualize the knowledge data into a knowledge graph and therefore to gain insights about relationships between data. Furthermore, it is also in interest to get information about data that could be inferred and without a proper visualization can´t be seen on the first look.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

IEE	Institute of Electrical and Electronics Engineers
AI	Artificial Intelligence
MCTS	Monte Carlo Tree Search
FCP	First Contentful Paint
SI	Speed Index
LCP	Largest Contentful Paint
TTI	Time to Interactive
TBT	Total Blocking Time
CLS	Cumulative Layout Shift

1 Einleitung

Mit einer immer größer werdenden Anzahl an Daten, die die Menschheit Tag für Tag generiert, leiden sowohl Überschaubarkeit bezüglich der Daten als auch der eigentliche Zweck dessen, aus diesen Daten Wissen und Informationen zu generieren. Statistiken zufolge, wird weltweit bis 2025 mehr als 175 Zettabytes an Daten auf der Erde vorhanden sein [1]. Diese enorme Menge an Daten ist jedoch wertlos (nicht so wertend bitte), sofern wir (wir schonmal gar nicht, es heißt man) keine Informationen aus diesen extrahieren können [2]. In der folgenden Abschlussarbeit wird das Ziel, via Web Scraping extrahierte Daten aus Onlinezeitungsartikeln zu visualisieren, angestrebt. Dabei wird besonderes Interesse auf Informationen über Standorte, Organisationen sowie Personen, welche in den Artikeln vorkommen, gelegt. Ein Beispiel hierfür wäre ein Artikel, welcher thematisch Elon Musk behandelt. Dadurch wird Elon Musk als Person respektive Tesla als Organisation und Amerika als Standort näher betrachtet. Die Wissensdaten werden anhand eines Wissensgraphen visualisiert. Für das Erstellen des Wissensgraphen wird eine Graphbibliothek namens „React-Force-Graph“ verwendet.

1.1 Aufgabenstellung und Zielsetzung

Im Vorfeld ist zu erwähnen, dass im Rahmen dieser Arbeit von Knoten als Nodes und von Kanten als Links gesprochen wird. Ein Artikel Node repräsentiert einen Artikel, während eine Entity Node wiederum eine Entität (Person, Standort oder Organisation) des Artikels darstellt.

Aufgabe der Arbeit ist die Programmierung einer Weboberfläche, welche Wissensdaten in Form eines Wissensgraphen visualisieren soll. Die genaue Definition und Erklärung eines Wissensgraphen werden im weiteren Verlauf noch tiefgehender behandelt. Die Weboberfläche wird durch das JavaScript Framework React entwickelt. Für die Aufgabenbewältigung kann man sich bezüglich der Framework Auswahl für die Programmierung der Weboberfläche frei entscheiden.

Ein Wissensgraph kennzeichnet sich vor allem durch Nodes und Links. Dementsprechend werden die erhaltenen Wissensdaten, welche aus Onlinezeitungsartikeln entnommen worden sind, durch Nodes repräsentiert. Ein Node wird daher als ein Artikel wiedergeben und visualisiert.

Die Links bestimmen die Ähnlichkeit zweier Nodes zueinander. Für die Bestimmung, ob ein Link zwischen zwei Nodes gesetzt wird, soll auf Ähnlichkeit beider Nodes zueinander verglichen werden.

Der Benutzer soll die Möglichkeit haben, nach Stichwörtern suchen zu können. Hierfür wird eine Suchleiste eingebunden, welche es erlauben soll, nach drei Entitätstypen suchen zu können: Gesucht wird nach Standorten, darunter Länder und Städte, und nach Personen, darunter meistens CEO´s der Organisationen. Unter einer Organisation wird hier hauptsächlich von Unternehmen und Firmen gesprochen. NGO´s und Vereine werden hier nicht betrachtet. Mithilfe einer Ähnlichkeitsmetrik soll vor dem Laden der Seite, eine Berechnung stattfinden, welche die Ähnlichkeit der einzelnen Entitäten der Artikel miteinander vergleicht und eine Gesamtähnlichkeit zweier Artikel zueinander berechnet. Sind zwei Artikel Nodes prozentual ähnlich genug, so soll ein Link, also eine Verbindung der beiden Nodes erstellt und visualisiert werden. Der Nutzer soll die Möglichkeit haben, den Schwellwert für die Linkerstellung selbst entscheiden zu dürfen. Hierfür wird ein Regler eingebunden, welcher von 0.1 – 1 bestimmmt ist. 0.1 wäre eine minimale Ähnlichkeit von 10% für die Erstellung eines Links. Ein Schwellwert von 1 bestimmt, dass eine Ähnlichkeit von 100% vorhanden sein muss, um einen Link erstellen zu können. Das Label der Artikel Nodes, welches die Überschrift des jeweiligen Artikels repräsentiert, soll auf den ersten Blick direkt ersichtlich sein. Nachdem ein Node ausgewählt und angeklickt wird, soll die zuvor angezeigten restlichen Nodes ausgefiltert werden. Eine neue Sicht soll erstellt werden, in der das angeklickte Node im Zentrum und die drei zuvor erwähnten Entitäten (*Organisationen, Personen und Standorte*) um dieses herum angezeigt werden sollen. Die Entitäten werden mit Links dem zentralen Artikel Node verknüpft. Wenn auf einer der Entitäten geklickt wird, so soll eben dieses im Zentrum stehen und weitere Artikel Nodes sollen um diese Entität herum mit Links verbunden sein. Der Flow wird damit geschlossen und es kann wieder auf einen Artikel werden.

1.2 Ziele der Arbeit

Hauptziel dieser Bachelorarbeit ist es, unstrukturierte Daten, welche allein nur begrenztes Wissen bereitstellen, durch einen Wissensgraphen zu visualisieren, um so die Daten einfacher zu durchsuchen und essenzielles Wissen auf einen Blick erhalten zu können. Die Daten werden dabei nicht verändert, jedoch soll durch die veränderte Darstellung ein besserer Überblick gewährt werden. Dem Benutzer wird die Möglichkeit gegeben, nach beliebigen Schlüsselbegriffen zu suchen. Folglich werden dem Nutzer eine durch Ihn festgelegte Anzahl von News Nodes angezeigt. Nach Auswahl eines Nodes erhält der Benutzer die wichtigsten Informationen über den angeklickten News Node. Das

resultierende Ziel der Vereinfachung und Veranschaulichung der Daten soll so erreicht werden.

1.3 Aufbau der Arbeit

Der inhaltliche Aufbau der Arbeit wird durch eine Einführung in den Stand der Technik eingeleitet. Hier wird zunächst der technische Rahmen erläutert, welcher für die Verständlichkeit der Thematik notwendig ist. Dabei sind insbesondere die Themen „Visualisierung von Daten durch Wissensgraphen“, die Auswahl der Technologie für die Visualisierung selbst React und „Wissensgraphen“ von größerer Bedeutung. Anschließend wird der gewählte Lösungsansatz beschrieben, um die darauffolgenden Ergebnisse besser nachvollziehen zu können. Abschließend werden die Ergebnisse noch einmal diskutiert und die Abschlussarbeit wird mit einer Zusammenfassung beendet.

2 Stand der Technik

2.1 Datenvisualisierung

Unter Datenvisualisierung oder Datenrepräsentation versteht man den Prozess, in welcher man Daten in einer bildhaften, simplifizierten Art und Weise darstellt, um essenzielle Informationen schnell und einfach erkennen zu können. Durch die mittlerweile starke Prozessorleistung gängiger Computer, ist es möglich, sehr große Mengen an Daten in kürzester Zeit verarbeiten zu können. Dadurch kann eine visuelle Repräsentation dieser Daten in Zeiten der „Big Data“ einfacher erreicht werden. “So definiert Adrian Merv Big Data als Daten, die in ihrer Größe klassische Datenhaltung, Verarbeitung und Analyse auf konventioneller Hardware übersteigen“ [3]. Es gibt eine unzählige Anzahl von Datenvisualisierungstechniken. Für das weitere Verständnis allerdings sind nur die Graph-Visualisierungstechniken von weiterer Bedeutung.

2.2 Was ist ein Graph?

Die Definition eines Graphen ist unter den verschiedenen Fachgebieten unterschiedlich. Zum Beispiel könnte man sagen, dass „ein Graph oder auch Diagramm eine Repräsentation von einer Serie von Punkten, Linien oder Kurven ist, welche untereinander verglichen und dargestellt werden können“ [4]. Dabei ist die Quintessenz, die Tatsache, dass Graphen eine Reihe verschiedenster Daten repräsentieren und dem Betrachter einen schnellen Überblick über diese Daten verschaffen sollen.

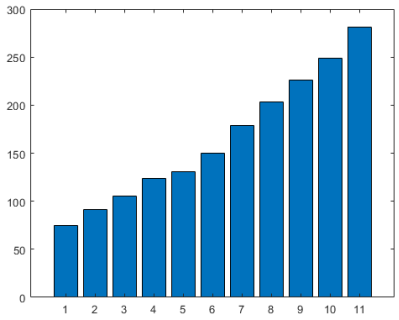


Abbildung 2.1 Beispiel Balkendiagramm [5]

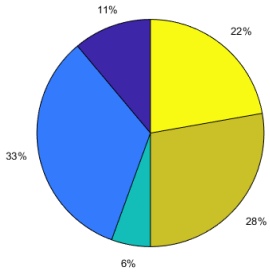


Abbildung 2.2 Beispiel Kreisdiagramm [6]

Graphen können in verschiedensten Formen auftreten. In diese Kategorie fallen beispielsweise die ubiquitären Balken/- Säulendiagramme sowie die klassischen Kreisdiagramme. Wie zuvor erwähnt jedoch werden in dieser Arbeit lediglich Wissensgraphen im Detail betrachtet.

2.3 Was ist ein Wissensgraph?

Nach den Datenanalysten Richard Brath und David Jonker ist ein Wissensgraph nichts anderes als eine strukturierte Repräsentation von Informationen, welche miteinander verbunden sind [7]. Einen Wissensgraphen zeichnet aus, dass er Nodes und Edges/ Links beinhaltet. Ein Node ist eine Entität, welcher in seiner gängigsten Form als Kreis in einem Wissensgraphen dargestellt wird. Diese können aber auch in anderen Formen wie zum Beispiel Bilder, Texte oder ähnlichem erscheinen. Sie haben die Funktion der Repräsentation von Datenobjekten. Die Aufgabe von Links, auch teils Edges oder Kanten genannt, ist es, eine Beziehung oder Relation unter den Nodes darzustellen. Dabei können Links viele verschiedene Arten von Beziehungen reflektieren.

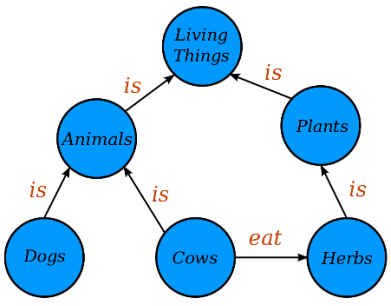


Abbildung 2.3 Beispiel Wissensgraph [8]

Schaut man sich das Beispiel eines Wissensgraphen in Abbildung 3 an, so erkennt man, dass ein Hund eine „Ist- Beziehung“ zu einem Tier hat. Das Tier wiederum hat eine „Ist-Beziehung“ zum Node „Living Things“, daher Lebewesen. Andererseits können auch

andere Beziehungen erstellt und dargestellt werden. In dem Beispiel erhält eine Kuh eine „Isst- Beziehung“ zu einem Pflanzen/ Kräuter (Herbs) Node, welches darstellt, dass Kühe Pflanzen/ Kräuter fressen. So erlauben es Wissensgraphen Information schnell und einfach darzustellen. Das dargestellte und angewendete Beispiel ist recht simpel gehalten worden, reflektiert jedoch die immanente und - für die Wissenschaft – obligatorische Funktionalität und Bedeutung von Wissensgraphen.

2.4 Kennzeichnung eines Wissensgraphen

Wissensgraphen können jedoch auch hochkomplexe Präsentationen von Daten und Verbindungen untereinander darstellen. Ein Wissensgraph kennzeichnet sich durch drei Charakteristiken:

- Datenbank à Daten sind in der Lage durchforscht und abgefragt zu werden.
- Graph à Repräsentieren Daten und können analysiert werden.
- Wissensbasis à Bergen Information, welche sowohl klar als auch interpretierbar vorliegen.

