Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Ego-Graph-basierte visuelle Exploration semantischer Wissensgraphen

Jan Robert Gruhnert

Studiengang:	Softwaretechnik
Prüfer/in:	Dr. Steffen Koch
Betreuer/in:	Samuel Beck, M.Sc. Max Franke, M.Sc.
Beginn am:	23. März 2022
Beendet am:	23. September 2022

Kurzfassung

Wissensgraphen erfreuen sich in der Forschung großer Beliebtheit, da sie Weltwissen repräsentieren. Um diese riesigen Strukturen für den Menschen verständlich aufzubereiten ist ein weit verbreiteter Ansatz, der des Knoten-Kanten Diagramms. Diese Diagramme werden häufig durch kräfte-basierte Layoutalgorithmen generiert. Ein Nachteil dabei, ist die zunehmende Unübersichtlichkeit des Diagramms, bei immer größer werdenden Graphen. Dabei gibt es durchaus Szenarien, in denen man nicht am gesamten Graphen interessiert ist, sondern nur an einzelnen Knoten und deren Nachbarn. In dieser Arbeit wird ein inkrementelles Verfahren vorgestellt, in welchem nicht der gesamte Graph visualisiert wird, sondern erst mal nur ein ausgewählter Startknoten. Ausgehend von diesem Startknoten ist es möglich, Nachbarknoten manuell zu expandieren und der Visualisierung hinzuzufügen. Die Auswahl der Knoten wird dabei den Nutzer*innen überlassen und findet mit Hilfe eines Knoten-basierten Menüs statt, in dem Nachbarknoten nach Kategorien ausgewählt werden können. Für die Umsetzung dieser Ego-Graph-basierten Exploration wurden mehrere verwandte Arbeiten untersucht, die eine ähnliche Herangehensweise vorgestellt haben. Daraus wurde dann experimentell ein eigener webbasierter Prototyp entworfen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

D3 Javascript Bibliothek D3.js. 14

DOI Degree of Interest. 15

DOM Document Object Model. 39

Ego-Graphen Egozentrische Graphen. 17

LA Layoutalgorithmus. 13

SVG Scaleable vector grafics. 39

1. Einleitung

In vielen Bereichen des täglichen Lebens und auch der Wissenschaft helfen uns Wissensgraphen, komplexe Zusammenhänge zu erkennen und zu verstehen. Oft werden diese Wissensgraphen als Knoten-Kanten-Diagramme dargestellt. Diese Art der Darstellung ist sehr vorteilhaft, um Relationen zwischen Entitäten in eine für den Menschen anschauliche Form zu bringen. Für Wissensgraphen mit wenigen Entitäten ist die Umsetzung eines Knoten-Kanten-Diagramms manuell noch sehr gut zu erreichen. Allerdings sind Wissensgraphen oft in Bereichen von Interesse, in denen sehr große Datenmengen auftreten. So werden diese z.B. bei der Analyse des World Wide Web eingesetzt [New18]. Wissensgraphen von dieser Komplexität lassen sich nur mit Hilfe von Algorithmen in eine anschauliche Form bringen. Hierzu werden Layoutalgorithmen benutzt, die ein Knoten-Kanten-Diagramm erstellen. Ein optimales Knoten-Kanten-Diagram hat keine Überlappungen von Knoten, auch sollen sich die Kanten möglichst wenig schneiden. Eben dieser letzte Aspekt ist keine triviale Anforderung an den Algorithmus. Je nach Topologie ist dies im zwei dimensionalen Raum auch gar nicht möglich. Mit kräftebasierten Layoutalgorithmen wie dem von Fruchterman und Reingold [FR91], wird versucht ein optimales Knoten-Kanten-Diagram zu erzeugen. Diese Art von Algorithmen nennt man kräftebasiert, da sie physikalische Kräfte zwischen einzelnen Knoten und Kanten simulieren und so ein Layout generieren. Üblicherweise werden abstoßende Kräfte zwischen Knoten simuliert. Sind zwei Knoten durch eine Kante verbunden, dann werden anziehende Kräfte simuliert. Als Ergebnis erhofft man sich dadurch räumliche Nähe von verbundenen Knoten und den umgekehrten Effekt bei nicht verbundenen Knoten, da die physikalische Simulation die Knoten in einen minimalen Energiezustand bringt.