Für die Bergung von Wissen aus Wissensgraphen werden einige verschiedene Methoden verwendet. Hierfür können beispielsweise Klassen verwenden. Dem objektorientierten Modell zufolge können Entitäten Klassen zugeordnet werden, welche sowohl Super- als auch Subklassen enthalten können. So könnte man Lebewesen als Superklasse der Tiere zuordnen. Eine Kuh, wäre dann wiederum eine Subklasse gegenüber den Tieren. Dies hat den Vorteil, dass Datenmodelle besser dargestellt werden können. Es erfolgt eine bessere und anschaulichere Wahl der Entitäten als Nodes. Dadurch werden Nodes vermieden, welche für die Visualisierung eines Wissensgraphen weniger nutzbare Informationen bieten. Eine weitere Methode der verbesserten Darstellung von Wissensgraphen ist die Nutzung von Linktypen. Wie zuvor erwähnt, ist ein Link eine Verbindung zwischen zwei Entitäten, welche als Node dargestellt werden. In einem Wissensgraphen bilden Links wichtige Informationen über die Relation und Beziehung zwischen Nodes und bieten die Möglichkeit, verborgenes, nicht sofort erkennbares Wissen zu erkennen. Links können ohne einen Linktypen zueinander bestehen, jedoch haben Sie den Nachteil, dass die Beziehung der verbundenen Entitäten möglicherweise seitens Betrachter nicht impliziert werden können. Daher ist die Nutzung von Linktypen wie z.B. Kuh „ist“ Tier sinnvoll und empfehlenswert. Eine weitere Einordnung von Entitäten in Kategorien, beziehungsweise Taxonomien erlaubt es diese, ferner einzuordnen und zu beschreiben. Folgen wir unserem Tierbeispiel von vorher, könnte eine beispielhafte Taxonomie die Art der Säugetiere sein, welche sich durch bestimmte Merkmale in eine Klasse, daher Taxonomie einordnen lassen. Letztere Methode wäre die Einbindung von freien

Textbeschreibungen. Diese sollen den einfachen Zweck haben, weitere Klarheit über Entitäten zu verschaffen und das „Querying“ zu verbessern [9].

2.5 Use- Cases für Wissensgraphen

Wissensgraphen bieten ein großes Anwendungsgebiet in allen möglichen Branchen und Fachgebieten. Zum Beispiel werden diese in der Analyse der Lieferketten für Firmen verwendet. Dadurch das Wissensgraphen Informationen, welche vorher nur sehr schwer zu erkennen waren, hervorbringen können, kann man Information und Wissen über die Beziehung von Firmen untereinander erhalten. Dieses Wissen bietet die Möglichkeit auch Information über Lieferanten zu gewinnen, mit denen einige Firmen in der Öffentlichkeit weniger Kontakt haben. Sie befinden daher im Verborgenen und können so durch die richtigen Daten in Kombination mit einer Repräsentation durch Wissensgraphen gefunden werden. Ein weiteres Anwendungsgebiet ist die Marktanalyse. Daten aus Sozialen Medien wie beispielsweise Twitter, Reddit oder weiteren bieten viele Daten rund um Rezensionen und Bewertungen über Firmen und Unternehmen. Diese Daten können genutzt werden, um als Wissensgraph Informationen über die Verbindungen zwischen den Unternehmen, sowie zusätzlich deren Marktstellung hervorzubringen. Des Weiteren kann neues Wissen impliziert werden. Hier könnten Vorhersagen über Aktienentwicklungen gewonnen werden. Die rasante Entwicklung von Aktien, welche durch Soziale Netzwerke beeinflusst werden, kann man anhand der Bitcoin Aktie sehr gut veranschaulichen. Die Aktie fiel durch einen Tweet von Elon Musk um mehr als zehn Prozent. Dies waren nur einige von vielen Anwendungsgebieten der Wissensgraphen.

2.6 Navigierbarkeit in Wissensgraphen

Mit steigender Komplexität und Größe einiger Wissensgraphen, steigt auch die Schwierigkeit, in diesem Wissensgraphen navigieren zu können. Die Navigation kann durch diverseste Arten und Weisen erfolgen. Um diese für die Nutzer zu erleichtern, können Methoden wie beispielsweise eine Suchfunktion, welche die wesentlichen Nodes präsentiert, genutzt werden. Darüber hinaus muss die Navigation intuitiv erfolgen können. Dies bedeutet, dass ein Klick auf einen Node zum einen die Navigation zum angeklickten Node ansteuern, zum anderen aber auch dessen untergeordnete Nodes anzeigen soll. Das Mausrad soll den Zoom kontrollieren, die rechte Maustaste wiederum die Schwankung. Ein großes Problem in der Navigation durch einen Wissensgraphen ist die richtige Entscheidung des Pfades, den man durch die Nodes wählt. Oft stößt man auf Sackgassen, in denen die Nodes keine weiteren untergeordneten Nodes mehr haben [10]. Eine Variante, welches das Institut für Elektrische und Elektronische Ingenieure (IEEE)

gefunden hat, ist ein von AlphaGo inspirierter Algorithmus. AlphaGo ist eine künstliche Intelligenz, welche entwickelt wurde, um einer der komplexesten Brettspiele der Welt, namens „Go“ perfekt spielen zu können. Der Algorithmus lautet “Monte Carlo tree search“ kurz “MCTS“. Die Grundidee sowohl des Spielens von „Go“ als auch des Navigierens durch einen Wissensgraphen sind prinzipiell dieselben. Während die AI (künstliche Intelligenz) einen nächsten Zug für einen Stein wählen muss, wird am Beispiel von Wissensgraphen eine Entscheidung über die Wahl des nächsten Nodes bestimmt [11].

2.7 Technologiewahl für die Datenvisualisierung

2.7.1 React

Datenvisualisierung kann durch die verschiedensten Technologien umgesetzt werden. In dieser Arbeit wurde das die JavaScript Bibliothek/ Framework React verwendet.

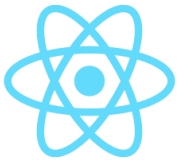


Abbildung 2.4 React Logo [12]

Dieses Framework eignet sich besonders gut für die Frontend Entwicklung, da hier der Fokus auf die View an sich gelegt wird. Im Gegensatz zu anderen JavaScript Frameworks wie beispielsweise Angular wurde React verwendet, da es während der Studienzeit als primäres Framework seitens der Lehrenden vermittelt worden ist.

2.7.2 React-Force-Graph

Für die Realisierung eines Wissensgraphen wurde Gebrauch einer Graphbibliothek namens „react-force-graph“ [13] gemacht. Die Entwickler der Bibliothek ist die Firma Vasco Asturiano. Die Bibliothek bietet Komponenten für das Erstellen von Wissensgraphen. Dabei können neben der 2-dimensionalen Variante auch 3-dimensionale Wissensgraphen erstellt werden. Für die Implementierung eines Wissensgraphen wird zunächst die gewünschte Komponente importiert. Anschließend kann der sogenannte „Forcegraph“ als React Komponente eingebunden werden. Der Forcegraph nimmt ein „graphData“ Attribut (Prop) an, in welcher ein Objekt zu übergeben ist, welches ein „nodes“ array sowie ein „links“ array zu beinhalten hat. Das Nodes Array besteht aus Objekten von Nodes. Ein Node Objekt zeichnet sich vor allem durch einen unique identifier (ID) aus. Des Weiteren können den Nodes Objekt noch weitere Werte (Values) mitgegeben werden. Diese werden später in der Implementierung weiter erläutert. Beispielsweise könnte

man „nodeColor“ als Value mitgeben, welcher später die Farbe des Node Objektes bestimmt. Das Links Array enthält Links ebenfalls als Objekte. Ein Link besteht aus einer „Source“ (Ursprung) und einem „Target“ (Ziel). Ein Link wird immer vom Ursprung hin zum Ziel gebildet. Dies ist ein wichtiger Punkt, welcher beachtet werden muss, wenn man Partikel oder Pfeile in Zielrichtung anzeigen lassen möchte. Links können dementsprechend mit Farben und weiteren Stilmitteln versehen werden. Diese werden als Props an die ForceGraph Komponente übergeben.

3 Implementierung

3.1 Projektaufbau & Hierarchie

Die Implementierung des Projektes ergab folgenden Projektaufbau:

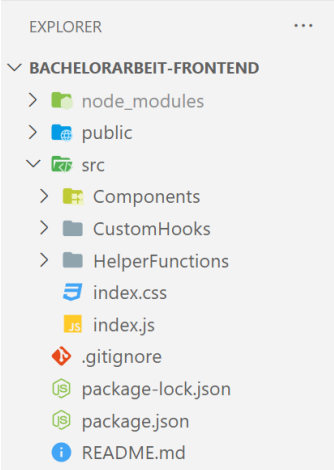


Abbildung 3.1 Projektaufbau

Im ersten Schritt wurde die Weboberfläche an sich programmiert. Wie zuvor genannt wurde hierfür das JavaScript Framework React verwendet. Wichtig zu erwähnen ist, dass die React Entwicklung komponentenorientiert ist. Dies bedeutet, dass die Weboberfläche, durch eigens erstellte Bausteine in Form von JSX- Komponenten zusammengesetzt wird. JSX ist eine Syntax Erweiterung von JavaScript und erlaubt es Elemente zu erstellen, welche eine Mischung aus klassischem HTML Template und JavaScript sind.

Folgende Komponenten- Hierarchie wurde aufgebaut:

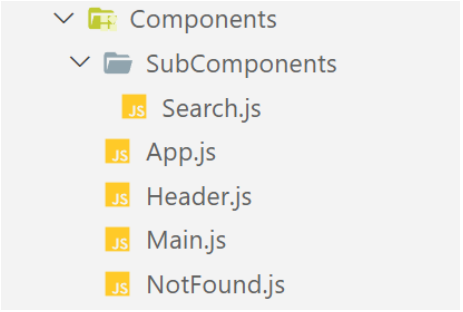


Abbildung 3.2 Komponenten

Initial wird die **App** Komponente gerendert. Diese setzt sich aus einer Header Komponente sowie einer Main Komponente zusammen. Die **Header** Komponente beinhaltet den Titel der Website sowie einen Reload Icon, welcher durch einen Klick die Seite neu lädt. Die Main Komponente ist die Haupt Komponente der Website und beinhaltet sowohl

eine **Search** als auch eine **ForceGraph** Komponente. Die Search Komponente nimmt eine Suchleiste mit sich, in welcher die Entitäten gesucht werden können. Hier können Organisationen, Personen oder Standorten suchen. Unter Standorte werden sowohl Städte als Länder mitgezählt. Um den Entitäten Typen festlegen zu können, welcher bei der Suche notwendig ist, wurden drei Radio Inputs eingebunden. Um beispielsweise nach einer Person zu suchen, muss ebenfalls die „Person“ Checkbox ausgewählt sein, um einen erfolgreichen Abruf durchführen zu können. Ebenfalls kann der User die Anzahl der anzuzeigenden Nodes bestimmen. Hierfür wurde ein Zahlen Input Feld miteingebunden. Von großer Bedeutung sind die Links, welche Nodes verbinden. Um festzulegen, welche Nodes miteinander verbunden werden sollen, wurde Gebrauch der Jaccard-Ähnlichkeit beziehungsweise des Jaccard- Index gemacht. Dieser zeichnet einen Wert zwischen 0 und 1, wobei 0 keine Ähnlichkeit und 1 eine maximale Ähnlichkeit zweier Mengen bestimmt. Der Schwellwert, also die minimale Ähnlichkeit, welche ein Node zum anderen haben muss, kann durch einen Slider seitens Benutzer eingestellt werden. Die ForceGraph Komponente nimmt wie zuvor erwähnt „**graphData**“ in Form eines Objektes an. Dieses Objekt hat einen „**nodes**“ und „**links**“ Array. Besonderer Fokus wird auch die Implementierung der Main Komponente gelegt, da es die relevanteste Logik für die Weboberfläche und dem endgültigen Ziel der Visualisierung eines Wissensgraphen beinhaltet.

3.2 News Data Abrufen

Initial wird beim Starten der Website eine Custom React Hook aufgerufen.

Die „useFetchSearch“- Hook nimmt folgende vier Argumente an:

1. url → *String*, Uniform Resource Locators – Link zum GraphQL Endpunkt
2. entityType → *String*, Typ der Entität, nach der gesucht werden soll (z.B. „Person“)
3. entity → *String*, Entität, nach der gesucht werden soll (z.B. „Elon Musk“)
4. nodeAmount → *Int*, Anzahl der Objekte, welche beim Aufruf zurückgegeben werden sollen

Die Hook initialisiert drei „useState“- Hooks:

1. data → *Array* mit abgerufenen Daten
2. isLoading → *Boolean*, mit Ladezustand der abgerufenen Daten
3. error → String, mit möglichem Abruf Fehler

```
const [data, setData] = useState([]);
const [isLoading, setIsLoading] = useState(true);
const [error, setError] = useState(null);
```

„useFetchSearch“ bildet zunächst einen „QUERY“ Template String, welcher dann per „POST“- Anfrage an den GraphQL Endpunkt gesendet wird.

```
`query {
   newsSearch(limit: ${nodeAmount}, entity: "${entityType}", keywords: "${entity}"){
    __typename
    id
    headline
    org{
      name
      sameAs
    }
    per{
      name
      sameAs
    }
    loc{
      name
      sameAs
    }

  }
 }`
```

Innerhalb der Hook wurde eine „getData“ Arrow Funktion geschrieben. Diese Funktion setzt zunächst „isLoading“ auf „true“ und leitet damit den Ladevorgang der Daten ein. Anschließend wird die „fetch“- Funktion der JavaScript- FetchAPI aufgerufen. Diese Funktion nimmt einen URL- String an und schickt eine „GET“- Anfrage an den mitgegebenen Endpunkt. Da GraphQL Anfragen jedoch über „POST“ Anfragen laufen, wird hier als optionales Argument, ein Objekt mitgegeben. Dieses Objekt enthält die „HTTP“- Methode, den Header sowie den Body in welcher der Query String überreicht wird.

Das Objekt sieht wie folgt aus:

```
{
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ query: QUERY }),
}
```

Bei erfolgreicher Anfrage erhält man ein JavaScript Promise. Die „.json()“- Funktion auf den Antwort Stream gibt ein weiteres Promise zurück, welches ein JavaScript Objekt übergibt.

```
.then((response) => response.json())
```

Das zurückgegebene JavaScript- Objekt kann dann entpackt dem „data“- State zugewiesen werden.

Zusätzlich wird „isLoading“ auf „false“ gesetzt und der „error“ verbleibt weiterhin auf „null“.

```
.then((data) => {
```

```
      setData(data.data.newsSearch);
      setIsLoading(false);
      setError(null);
    })
```

Bei nicht erfolgreicher Anfrage wird der Fehler über eine „catch“- Funktion aufgefangen und dem „error“- State zugewiesen. „isLoading“ wird dabei ebenfalls auf „false“ gesetzt.

Diese „getData“- Funktion ist in einer „useEffect“- Hook eingebunden.

```
useEffect(() => {
  const timer = setTimeout(() => {
    getData();
  }, 500);

  return () => clearTimeout(timer);
}, [entity, entityType, nodeAmount]);
```

Ein „useEffect“ nimmt als zweites Argument einen Dependency Array an. Wird ein Element aus diesem Array verändert, so wird automatisch auch der „useEffect“ erneut aufgerufen. In diesen Array wurden die Argumente der „useFetchSearch“- Hook eingebaut. Ebenfalls wurde ein Timer implementiert. Grund hierfür war, dass wenn eine Person in die Suchleiste geschrieben wurde, für jede Zeichenänderung ein erneuter Aufruf der Hook ausgelöst wurde. Dies würde zu unnötigen Datenabrufen führen. Deshalb wartet die Funktion jedes Mal erneut 500 Millisekunden, nachdem einer der Variablen aus dem Dependency Array verändert wurde. Wenn also zu Ende getippt wurde, so wird nach 500 Millisekunden die Funktion aufgerufen und der Aufruf der Daten erfolgt.

3.3 News Nodes erstellen

Die „useFetchSearch“- Hook wurde folgendermaßen in „Main“ eingebunden:

```
//! Fetching Data & Storing in State
 const {
   data: newsData,
   isLoading: isLoadingNewsData,
   error: errorNewsData,
 } = useFetchSearch(
   "http://195.37.233.209:4000/graphql",
   selectedEntityType,
   searchbarText,
   initialNodeAmount
 );
```

Als Argumente wurden drei „useState“- Hooks erstellt und mitgegeben. Initial wurden „selectedEntityType“ und „searchbarText“ als Leeren String gesetzt. Die initiale Anzahl der zu präsentierenden Nodes wurde beispielsweise auf 50 gesetzt.

```
const [searchbarText, setSearchbarText] = useState("");
const [selectedEntityType, setSelectedEntityType] = useState("");
const [initialNodeAmount, setInitialNodeAmount] = useState(50);
```

Die Datenbank gibt einen Array aus News- Objekten zurück. Ein solches Beispiel Objekt sieht wie folgt aus:

```
{
  "__typename": "News",
  "id": "someID",
  "headline": "someHeadline",
  "org": [
    {
      "name": "Chevron",
      "sameAs": "someID"
    },
  ],
  "per": [
    {
      "name": "Stanley Druckenmiller",
      "sameAs": " someID "
    },
  ],
  "loc": [
    {
      "name": "US",
      "sameAs": " someID "
    },
]}
```

Nachdem „NewsData“ erhalten wurde, wird eine „useEffect“- Hook ausgelöst, welche als Dependency Array sowohl die erhaltene „NewsData“ enthält als auch den den „jaccardThreshold“. Ändert sich einer der beiden States, so wird mit Aufruf der Funktion zunächst eine Entfernung der Duplikate eingeleitet. Die Duplikate sind seitens der Datenbank dadurch entstanden, dass einige Artikel mehrmals eingebunden wurden. Dadurch das jeder Artikel ein eigenes Objekt in der Datenbank ist, hatten die Duplikate

verschiedene ID´s. Dies hat die Entfernung der Duplikate erschwert, da diese nicht zu 100% gleich waren. Die Herangehensweise war wie folgendermaßen:

```
const nodes = [];
const links = [];
const uniqueHeadlines = new Set([]);
const uniqueIDs = new Set([]);

//* Removing Duplicates from Newsdata
  newsData.forEach((news) => {
    const isDuplicate = uniqueHeadlines.has(news.headline);
    if (!isDuplicate) {
      uniqueHeadlines.add(news.headline);
      uniqueIDs.add(news.id);
      pushNewsNode(nodes, news);
    }
  });
```

Für die Entfernung der Duplikate wird einmal über die „newsData“ geloopt. Für jedes news Objekt wurde die headline auf Einzigartigkeit geprüft. Wenn die Headline noch nicht im „uniqueHeadlines“ Set war, welches zuvor leer initialisiert wurde, wurde die Headline in das Set hinzugefügt. Zudem wurden die ID´s in ein separates uniqueID´s Set eingebunden. Dieses Vergleichs Set war für die Linkerstellung danach relevant. Nachdem die Headline zum Set hinzugefügt wurde, wurde ebenfalls mit der „pushNewsNode“- Funktion das News-Objekt als Node einem „nodes“ Array angehängt. Die Funktion nimmt zwei Argumente an: Zum einen einen Array, in den das Node Objekt angehängt werden soll und zum anderen „data“, welches in dem Fall das von der Schleife erhaltene „news“ Objekt ist.

```
function pushNewsNode(arrToPushTo, data) {
  arrToPushTo.push({
    id: data?.id,
    name: data?.headline,
    __typename: data?.__typename,
    color: "rgba(233, 245, 10, 0.8)",
    sizeInPx: 6,
  });
}
```

Nachdem alle Nodes dem „nodes“ Array angehängt wurden, werden die Node Links erstellt. Wie zuvor erwähnt wurde hier die Entscheidung, ob zwei Nodes verbunden werden

durch den Jaccard Index bestimmt. Im Falle dieser Arbeit wurden für die Berechnung des Index, die Ähnlichkeit der Personen, Standorte und Unternehmen zweier Nodes betrachtet.

Die Formel für den Jaccard Index bildet sich aus dem Quotienten zweier Mengen. Im Zähler wird die Anzahl der Schnittmenge beider Mengen bestimmt. Im Nenner wiederum die Anzahl der Vereinigungsmenge. Das Ergebnis liegt zwischen 0 und 1.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Abbildung 3.3 Jaccard Koeffizient - Formel [13]

```
//* Setting Node Links
    newsData.forEach((cmpNode1) => {
      newsData.forEach((cmpNode2) => {
        if (uniqueIDs.has(cmpNode2.id) && uniqueIDs.has(cmpNode1.id)) {
          if (cmpNode1 !== cmpNode2) {
            let jacPer = getJaccardIndexOf(cmpNode1.per, cmpNode2.per);
            let jacOrg = getJaccardIndexOf(cmpNode1.org, cmpNode2.org);
            let jacLoc = getJaccardIndexOf(cmpNode1.loc, cmpNode2.loc);
            let jaccardIndex = (jacOrg + jacPer + jacLoc) / 3;

            if (jaccardIndex >= jaccardThreshold) {
              pushNewsLink(links, cmpNode1, cmpNode2);
            }
          }
        }
      });
    });
```

Für die Berechnung des Jaccard Index mehrerer Mengen wird jeweils der Jaccard Index einzeln berechnet. Anschließend wird die Summe der erhaltenen Jaccard Indexe durch die Anzahl der summierten Indexe geteilt. Hierbei wird die Ähnlichkeit der drei Entitäten verglichen und durch Drei geteilt. Im Programm wurde eine Schleife in der Schleife erstellt, um jeweils alle anzuzeigenden Nodes miteinander zu vergleichen und anschließend einen Link zu bilden, falls der berechnete Jaccard Index größer oder gleich dem Jaccard Schwellwert ist. Abschließend wird nach der Entfernung der Duplikate und der Erstellung der Links, dass „nodes“- State mit entsprechenden Daten gesetzt.

```
setNodes({ nodes: nodes, links: links });
```

3.4 Node Click Handhabung

Die ForceGraph Komponente hat einen „onNodeClick“- Handler, welche bei Klick auf einen Node, ein Node und Event Objekt mitgibt. Für die Handhabung der Node Klicks, war in diesem Fall, dass Node Objekt von größerer Bedeutung.

```
onNodeClick={(node) => {
        if (node.__typename !== "News") {
          const arr = [node];
          handleNodeClick(arr);
        } else
          handleNodeClick(newsData.filter((news) => news.id === node.id));
      }}
```

Es wird zunächst geschaut, ob der Typ des Nodes einem Artikel also einer News entspricht. Falls dies so ist, so wird über die „newsData“ gefiltert und nach dem News Objekt gesucht, welches dieselbe „ID“ wie das angeklickte Node hat. Man könnte sich fragen, weshalb nicht einfach das „node“ Objekt weitergeben wird. Der Grund hierfür ist, dass das „node“ Objekt, welches vom Klick Handler zurückgegeben wird, mangelnde Informationen enthält, welche für die weitere Visualisierung der untergeordneten Nodes notwendig ist.

```
{
 color: "lightgreen",
 fx: undefined,
 fy: undefined,
 id: "Q274591",
 index: 1,
 name: "EDF",
 sizeInPx: 5,
 vx: 0.0001927912606040916,
 vy: 0.00005494588017654895,
 x: -8.154550224621786,
 y: 28.610182557064373,
 __indexColor: "#180032",
 __typename: "org",
}
```

Die Informationen darüber, welche Entitäten zum angeklickten News Node gehören müssen daher aus den vorher abgerufenen Daten entnommen werden.

```
const handleNodeClick = (nodeArr) => {
  setClickedNode(nodeArr[0]);
};
```

Die „handleNodeClick“ Funktion nimmt einen Array an, den man vorher nach dem passenden News Node gefiltert hat. War das angeklickte Node, nicht vom Typen „News“, so wird das Node Objekt in einen Array gepackt und weitergegeben. Hier benötigt man lediglich die „ID“, um einen neuen Datenabruf einer Entität einzuleiten.

Ein weiterer „useEffect“ wird aufgerufen, da dieser als Abhängigkeit die “clickedNode“ State enthält. Dabei wird über den Typen des States eine Switch- Abzweigung aufgerufen.