Allerdings hat diese Art von Layoutalgorithmus (LA) auch ihre Schwächen. Zum einen ist die Simulation eines solchen Systems sehr rechenintensiv. Das liegt daran, dass die Berechnung von Kräften zwischen allen Knoten der Knotenmenge V eine Zeitkomplexität von $O(|V|^2)$ aufweist. Diese Zeitkomplexität lässt sich beispielsweise auf $O(|V| \log |V|)$ verkürzen, indem man die Barnes-Hut-Approximation benutzt [BH86]. Zum anderen ist der minimale Energiezustand, der durch solch einen kräftebasierten LA erreicht wird, nur ein pseudooptimaler Zustand. Denn es ist durchaus möglich, dass die Simulation in einem lokalen Minimum endet und somit das globale Minimum nicht erreicht. Unter anderem beschäftigt sich Hu [Hu05] mit der Lösung dieses Problems.

Die beiden genannten Probleme nehmen an Relevanz zu, wenn die Größe des Graphen zunimmt. Oft ist es aber gar nicht notwendig den gesamten Graphen anzuzeigen. Wenn Nutzer*innen nur an bestimmten Knoten interessiert sind, dann ist es sinnvoll, nur einen kleinen Teil des Graphen anzuzeigen und den redundanten Teil des Graphen wegzulassen oder auszublenden. Auf diese Weise umgeht man die Probleme, die die Größe eines gesamten Graphen mit sich bringt.

In dieser Arbeit verfolge ich den eben genannten Ansatz und stelle ein Konzept zur visuellen Exploration eines Wissensgraphen vor. Die Exploration basiert auf dem Modell der inkrementellen Graphexpansion, mit welcher Nutzer*innen die Möglichkeit gegeben wird, adjazente Knoten aufzudecken. Diese können dann dem visualisierten Graphen hinzugefügt werden. Diese Exploration

findet auf einem Ego-Graphen statt. Sie geht damit von einem bestimmten Startknoten aus. Ein besonderes Augenmerk liegt dabei darauf, wie man Nutzer*innen eine Übersicht über aufklappbare adjazente Knoten gibt und wie eine Vorschau über diese Knoten umgesetzt wird. Ein weiterer wichtiger Aspekt dieser Arbeit ist die Umsetzung des inkrementellen Layouts selber. Hierbei soll das Hinzufügen und Entfernen von Knoten mit minimalen Ortsverschiebungen der bereits im Graphen befindlichen Knoten geschehen. Wie dies umgesetzt werden kann, ist ebenfalls eine relevante Forschungsfrage, mit der ich mich in dieser Arbeit auseinandersetze.

Für die Entwicklung des hier vorgestellten Prototypen habe ich relevante Forschungsarbeiten analysiert und zur Ideenfindung genutzt. Die daraus enstandenden Entwürfe habe ich in Hinblick auf die Fragestellung evaluiert und den am besten funktionierende Entwurf letzten Endes im finalen Prototypen zur Ego-Graph-basierten visuellen Exploration umgesetzt. Da ich zur Umsetzung des Prototypen moderne Webtechnologien wie Javascript Bibliothek D3.js (D3)[1], TypeScript[2] und WebPack[3] eingesetzt habe, wird die Anwendung im Internetbrowser ausgeführt.

[1] https://d3js.org/
[2] https://www.typescriptlang.org/
[3] https://webpack.js.org/

2. Verwandte Arbeiten

Es gibt bereits einige Forschungsarbeiten, die sich mit der dynamischen Anpassung von bereits bestehenden Layouts befassen. In diesen Arbeiten geht es darum, Änderungen in einem Graphen im Layout widerzuspiegeln, ohne dessen Layout komplett neu zu berechnen. So untersuchen Misue et al. [MELS95], wie das vorhandene mentale Bild des Knoten-Kanten-Diagramms bei Nutzer*innen erhalten bleibt, während Änderungen am Diagramm vorgenommen werden. Die Autoren liefern hierzu einen Algorithmus, der die relativen Positionen der bereits vorhandenen Knoten zueinander beibehält. Lin et al. [LLY11] nutzen ebenfalls das Konzept des mentalen Bildes, um ein dynamisches Layout mittels Simulated Annealing zu generieren. Hier werden Änderungen von Knotenpositionen in der Layoutberechnung anhand einer Kostenfunktion bewertet. Die Kostenfunktion berechnet die Kosten, die diese Änderung auf das mentale Bild der Nutzer*innen hat und verwirft jede Änderung, bei der ein festgelegter Schwellenwert überschritten wird.