„ClickedNode“ vom Typ: „News“:

```
// Click on News Node
      case "News":
        setCanGoBack(true);
        pushNewsNode(nodes, clickedNode);
        clickedNode?.org?.forEach((orgNode) => {
          if (orgNode.sameAs) {
            pushNodes(nodes, orgNode, "org", "lightgreen");
            pushLinks(links, clickedNode, orgNode);
          }
        });
        setLastNodes((prevState) => [
          ...prevState,
          { nodes: nodes, links: links },
        ]);
        setNodes(() => setNodes({ nodes: nodes, links: links }));
        break;
```

Bei einem Klick auf einen News Node wird zunächst das angeklickte Node in ein Array eingefügt. Anschließend wird über die Entitäten des News Nodes durchgelaufen. Für jede Organisation im „org“ Array wird zunächst geschaut, ob eine ID vorhanden ist (sameAs). Falls eine ID vorhanden ist, wird das Entity Node in den nodes Array eingefügt. Dies wird ebenfalls für Standorte („loc“) und Personen („per“) gemacht. Aus Gründen der Überschaubarkeit wurden die beiden letzteren oben im Code weggelassen.

„ClickedNode“ vom Typ: „Organisation, Person oder Standort“:

```
// Click on Entity Node
      case "org":
      case "per":
      case "loc":
        setCurrentEntityType(clickedNode.__typename);
        setCurrentEntityQid(clickedNode.id);
        setCurrentEntityName(clickedNode.name);
        break;
```

Um einen Entity Node als zentralen Node anzeigen, und dessen untergeordnete Nodes anzeigen zu können, muss ein neuer Datenaufruf getätigt werden. Dieses Mal wird jedoch nur nach einer bestimmten Entität gesucht. Deshalb ist Entitäten Typ, Name und Qid notwendig. Diese werden in separate States gesetzt und als Argumente in die zweite Custom Hook weitergegeben.

3.5 Datenabruf Entitäten

Die „useFetchEntity“- Hook nimmt folgende vier Argumente an:

1. url → *String*, Uniform Resource Locators – Link zum GraphQL Endpunkt
2. entityType → *String*, Typ der Entität, nach der gesucht werden soll (z.B. „Person“)
3. qid → *String*, ID, nach der gesucht werden soll.
4. entityName → *String*, Name, name der Entität

Die Hook initialisiert drei „useState“- Hooks:

1. data → *Array* mit abgerufenen Daten
2. isLoading → *Boolean*, mit Ladezustand der abgerufenen Daten
3. error → String, mit möglichem Abruf Fehler

Ähnlich der „useFetchSearch“- Hook wird nun ein Datenabruf ausgeführt. Dabei wird jedoch zunächst geschaut, ob eine „qid“ und eine „entityType“ existieren, da ansonsten der Datenabruf fehlschlagen wäre.

```
useEffect(() => {
  setIsLoading(true);
  if (entityType && qid) {
    fetch(url, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ query: QUERY }),
```

```
    })
      .then((response) => response.json())
      .then((data) => {
        setData(data["data"][getDataUnwrapper(entityType)]);
        setIsLoading(false);
        setError(null);
      })
      .catch((err) => {
        setIsLoading(false);
        setError("Cannot find any Data concerning the Entity: " + entityName);
      });
  }
}, [url, entityType, qid, entityName]);
```

In der Main Komponente wurde die Custom Hook folgendermaßen eingebunden:

```
const {
  data: entityData,
  isLoading: isLoadingEntityData,
  error: errorEntityData,
  setError: setError,
} = useFetchEntity(
  "someURL",
  currentEntityType,
  currentEntityQid,
  currentEntityName
);
```

Entsprechend für eine Änderung des „entityData“ States wird ein useEffect ausgelöst. Dieser hat die Funktion, dass „nodes“ State zu aktualisieren und die untergeordneten Nodes einer Entity Node anzuzeigen. Zunächst wird geschaut, ob „entityData“ existiert. Anschließend wird wieder eine Switch Verzweigung auf den Typen der Entität angewendet. Je nach Typ, stehen unterschiedliche Daten zur Verfügung, welche benutzt werden können, um untergeordnete Nodes zu erstellen.

```
case "per":
        pushEntityMainNode(nodes, entityData, clickedNode?.__typename, "red");
        entityData?.employer?.forEach((org) => {
          pushEntityNode(nodes, org, "org", "lightgreen");
```

```
        pushEntityLinks(links, clickedNode, org);
      });
      break;
```

Eine Datenabruf auf eine Person gibt Informationen darüber, in welchen Organisationen Er oder Sie Arbeitgeber ist. Dementsprechend werden die Nodes erstellt und als untergeordnete Nodes mit einem Link der Person verknüpft. Zudem ist erwähnenswert, dass eine unterschiedliche Farbgebung je nach Typ gegeben wurde. Organisationen haben als eine dunkelgrüne Farbe, wenn Sie als zentrales Node dargestellt werden und eine hellgrüne Farbe, wenn Sie ein Untergeordnetes sind. Standorte sind Dunkelblau und Hellblau gewählt worden und Personen haben eine rote Farbe als zentrale und rosa Farbe als untergeordnetes Node erhalten. News Nodes sind gelb gewählt. Letztlich wurden wieder die einem Array eingefügten Nodes und Links dem „nodes“ State eingefügt, welches als „graphData“ der ForceGraph Komponente mitgegeben wird.

Bezüglich der Implementierung war die vorangegangene Erklärung für das Verständnis des Programmes von größerer Bedeutung. Insgesamt wurden noch viele Nebenfunktionen geschrieben, welche im Anhang erwähnt werden, falls diese als notwendig für weiteres Verständnis sind. Das gesamte Projekt wurde in einer Github Repository hinterlegt und kann unter folgendem Link erreicht werden:

https://github.com/dglalperen/bachelorarbeit-frontend.git

4 Ergebnisse

4.1 Überblick

Die Programmierte Single Page Weboberfläche sieht wie folgt aus:

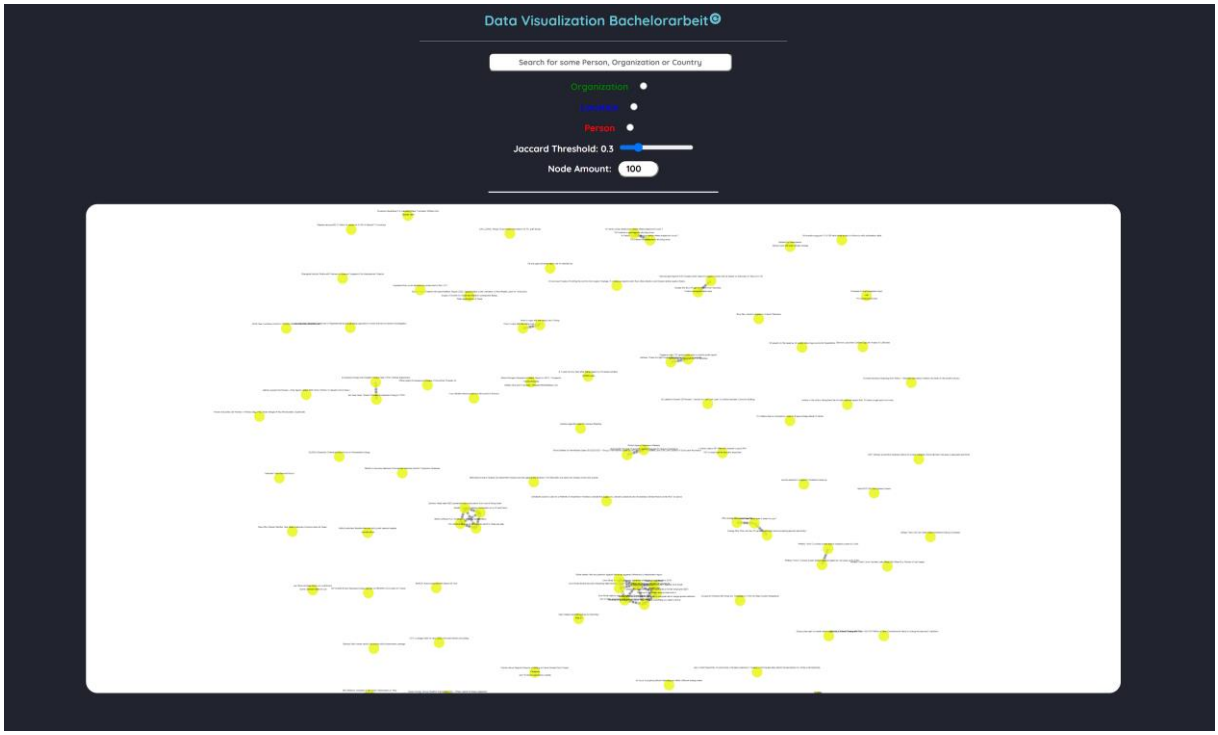


Abbildung 4.1 Weboberfläche

Zu erkennen ist der beschriebene Header, welcher vergrößert in Abbildung 4.2 erkennbar ist. Die Hauptkomponente als ForceGraph (siehe Abbildung 4.1) lädt initial die ersten 50 News Nodes der Datenbank und präsentiert diese. Die Nodes können beliebig bewegt und angeklickt werden. Um eine bestimmte Anzahl von Nodes zu erhalten, kann der Jaccard Schwellwert beliebig verstellt werden. Hierzu kann der in Abbildung 4.2 zu sehende Schieberegler mit der Bezeichnung „Jaccard Threshold“ benutzt werden.

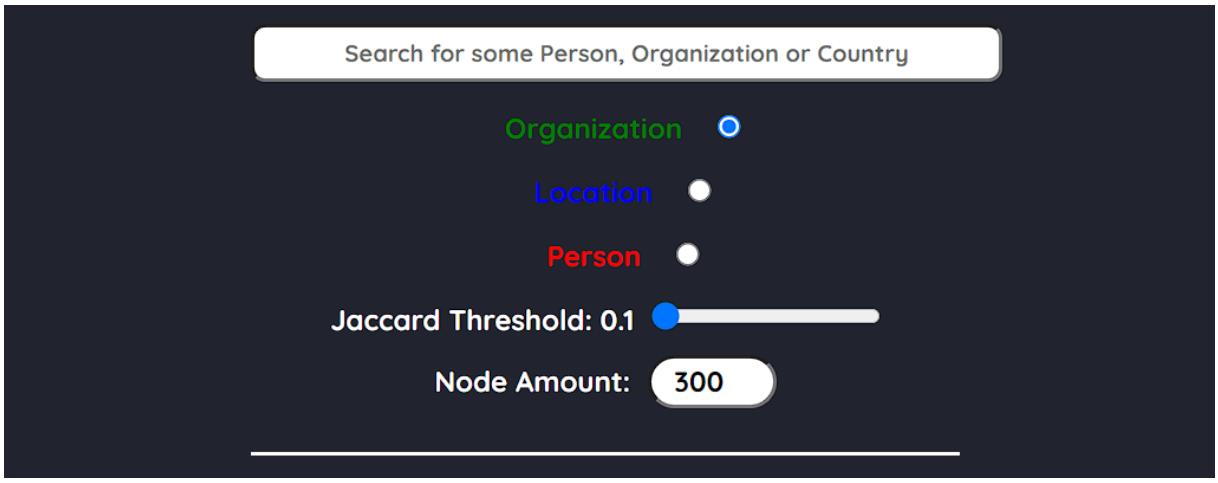


Abbildung 4.2 Header mit Search

4.2 Suche nach einer Entität

Möchte der Nutzer jedoch spezifische Nodes haben, in welcher eine bestimmte Entität auftritt, so kann die Suchfunktion genutzt werden. Um ein erfolgreiches Ergebnis aus der Suche zu erhalten, muss zudem der Entitäten Typ ausgewählt werden. Eine Mögliche Beispiel Suche ergibt folgendes:

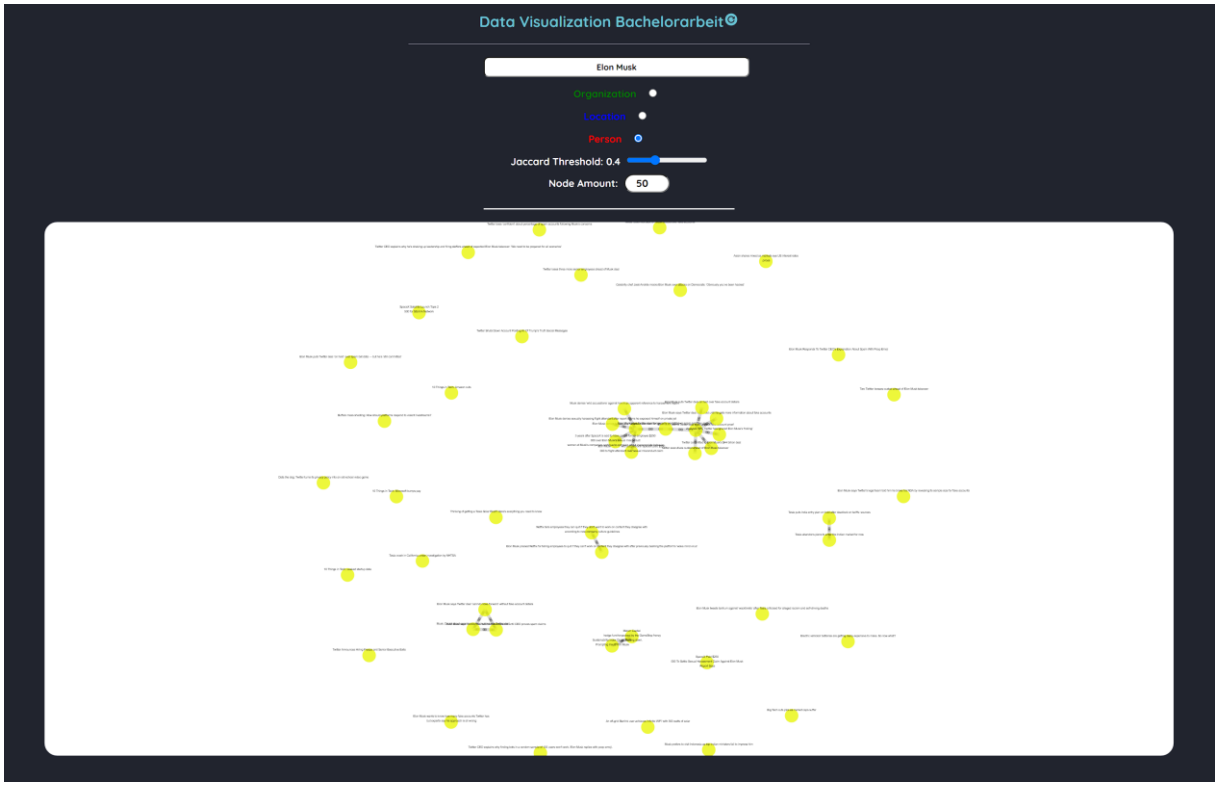


Abbildung 4.3 Beispiel Suche - Elon Musk

Hier wurde beispielsweise nach Elon Musk gesucht. In das Input Feld „Nodes Amount“ wird eine gewünschte Anzahl an News Nodes eingegeben. Aus einem Ergebnis von

News Nodes kann dann ein gewünschter Node ausgewählt und angeklickt werden. Hierfür wurden die Überschriften der Artikel, aus welchen die News Nodes bestehen, über dem jeweiligen Node platziert. Es können aber Cluster gesucht werden, welche gewisse Ähnlichkeiten bezogen auf die vorhandenen Entitäten aufweisen. Hierfür kann der Jaccard Threshold nach Belieben verschoben werden. Eine hoher Jaccard Threshold / Schwellwert würde bedeuten, dass eine hohe Ähnlichkeit vorhanden sein muss, um einen Link zwischen zwei News Nodes zu erstellen.

4.3 Fehler Handhabung & Navigation

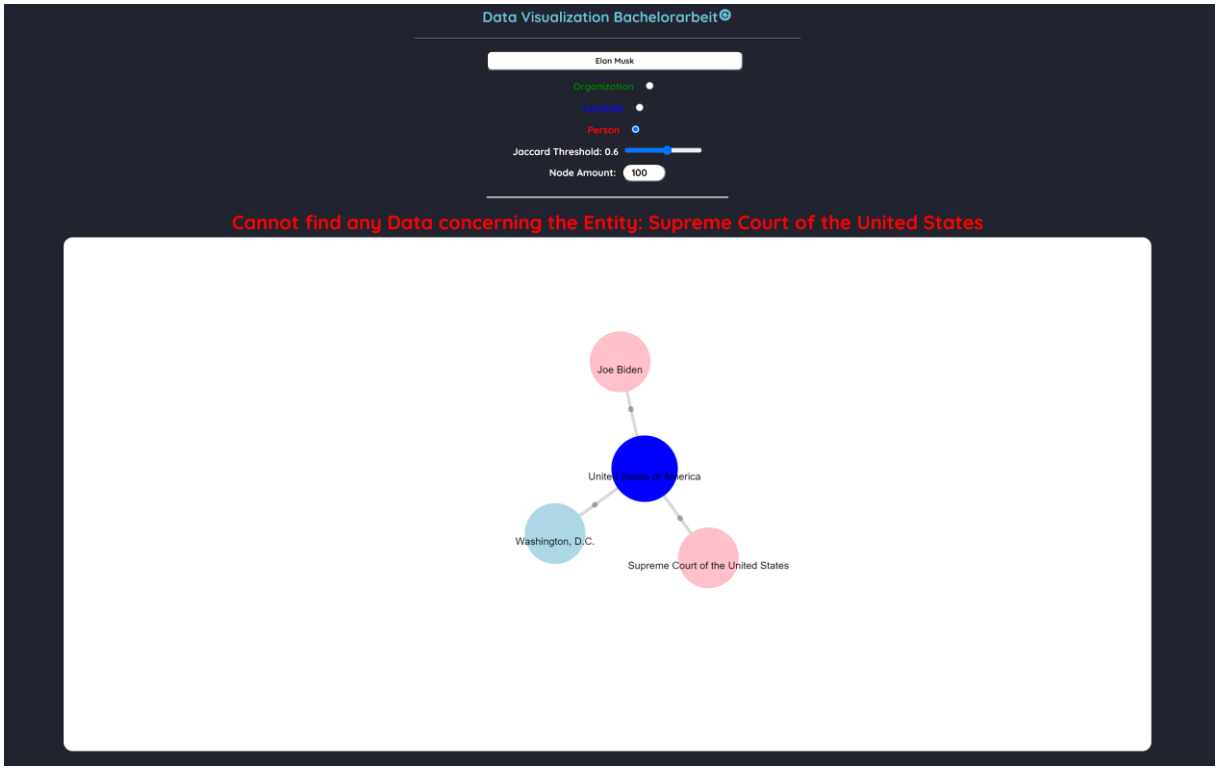


Abbildung 4.4 Error Handling

Können keine Daten, bezüglich einer angeklickten Entity Node gefunden werden, so wird eine Fehlermeldung angezeigt (siehe Abbildung 4.4). Dies kann durchaus passieren, da Daten in der Datenbank fehlen oder fehlerhaft sein können. In solch einem Fall kann der Nutzer ein anderes Node anklicken, eine neue Suche starten oder den Reload-Button Rechts neben dem Haupttitel anklicken.

Der Nutzer hat ebenfalls die Möglichkeit eine Nodes Ebene zurückzugehen. Hierfür wurde ein „Zurück“- Button (siehe Abbildung 4.6) eingebunden. Dies kann häufig im Falle eines „Dead Ends“ wie beispielsweise in Abbildung 4.5 zu sehen, also in einer Situation, in der ein angeklicktes Node keine untergeordneten Nodes enthält, genutzt werden. Der

Zurück Knopf erscheint unter dem Input Feld „Node Amount“. Zu beachten ist, dass dieser nur erscheint, nachdem mindestens eine Node Ebene tiefer geklickt wurde.

Abbildung 4.5 Node mit Dead End

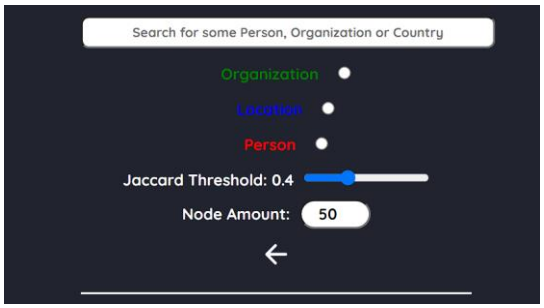


Abbildung 4.6 Zurück Knopf

4.4 Node Click Handhabung

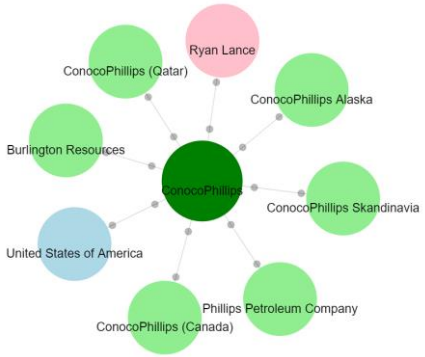


Abbildung 4.7 Entity Node + Untergeordnete Nodes

Nachdem eine Entität angeklickt wurde, wird dieses als zentrales Node platziert. Um dieses herum werden alle untergeordneten Entitäten als individuelle Nodes eingefügt. Eine Veranschaulichung dessen ist in Abbildung 4.7 erkennbar. Ein Link wird immer zum Ursprung hin erstellt. Helle Farben symbolisieren immer untergeordnete Nodes, während dunkle stehts einen zentralen Eltern Node darstellen.

5 Diskussion der Ergebnisse

5.1 Performance Test

Um eine kritische Betrachtung der Ergebnisse durchführen zu können, wurden Performance Test durchgeführt. Hierfür wurde das Performance Test Tool „Lighthouse“ verwendet, welches sich durch einfache Nutzung und realistische Performance Ergebnisse bewiesen hat.

Die Performancetests wurden mit folgenden Rahmenbedingungen durchgeführt:

- CPU → AMD Ryzen 5 2600, 6 Kerne, 3.4GHz
- Betriebsystem → Windows 11 Pro
- GPU → Radeon RX570 8GB
- Browser → Google Chrome Version 102.0.5005.115

Getestet wurde die Performanceentwicklung in Bezug auf die Anzahl der zu präsentierenden Nodes. Dabei wurden insgesamt vier Fälle betrachtet:

1. 10 Nodes
2. 100 Nodes
3. 500 Nodes
4. 1000 Nodes

Für jeden Fall wurde der Performancetest drei Mal durchgeführt, um zufällige Abweichungen vermindern, gar ausschließen zu können. Die Ergebnisse der individuellen Tests wurden im Anhang hinterlegt. Nach Angaben der Lighthouse Dokumentation sind mögliche Einflussfaktoren auf das Endergebnis:

- Internetverbindung
- Das ausführende Gerät
- Antivirus Software
- Browser Erweiterungen, welche JavaScript einbinden und Netzwerk Anfragen verändern

Lighthouse gibt auf einen durchgeführten Performancetest, einen Prozentscore von 0 – 100% zurück. Ein Score von 100 wäre damit ein perfekter Score, während 0 das schlechteste Score-Ergebnis ergeben würde. Für die Bestimmung eines Scores werden insgesamt sechs Punkte betrachtet. Die Bedeutung der einzelnen Metrikpunkte werden in der Tabelle 5.1 beschrieben. Die Gewichtung der einzelnen Metrik Punkte sind unterschiedlich gesetzt. Dementsprechend ist der Fokus auf höher gewichtete Punkte zu legen.

Tabelle 5.1 Lighthouse Score Metrik Punkte

FCP 10%	SI 10%	LCP 25%	TTI 10%	TBT 30%	CLS 15%
Zeit, die die Seite benötigt, um den ersten Text oder das erste Bild anzuzeigen.	Misst, wie schnell der anzuzeigende Content angezeigt wird.	Zeit, die die Seite benötigt, um den größten Text oder das größte Bild anzuzeigen.	Zeit, die die Seite benötigt, um komplett interaktiv zu sein.	Summe der Zeitperiode, zwischen FCP und TTI.	Misst die Beweglichkeit der zu sehenden Elemente im Viewport

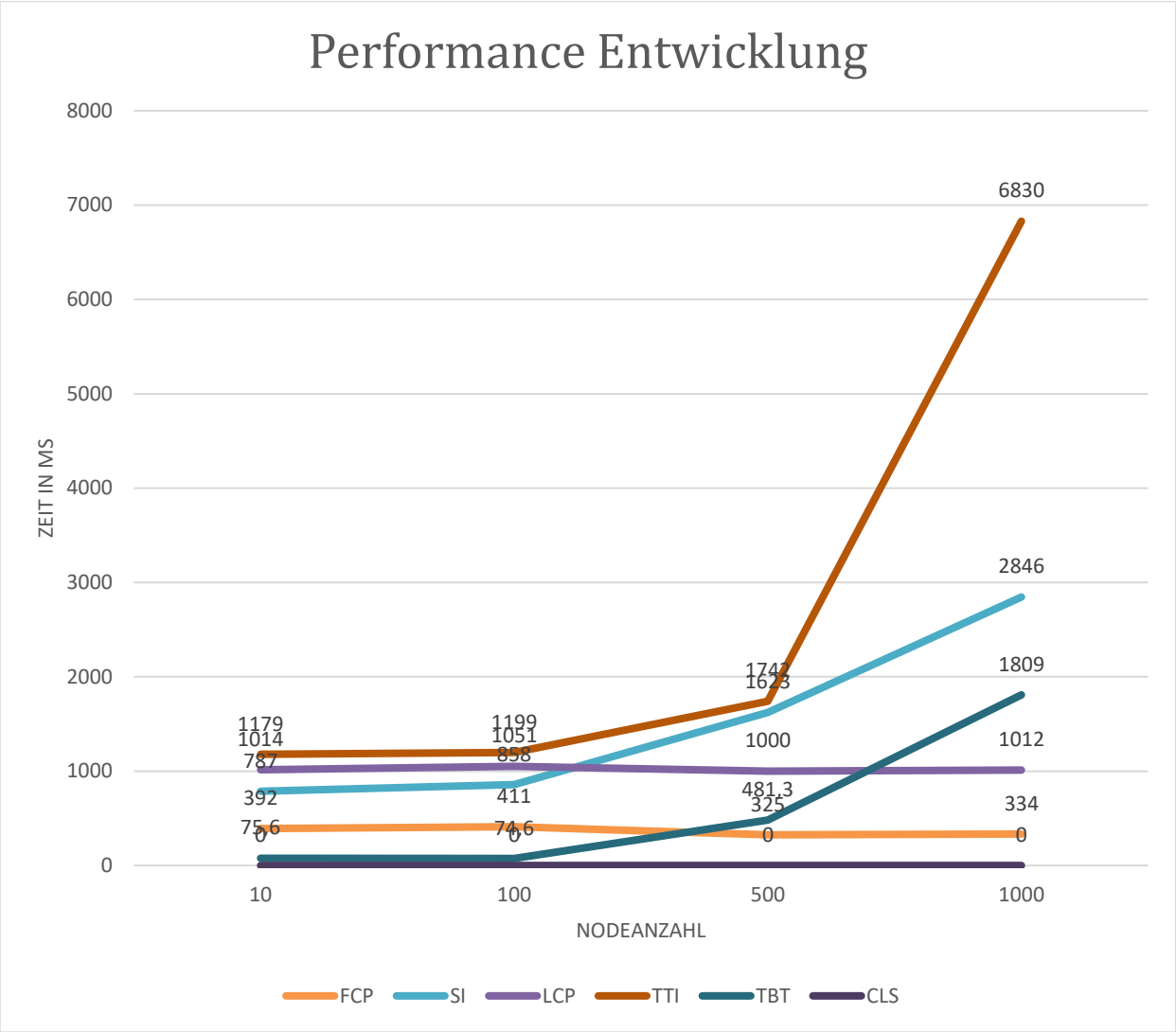
Die Performance Scores der 4 Fälle sind auf Tabelle 5.2 folgendermaßen:

Tabelle 5.2 Performance Scores

10 Nodes	100 Nodes	500 Nodes	1000 Nodes
98%	98%	76%	54%

Sowohl 10 als auch 100 Nodes ergaben einen nahezu perfekten Score von 98%. Mit steigender Node Anzahl wurde jedoch der Score immer schlechter. Bei 500 Nodes gab es einen Performance Score von 76%, welcher für die fünffache Anzahl an Nodes zu seinem Vorgänger zwar bemerkenswert ist, jedoch einen Trend für die sinkende Gesamtperformance darstellt. Mit 1000 Nodes war das Ergebnis bei lediglich 54% und hat damit einen starken Performance-Verlust trotz performanter Hardware gezeigt. Die Gründe für den Verlust von Performance werden klarer, wenn man sich die einzelnen Metrik Punkte für die Bewertung des Scores näher betrachtet.

Die Performance Entwicklung wurde anhand eines Diagrammes dargestellt. Dabei wurde jeweils der Mittelwert für die drei durchgeführten Tests bestimmt. Die y-Achse gibt die Zeit in Millisekunden wieder. Die x-Achse wiederum die Node Anzahl.



Während der FCP und LCP nahezu gleichgeblieben sind, erkennt man eine Steigerung in den Punkten TTI, SI und TBT. Der CLS Wert ist stetig bei 0 geblieben. Mit steigender Node Anzahl ist eine Verlangsamung des Speed Index zu erkennen. Zudem stieg die Total Blocking Time, welcher eine erhöhte Zeitperiode zwischen FCP und Interaktivität (TTI) repräsentiert. Bei 500 Nodes dauerte es durchschnittlich bis zu 1.7 Sekunden für eine vollständige Interaktivität der Seite. Diese Dauer stieg bei Verdopplung der Node Anzahl von 500 zu 1000 Nodes um etwa das Vierfache auf 6.83 Sekunden an. Verglichen zum Standard schlug sich der Speed Index bei 500 Nodes mittelmäßig, während er bei 1000 Nodes mit durchschnittlich 32% unterdurchschnittlich schlecht war. Die Total Blocking Time war bei beiden unterdurchschnittlich schlecht ausgefallen. 500 Nodes hatte dabei 31% und 1000 Nodes 12% erhalten. Der letzte Metrik Punkt, die TTI fiel nur bei 1000 Nodes in den unterdurchschnittlichen Bereich mit einem Score von 12%. Bei 500 Nodes war dieser mit 98% im grünen Bereich. Aus diesen Erkenntnissen lässt sich folgendes schlussfolgern: Bei Datenabruf von größeren Mengen an Daten, in unserem Fall 1000 News Objekte für 1000 Nodes, so erkennt man, dass eine Verlangsamung der

Seite stattfindet. Die Seite ist erst bei Ankunft aller Daten völlig aktiv. Deshalb wird sowohl SI als auch der TTI drastisch erhöht. Dies hat Auswirkungen auf das fertige Laden der Seite. Weitere Gründe für die erhöhte Zeitdauer und die schlechtere Performance bei hoher Node Anzahl ist die Linkbildung. Wie in der Implementierung behandelt, ist für die Bestimmung der Links, die Berechnung des Jaccard Index notwendig. Um jeden Node, außer sich selbst zu vergleichen und den Jaccard Index zu berechnen, wird eine Schleife in einer Schleife eingebunden. Dieser erhöht die Algorithmus Komplexität auf O(n^2), daher auf das Quadratische. Bei 10 Nodes wären dies (100 - 10), daher 90 Berechnungen. Für 1000 Nodes wären dies jedoch 999.000 Berechnungen und könnte ein Grund für die Verlangsamung mit steigender Node-Anzahl erklären. Des Weiteren spielen Faktoren wie Internetverbindung und Latenz eine Rolle und können mit erhöhter Datenrate, eine länger dauernde Anfrage zur Folge haben. Dementsprechend kämen die Daten verspätet an und würden die Performance beeinflussen.

5.2 Überprüfung des Jaccard Index

Um die Richtigkeit der Implementierung bezüglich des Jaccard Index feststellen zu können, wurde ein Beispiel Cluster ausgewählt und der Jaccard Index manuell berechnet. Dabei wurde der in Abbildung 5.1 zu dargestellte Cluster mit einem Jaccard Index von 1 gewählt.

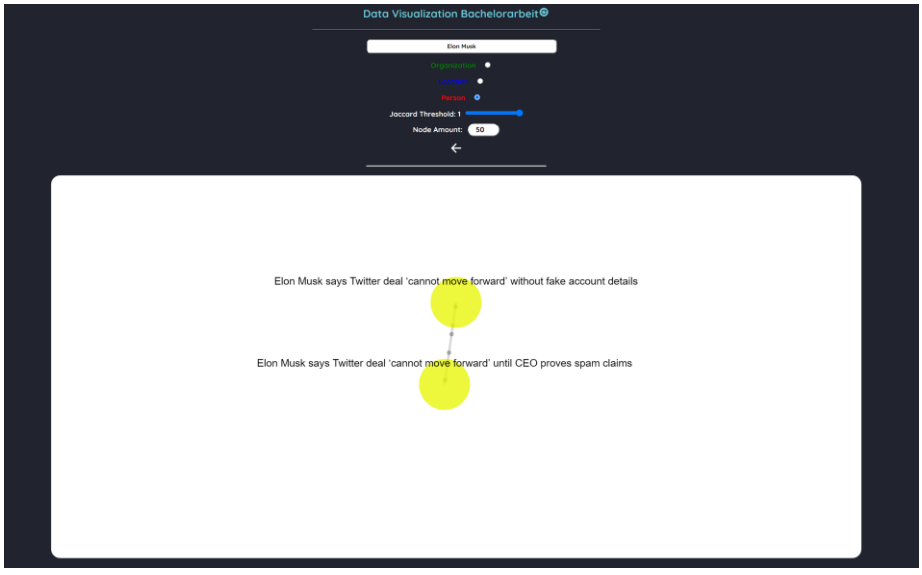


Abbildung 5.1 Beispiel Cluster mit Jaccard Index 1

Nach Einstellung des Jaccard Schwellwertes auf 1, hat folgender Cluster bestehend aus zwei Nodes einen Link erhalten. Wie zuvor erwähnt, wird der Jaccard Index aus den Entitäten der Nodes berechnet. Da man drei Entitäten Typen haben, muss für jeden Typen ein Jaccard Index berechnet und anschließend die Summe der drei entstandenen

Indexe zusammengeführt werden. Das Endergebnis berechnet sich aus dem Quotienten der Summe und der Anzahl der Entitäten Typen also Drei. Schaut man sich nun den oberen Node mit der Überschrift „Elon Musk says Twitter deal ‚cannot move forward' without fake account details" an, so erhält von folgende untergeordnete Nodes:

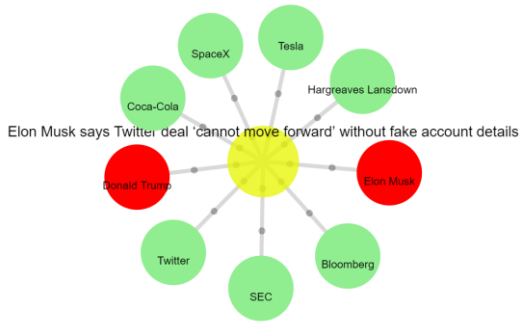


Abbildung 5.2 Vergleichs Node 1

Der verbundene Node wiederum hat folgende:

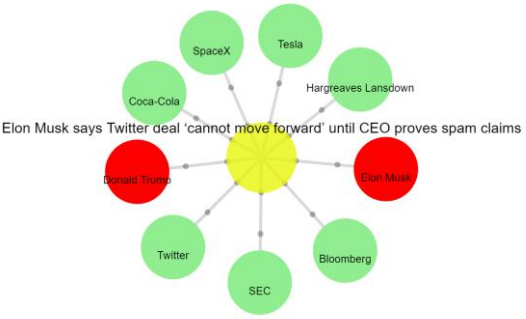


Abbildung 5.3 Vergleichs Node 2

Zu erkennen ist, dass beide Nodes, dieselben untergeordneten Nodes haben. Tabellarisch betracht ist dieser Fakt nocheinmal zu unterstreichen:

Untergeordnete Nodes	**Node 1 – Abbildung 5.2**	**Node 2 – Abbildung 5.3**
Personen	Donald Trump, Elon Musk	Donald Trump, Elon Musk
Standorte	/	/
Organisationen	Coca-Cola, SpaceX, Tesla, Hargreaves	Coca-Cola, SpaceX, Tesla, Hargreaves

	Lansdown, Twitter, SEC, Bloomberg	Lansdown, Twitter, SEC, Bloomberg

Um nun den Jaccard Index zu berechnen, wird für Personen, Standorte und Organisationen jeweils der Index einzeln berechnet. Da es keine Organisationen bei beiden Nodes gibt, können diese in der Berechnung weggelassen werden. Der Jaccard Index der Personen beider Nodes wäre 1. Dieser berechnet sich folgendermaßen:

Länge der Schnittmenge / Länge der Vereinigung

Im Falle der Personen hätten wir eine Schnittmengenlänge von 2 und eine Vereinigungsmengenlänge von 2. Der Quotient hieraus ist ein Jaccard Index von 1. Das gleiche wird bei den Organisationen berechnet und ergibt ebenfalls einen Jaccard Index von 1.