Andere Publikationen wiederum beschäftigen sich speziell mit kräfte-basierten Layoutanpassungen in dynamischen Graphen [CCM15; SMK13]. Die Arbeit von Crnovrsanin et al. [CCM15] stellt vor, wie man neue Knoten im Layout platzieren kann. Außerdem stellen sie ein kräfte-basierten LA vor, der den Knoten nur dann einen neuen Platz im Layout zuweist, wenn sich diese in einem hohen Energiezustand befinden. Das vorgestellte Konzept von Steiger et al. [SMK13] erlaubt es wiederum, allen Knoten einen neuen Platz zu erreichen. Dafür werden die abstoßenden und anziehenden Kräfte-Algorithmen von Fruchterman und Reingold [FR91] genutzt, sie fügen für einzelnen Knoten jeweils eine Masse in die Berechnung mit ein, so dass Knoten mit höherer Masse weniger von den Kräften beeinflusst werden als solche mit niedriger Masse. Die Positionsänderung der Knoten zum vorherigen Layout lässt sich beeinflussen, indem man Knoten, welche sich schon länger im Layout befinden, eine höhere Masse zuschreibt. Des Weiteren stellen Steiger et al. [SMK13] eine Technik zum erstmaligen Platzieren neuer Knoten im Layout vor. Außerdem werden bereits Knoten in die Simulation geladen, die im Layout noch nicht vorhanden sind. Diese Knoten werden durch eine Degree of Interest (DOI)-Funktion ermittelt und sollen helfen das Layout stabil zu halten.

Andere Konzepte verzichten auf kräfte-basierte Layoutalgorithmen und verfolgen einen anderen Ansatz. So nutzt *TreePlus* [LPP+06], wie der Name es schon andeutet, eine Baumstruktur um die Entitäten eines Graphen darzustellen. Diese Art der Darstellung hat den Vorteil, dass das Layout übersichtlich und leicht zu expandieren ist. Außerdem ist die Laufzeit der Layoutberechnung klein, da hier keine aufwändige Simulation von Kräften benutzt wird. Nachteile dieser Darstellung ergeben sich dann, wenn Graphen betrachtet werden, die in ihrer ursprünglichen Form nicht hierarchisch sind. In diesem Fall werden viele Kanten, welche nicht in diese Baumstruktur passen, ausgeblendet. *TreePlus* löst das Problem, dadurch dass alle Nachbarknoten eines fokussierten Knotens hervorgehoben werden.

Über diese so eben beschriebenen Arbeiten hinaus, gibt es weitere Veröffentlichungen, die sich mit den Problemen der dynamischen oder inkrementellen Layouterstellung auseinandergesetzt haben. Diese Arbeiten decken nur einen Teil der hier aufgeworfenen Fragestellungen ab und lassen somit

Potential für weitere Forschung. Die Arbeit von van Ham et al. [HP09] beschäftigt sich mit der Frage, wie man mit Hilfe einer DOI-Funktion die Relevanz von Knoten in einem Graphen bestimmen kann. Mit Berücksichtigung von Metadaten aus dem Graphen und der bekannten Interessen von Nutzer*innen wird außerdem eine DOI-Funktion vorgestellt. Die daraus ermittelten, relevanten Knoten werden dann einem Layout hinzugefügt und es wird den Nutzer*innen die Möglichkeit gegeben, diese Knoten weiter zu expandieren.