Dadurch ergibt sich ein insgesamter Jaccard Index von 1, welcher die Richtigkeit der Implementierung verifizieren kann.

6 Fazit & Ausblick

Mit Rückblick auf die Arbeit, ist eine Visualisierung der erhaltenen Wissensdaten durch einen Wissensgraphen gelungen. Die geplanten Ziele und Funktionalitäten, welche in Zusammenarbeit mit meinem Betreuer entstanden sind, wurden umgesetzt und getestet. Dennoch sind einige Punkte zu erwähnen, die hätten verbessert werden können: Zum einen hätte man die Berechnung der Node Links Cachen können, um bei erneutem Laden derselben Nodes, eine verkürzte Ladezeit zu erbringen. Des Weiteren hätte man beispielsweise noch weitere Informationen über Nodes einbinden können. Weiterhin wäre eine Optimierung der Software für eine verbesserte Performance bei hohen Node Zahlen besonders vorteilhaft für die gesamte User Experience. Nichtsdestotrotz hat dieses Projekt noch viele Erweiterungsmöglichkeiten, welche in Zukunft implementiert werden können. Dies würde einen verbesserten Einblick in Wissensdaten geben und Möglichkeiten gewähren, aus den vorhandenen Daten Wissen zu generieren, welches impliziert werden kann. Mit einer steigenden Komplexität von Wissensdaten und der Erschwernis, aus diesen Informationen zu gewinnen, welche sich vor allem um die Beziehungen der Daten zueinander bezieht, können Wissensgraphen helfen, dieses Wissen schnell und einfach zu vermitteln.

7 Anhang

7.1 Programmier Code:

7.1.1 Apps.js

```
import Header from "./Header.js";
import { BrowserRouter as Router, Route, Routes } from "react-router-dom";
import NotFound from "./NotFound.js";
import Main from "./Main.js";

function App() {
  return (
    <Router>
      <div className="App">
        <Header />
        <div className="content">
          <Routes>
            <Route path="*" element={<NotFound />}></Route>
            <Route path="/" element={<Main />}></Route>
          </Routes>
        </div>
      </div>
    </Router>
  );
}

export default App;
```

7.1.2 Header.js

```
import { IoReloadCircleSharp } from "react-icons/io5";

const Header = () => {
  return (
    <nav className="navbar">
      <h1>Data Visualization Bachelorarbeit</h1>
      <h1 className="refresh-btn" onClick={() => window.location.reload()}>
        {<IoReloadCircleSharp />}
      </h1>
    </nav>
  );
};

export default Header;
```

7.1.3 Main.js

```
import ForceGraph2D from "react-force-graph-2d";
import Search from "./SubComponents/Search";
import { useEffect, useState } from "react";
import useFetchSearch from "../CustomHooks/useFetchSearch";
import useFetchEntity from "../CustomHooks/useFetchEntity";
import {
  pushEntityMainNode,
  pushEntityNode,
  pushNewsNode,
  pushNodes,
} from "../HelperFunctions/nodeAdders";
import {
  pushLinks,
  pushEntityLinks,
  pushNewsLink,
} from "../HelperFunctions/linkAdders";
import { getJaccardIndexOf } from "../HelperFunctions/getJaccardIndexOf";

const Main = () => {
  //! States
  const [currentEntityType, setCurrentEntityType] = useState(null);
  const [currentEntityQid, setCurrentEntityQid] = useState(null);
  const [currentEntityName, setCurrentEntityName] = useState(null);
  const [clickedNode, setClickedNode] = useState(null);
  const [jaccardThreshold, setJaccardThreshold] = useState(0.4);
  const [searchbarText, setSearchbarText] = useState("");
  const [selectedEntityType, setSelectedEntityType] = useState("");
  const [initialNodeAmount, setInitialNodeAmount] = useState(50);
  const [lastNodes, setLastNodes] = useState([]);
  const [canGoBack, setCanGoBack] = useState(false);
  const [nodes, setNodes] = useState({
    nodes: [],
    links: [],
  });

  //! Fetching Data & Storing in State
  const {
    data: newsData,
    isLoading: isLoadingNewsData,
    error: errorNewsData,
  } = useFetchSearch(
    "http://195.37.233.209:4000/graphql",
    selectedEntityType,
    searchbarText,
    initialNodeAmount
  );

  const {
    data: entityData,
    isLoading: isLoadingEntityData,
    error: errorEntityData,
```

```
  setError: setError,
} = useFetchEntity(
  "http://195.37.233.209:4000/graphql",
  currentEntityType,
  currentEntityQid,
  currentEntityName
);

//! Handler Functions:
const handleBackButton = () => {
  if (lastNodes) {
    setLastNodes((prevState) =>
      prevState.filter((elem, i) => i !== prevState.length - 1)
    );
    setNodes(lastNodes[lastNodes.length - 1]);
    setClickedNode(null);
    if (lastNodes.length <= 1) setCanGoBack(false);
    setError(null);
  }
};

const isEntitySelected = (value) => {
  if (selectedEntityType === value) {
    return true;
  } else return false;
};

const handleEntityTypeSelection = (e) => {
  setSelectedEntityType(e.target.value);
};

const handleJaccardThreshold = (e) => {
  setJaccardThreshold(e.target.value);
};

const handleInitialNodeAmount = (e) => {
  setInitialNodeAmount(e.target.value);
};

const handleNodeClick = (nodeArr) => {
  setClickedNode(nodeArr[0]);
};

//! useEffect Hook - Dependencies (lastNodes)
//! sets Nodes to the last element auf lastNodes Stack
useEffect(() => {
  setNodes(lastNodes[lastNodes.length - 1]);
}, [lastNodes]);

//! useEffect Hook - Dependencies (newsData & jaccardThreshold)
//! Creates initial Nodes and Links
//! Link creation is considering Jaccard Similarity
useEffect(() => {
```

```
    setCanGoBack(false);
    if (newsData) {
      const nodes = [];
      const links = [];
      const uniqueHeadlines = new Set([]);
      const uniqueIDs = new Set([]);

      //* Removing Duplicates from Newsdata
      newsData.forEach((news) => {
        const isDuplicate = uniqueHeadlines.has(news.headline);
        if (!isDuplicate) {
          uniqueHeadlines.add(news.headline);
          uniqueIDs.add(news.id);
          pushNewsNode(nodes, news);
        }
      });

      //* Setting Node Links
      newsData.forEach((cmpNode1) => {
        newsData.forEach((cmpNode2) => {
          if (uniqueIDs.has(cmpNode2.id) && uniqueIDs.has(cmpNode1.id)) {
            if (cmpNode1 !== cmpNode2) {
              let jacPer = getJaccardIndexOf(cmpNode1.per, cmpNode2.per);
              let jacOrg = getJaccardIndexOf(cmpNode1.org, cmpNode2.org);
              let jacLoc = getJaccardIndexOf(cmpNode1.loc, cmpNode2.loc);
              let jaccardIndex = (jacOrg + jacPer + jacLoc) / 3;

              if (jaccardIndex >= jaccardThreshold) {
                pushNewsLink(links, cmpNode1, cmpNode2);
              }
            }
          }
        });
      });
      const currentLastNodes = [];
      currentLastNodes.push({ nodes: nodes, links: links });
      setLastNodes(currentLastNodes);
      setNodes({ nodes: nodes, links: links });
    }
  }, [newsData, jaccardThreshold]);

  //! useEffect Hook - Dependency (entityData)
  //! On Change of entityData --> Updates the "Nodes" State
  //! Displays clicked Node and his children
  useEffect(() => {
    if (entityData) {
      const nodes = [];
      const links = [];
      switch (clickedNode?.__typename) {
        case "org":
          pushEntityMainNode(
            nodes,
            entityData,
```

```
            clickedNode?.__typename,
            "green"
          );
          entityData?.country?.forEach((loc) => {
            pushEntityNode(nodes, loc, "loc", "lightblue");
            pushEntityLinks(links, clickedNode, loc);
          });

          entityData?.ceo?.forEach((ceo) => {
            pushEntityNode(nodes, ceo, "per", "pink");
            pushEntityLinks(links, clickedNode, ceo);
          });

          entityData?.subsidiary?.forEach((org) => {
            pushEntityNode(nodes, org, "org", "lightgreen");
            pushEntityLinks(links, clickedNode, org);
          });
          break;
        case "per":
          pushEntityMainNode(nodes, entityData, clickedNode?.__typename,
"red");
          entityData?.employer?.forEach((org) => {
            pushEntityNode(nodes, org, "org", "lightgreen");
            pushEntityLinks(links, clickedNode, org);
          });
          break;
        case "loc":
          pushEntityMainNode(
            nodes,
            entityData,
            clickedNode?.__typename,
            "blue"
          );
          if (entityData.capital) {
            pushEntityNode(nodes, entityData.capital, "loc", "lightblue");
            pushEntityLinks(links, clickedNode, entityData?.capital);
          }
          if (entityData["head_of_state"]) {
            pushEntityNode(nodes, entityData.head_of_state, "per", "pink");
            pushEntityLinks(links, clickedNode, entityData?.head_of_state);
          }
          if (entityData.highest_judicial_authority) {
            pushEntityNode(
              nodes,
              entityData.highest_judicial_authority,
              "per",
              "pink"
            );
            pushEntityLinks(
              links,
              clickedNode,
              entityData?.highest_judicial_authority
            );
```

```
        }
        break;
      default:
        break;
    }
    setLastNodes((prevState) => [
      ...prevState,
      { nodes: nodes, links: links },
    ]);
    setNodes(() => setNodes({ nodes: nodes, links: links }));
  }
}, [entityData]);

//! useEffect Hook - Dependency (clickedNode)
//! Switches on clicked Node Type
useEffect(() => {
  if (clickedNode) {
    const links = [];
    const nodes = [];
    switch (clickedNode.__typename) {
      // Click on Entity Node
      case "org":
      case "per":
      case "loc":
        setCurrentEntityType(clickedNode.__typename);
        setCurrentEntityQid(clickedNode.id);
        setCurrentEntityName(clickedNode.name);
        break;
      // Click on News Node
      case "News":
        setCanGoBack(true);
        pushNewsNode(nodes, clickedNode);

        clickedNode?.org?.forEach((orgNode) => {
          if (orgNode.sameAs) {
            pushNodes(nodes, orgNode, "org", "lightgreen");
            pushLinks(links, clickedNode, orgNode);
          }
        });

        clickedNode?.per?.forEach((perNode) => {
          if (perNode.sameAs) {
            pushNodes(nodes, perNode, "per", "red");
            pushLinks(links, clickedNode, perNode);
          }
        });

        clickedNode?.loc?.forEach((locNode) => {
          if (locNode.sameAs) {
            pushNodes(nodes, locNode, "loc", "lightblue");
            pushLinks(links, clickedNode, locNode);
          }
        });
```

```
          setLastNodes((prevState) => [
            ...prevState,
            { nodes: nodes, links: links },
          ]);
          setNodes(() => setNodes({ nodes: nodes, links: links }));
          break;
      }
    }
  }, [clickedNode]);

  return (
    <div>
      <Search
        canGoBack={canGoBack}
        handleBackButton={handleBackButton}
        searchbarText={searchbarText}
        changeSearchbarText={(searchbarText) => setSearchbarText(searchbar-
Text)}
        isEntitySelected={isEntitySelected}
        selectedEntityType={selectedEntityType}
        handleEntityTypeSelection={handleEntityTypeSelection}
        handleJaccardThreshold={handleJaccardThreshold}
        jaccardThreshold={jaccardThreshold}
        initialNodeAmount={initialNodeAmount}
        handleInitialNodeAmount={handleInitialNodeAmount}
      />
      {isLoadingNewsData && !errorNewsData && <p>Loading...</p>}
      {!isLoadingNewsData && (
        <div className="error">
          {errorEntityData && <p>{errorEntityData}</p>}
          {errorNewsData && <p>{errorNewsData}</p>}
        </div>
      )}

      {!isLoadingNewsData && (
        <ForceGraph2D
          height={850}
          width={1800}
          nodeColor={"color"}
          nodeLabel={"name"}
          nodeVal={"sizeInPx"}
          linkColor={"red"}
          backgroundColor={"white"}
          linkWidth={5}
          linkDirectionalParticles={3}
          graphData={nodes}
          nodeCanvasObjectMode={() => "after"}
          nodeCanvasObject={(node, ctx, globalScale) => {
            const label = node.name;
            const fontSize = node.__typename === "News" ? 4 : 3;
            ctx.font = `${fontSize}px Sans-Serif`;
            ctx.textAlign = "center";
            ctx.textBaseline = "middle";
```

```
            ctx.fillStyle = node.__typename === "News" ? "black" : "black";
            if (node.__typename === "News") {
              const lineHeight = fontSize * 2;
              const lines = label.split(",");
              let x = node.x;
              let y = node.y - lineHeight;
              for (let i = 0; i < lines.length; ++i) {
                ctx.fillText(lines[i], x, y);
                y += lineHeight / 1.2;
              }
            } else if (globalScale >= 3.5) {
              ctx.fillText(label, node.x, node.y + 2.5);
            }
          }}
          onNodeClick={(node) => {
            if (node.__typename !== "News") {
              const nodeAsArr = [node];
              handleNodeClick(nodeAsArr);
            } else
              handleNodeClick(newsData.filter((news) => news.id ===
node.id));
          }}
        />
      )}
    </div>
  );
};
export default Main;
```

7.1.4 Search.js

```
import { IoArrowBackOutline } from "react-icons/io5";

const Search = ({
  searchbarText,
  changeSearchbarText,
  initialNodeAmount,
  handleInitialNodeAmount,
  jaccardThreshold,
  handleJaccardThreshold,
  handleEntityTypeSelection,
  isEntitySelected,
  canGoBack,
  handleBackButton,
}) => {
  const handleSubmit = (e) => {
    e.preventDefault();
  };
  return (
    <div className="search">
      <form onSubmit={handleSubmit}>
        <input
```

```
          className="textInput"
          type="text"
          value={searchbarText}
          placeholder="Search for some Person, Organization or Country"
          onChange={(e) => {
            changeSearchbarText(e.target.value);
          }}
        />
        <div className="radio">
          <label htmlFor="org" id="radioOrg">
            Organization
          </label>
          <input
            name="entity-type"
            checked={isEntitySelected("org")}
            onChange={(e) => handleEntityTypeSelection(e)}
            type="radio"
            value="org"
          />
        </div>
        <div className="radio">
          <label htmlFor="loc" id="radioLoc">
            Location
          </label>
          <input
            name="entity-type"
            checked={isEntitySelected("loc")}
            onChange={(e) => handleEntityTypeSelection(e)}
            type="radio"
            value="loc"
          />
        </div>
        <div className="radio">
          <label htmlFor="per" id="radioPer">
            Person
          </label>
          <input
            name="entity-type"
            checked={isEntitySelected("per")}
            onChange={(e) => handleEntityTypeSelection(e)}
            type="radio"
            value="per"
          />
        </div>
        <div className="jaccardSlider">
          <label htmlFor="jaccardSlider">
            Jaccard Threshold: {jaccardThreshold}
          </label>
          <input
            type="range"
            name="jaccardSlider"
            min={0.1}
            max={1.0}
```

```
            step={0.1}
            value={jaccardThreshold}
            onChange={(e) => handleJaccardThreshold(e)}
          />
        </div>
        <div className="jaccardSlider">
          <label htmlFor="nodeAmount">Node Amount:</label>
          <input
            id="inputNumber"
            type="number"
            name="nodeAmount"
            min={10}
            max={1000}
            value={initialNodeAmount}
            onChange={(e) => handleInitialNodeAmount(e)}
          />
        </div>
        {canGoBack && (
          <h1
            id="back-btn"
            onClick={() => {
              handleBackButton();
            }}
          >
            <IoArrowBackOutline />
          </h1>
        )}
      </form>
    </div>
  );
};

export default Search;
```

7.1.5 useFetchEntity.js

```
import { useState, useEffect } from "react";
import { getEntityQuery } from "../HelperFunctions/getEntityQuery";
import { getDataUnwrapper } from "../HelperFunctions/getDataUnwrapper";

const useFetchEntity = (url, entityType, qid, entityName) => {
  const [data, setData] = useState(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);
  const QUERY = getEntityQuery(entityType, qid);

  useEffect(() => {
    setIsLoading(true);
    if (entityType && qid) {
      fetch(url, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
```

```
        body: JSON.stringify({ query: QUERY }),
      })
        .then((response) => response.json())
        .then((data) => {
          setData(data["data"][getDataUnwrapper(entityType)]);
          setIsLoading(false);
          setError(null);
        })
        .catch((err) => {
          setIsLoading(false);
          setError("Cannot find any Data concerning the Entity: " + enti-
tyName);
        });
    }
  }, [url, entityType, qid, entityName]);

  return { data, isLoading, error, setError };
};

export default useFetchEntity;
```

7.1.6 useFetchSearch.js

```
import { useState, useEffect } from "react";

const useFetchSearch = (url, entityType, entity, nodeAmount) => {
  const [data, setData] = useState([]);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);
  const QUERY = `query {
    newsSearch(limit: ${nodeAmount}, entity: "${entityType}", keywords:
"${entity}"){
      __typename
      id
      headline
      org{
        name
        sameAs
      }
      per{
        name
        sameAs
      }
      loc{
        name
        sameAs
      }

    }
  }`;

  const getData = () => {
```

```
    setIsLoading(true);
    fetch(url, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ query: QUERY }),
    })
      .then((response) => response.json())
      .then((data) => {
        setData(data.data.newsSearch);
        setIsLoading(false);
        setError(null);
      })
      .catch((err) => {
        setIsLoading(false);
        setError(err.message);
      });
  };

  useEffect(() => {
    const timer = setTimeout(() => {
      getData();
    }, 500);

    return () => clearTimeout(timer);
  }, [entity, entityType, nodeAmount]);

  return { data, isLoading, error };
};

export default useFetchSearch;
```

7.1.7 nodeAdders.js

```
function pushNewsNode(arrToPushTo, data) {
  arrToPushTo.push({
    id: data?.id,
    name: data?.headline,
    __typename: data?.__typename,
    color: "rgba(233, 245, 10, 0.8)",
    sizeInPx: 6,
  });
}

function pushEntityNode(arrToPushTo, data, entityTypeStr, nodeColorStr) {
  arrToPushTo.push({
    id: data.qid,
    name: data.label,
    __typename: entityTypeStr,
    color: nodeColorStr,
    sizeInPx: 5,
  });
}
```

```
function pushEntityMainNode(arrToPushTo, data, entityTypeStr, nodeColorStr) {
  arrToPushTo.push({
    id: data.qid,
    name: data.label,
    __typename: entityTypeStr,
    color: nodeColorStr,
    sizeInPx: 6,
  });
}

function pushNodes(arrToPushTo, data, entityTypeStr, nodeColorStr) {
  arrToPushTo?.push({
    id: data?.sameAs,
    name: data?.name,
    __typename: entityTypeStr,
    color: nodeColorStr,
    sizeInPx: 5,
  });
}

export { pushNewsNode, pushNodes, pushEntityNode, pushEntityMainNode };
```

7.1.8 linkAdders.js

```
function pushLinks(arrToPushTo, source, target) {
  arrToPushTo.push({
    source: source?.id,
    target: target?.sameAs,
  });
}

function pushNewsLink(arrToPushTo, source, target) {
  arrToPushTo.push({
    source: source?.id,
    target: target?.id,
  });
}

function pushEntityLinks(arrToPushTo, source, target) {
  if (source && target) {
    arrToPushTo.push({
      source: source?.id,
      target: target?.qid,
    });
  }
}

export { pushLinks, pushEntityLinks, pushNewsLink };
```

7.1.9 getJaccardIndex.js

```
function intersectionOf(set1, set2) {
  let intersection = set1.filter((a) => set2.some((b) => a.name === b.name));
  return intersection.length;
}

function unionOf(set1, set2) {
  let union = [...set2, ...set1].filter(
    (
      (set) => (o) =>
        set.has(o.name) ? false : set.add(o.name)
    )(new Set())
  );
  return union.length;
}

function getJaccardIndexOf(set1, set2) {
  return intersectionOf(set1, set2) / unionOf(set1, set2);
}

export { getJaccardIndexOf };
```

7.1.10 getEntityQuery.js

```
const getEntityQuery = (entityTypeStr, qid) => {
  switch (entityTypeStr) {
    case "org":
      return `query companyByQid {
        companyByQid(qid: "${qid}") {
          type
          qid
          label
          description
          image
          country {
            qid
            label
          }
          ceo {
            qid
            label
          }
          subsidiary {
            qid
            label
          }
      }
      }`;
    case "loc":
      return `query LocationByQid {
        locationByQid(qid: "${qid}") {
```

```
          qid
          type
          label
          description
          head_of_state{
            qid
            label
          }
          capital{
            qid
            label
          }
          highest_judicial_authority{
            qid
            label
          }
        }
      }`;
    case "per":
      return `query PersonByQid {
        personByQid(qid: "${qid}") {
          qid
          type
          label
          description
          date_of_birth
          date_of_death
          place_of_birth {
            qid
            label
          }
        country_of_citizenship {
            qid
            label
          }
          employer {
            qid
            label
          }
      }
      }`;
  }
};

export { getEntityQuery };
```

7.1.11 getDataUnwrapper.js

```
function getDataUnwrapper(entityType) {
  switch (entityType) {
    case "org":
      return "companyByQid";
    case "per":
      return "personByQid";
    case "loc":
      return "locationByQid";
    default:
      break;
  }
}

export { getDataUnwrapper };
```

Literaturverzeichnis

[1] Vgl. Dykes, B. (2020).
Effective Data Storytelling: How to Drive Change with Data, Narrative and Visuals (1. Aufl.). Wiley.

[2] Vgl. Dykes, B (2020).

[3] Adrian M (2011) It's going mainstream, and it's your next opportunity. Teradata Magazine Online. Abgerufen am 19. Juni 2022, von http://www.teradatamagazine.com/v11n01/Features/Big-Data/.

[4] Merriam-Webster (o. D.) *graph*. The Merriam-Webster.Com Dictionary. Abgerufen am 21. Juni 2022, von https://www.merriam-webster.com/dictionary/graph

[5] Matlab (2022). Bar .Abgerufenam 21. Juni 2022, von https://www.mathworks.com/help/examples/graphics/win64/SingleDataSeriesExample_01.png

[6] Matlab (2022). Pie. Abgerufen am 21. Juni 2022, von https://www.mathworks.com/help/examples/graphics/win64/CreatePieChartExample_01.png

[7] Brath. (2015). *Graph Analysis and Visualization: iness Opportunity in Linked Data* (1. Aufl.). https://doi.org/10.1002/9781119183662

[8] Wikipedia contributors. (2022, 1. Mai). *Knowledge graph*. Wikipedia. Abgerufen am 21. Juni 2022, von https://en.wikipedia.org/wiki/Knowledge_graph

[9] Vgl. Fundamentals, O. (2020, 6. April). *What is a Knowledge Graph?* Ontotext. Ab gerufen am 22. Juni 2022, von https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/

[10] Vgl. L. He et al. (2022), "Neurally-Guided Semantic Navigation in Knowledge Graph,"
in IEEE Transactions on Big Data, vol. 8, no. 3, pp. 607-615, 1 June 2022,
doi: 10.1109/TBDATA.2018.2805363.

[11] Vgl. L. He et al (2022)

[12] Wikipedia-Autoren. (2016, 6. September). *React*. React. Abgerufen am 22.Juni 2022, von
https://de.wikipedia.org/wiki/React

[13] Asturiano, V. (o. D.). *GitHub - vasturiano/react-force-graph: React component for 2D,*
3D, VR and AR force directed graphs. GitHub.
Abgerufen am 22. Juni 2022, von
https://github.com/vasturiano/react-force-graph

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht. Die vorgelegte Arbeit hat weder in der gegenwärtigen noch in einer anderen Fassung schon einem anderen Fachbereich der Hochschule Ruhr West oder einer anderen wissenschaftlichen Hochschule vorgelegen.

Bottrop, 01.07.2022

Alperen Dagli