Bei der Exploration des Graphen spielt nicht nur das Expandieren von Knoten eine Rolle, auch andere Interaktions-Möglichkeiten, sind für die Informationsgewinnung hilfreich. Pienta et al. [PAKC15] geben eine umfassende Zusammenstellung verschiedener Möglichkeiten, mit einem Graphlayout zu interagieren. Hier verweise ich auf unterschiedliche Navigationsfunktionen, welche die Orientierung im Layout erleichtern können. Außerdem werden auch verschiedene Visualisierungstechniken von Graphen diskutiert. Für sehr komplexe Graphen kann es beispielsweise sinnvoll sein, mehrere Knoten im Layout zu aggregieren, wenn diese gleiche Eigenschaften teilen.

3. Grundlagen

Im Verlauf der Arbeit werden einige Begriffe und Konzepte verwendet, die zum Verständnis der Arbeit wichtig sind. Sie werden in diesem Kapitel erläutert.

3.1. Graphen

Ein Graph G wird durch ein Tupel (V, E) repräsentiert, welches aus der Knotenmenge V und der Kantenmenge $E \subseteq \{(a, b), \quad a, b \in V\}$ gebildet wird. Ein Graph heißt ungerichtet, wenn $\forall e \in E$ gilt: $e = \{(a, b) = (b, a), \quad a, b \in V\}$. Ein Graph ist zusammenhängend, wenn für jedes beliebiges Paar aus Knoten gilt, dass sie direkt oder indirekt durch Kanten verbunden sind.

Egozentrische Graphen (Ego-Graphen) Ego-Graphen sind Graphen, die aus der Perspektive eines einzelnen zentralen Knoten betrachtet werden. Im Fall dieser Arbeit startet die Exploration des Graphen auch in einer Ego-Graphen basierten Weise, die von einem Startknoten ausgeht. Von diesem Startknoten aus können Nutzer*innen Nachbarknoten explorieren.

Startknoten Als Startknoten bezeichnet man den Knoten, von dem aus die Ego-basierte Exploration startet, und der damit zum Ursprung der Ego-basierten Exploration wird.

Hop Mit einem Hop wird eine Kantenüberquerung gemeint. Alle Knoten, die sich eine Kante mit dem Ausgangsknoten teilen, sind einen Hop von diesem entfernt, da sie jeweils durch eine Kantenüberquerung erreicht werden können. Zwei-Hop Knoten sind alle Knoten, welche durch genau zwei Kantenüberquerungen erreicht werden können. Das gleiche Prinzip lässt sich auf beliebig viele Hops anwenden. Mit einem x-Hop Radius sind alle Knoten gemeint, die mit mindestens x-Hops von einem Ausgangsknoten oder einer Gruppe von Ausgangsknoten erreichbar sind.

Community Unter einer Community versteht man eine Knotenmenge, die eine höhere Kantendichte untereinander aufweisen, als im Vergleich zum Rest des Graphen. Schwierigkeiten beim Finden von Communities bereitet der Umstand, dass es keine genaue Definition einer Community gibt. Eine hohe Kantendichte beispielsweise ist eine ungenaue Anforderung, die immer in Relation zum betrachteten Graphen steht. Es gibt aber einige Ansätze solche Communitys im Graphen zu finden. Ein Paar dieser Ansätze sind im Newmans Buch [New18] aufgelistet. So gibt es beispielsweise heuristische Algorithmen oder statistische Algorithmen mit denen Knoten in Communities eingeteilt werden können.

3.2. Graphlayout

Unter einem Layout versteht man die räumliche Anordnung der Knoten eines Graphen. Ein Layout L bildet also jeden Knoten in einem Graphen auf eine Koordinate in einem Raum $\mathbb{R}^n$ ab. Diese Abbildung von Knoten auf Koordinaten wird hier als Konfiguration bezeichnet. Dieser Arbeit beschäftigt sich ausschließlich mit Layouts im zweidimensionalen Raum.

In dieser Arbeit wird zwischen zwei Layouttypen unterschieden. Zum Einen gibt es das statische Layout. Unter einem statischen Layout versteht man ein Layout, welches eine feste Konfiguration hat. Zum Anderen gibt es das dynamische Layout, um das es in dieser Arbeit hauptsächlich geht. Ein dynamisches Layout fügt dem statischen Layout noch eine zeitliche Komponente zur Konfiguration hinzu. Damit ist jede Konfiguration des Layouts eine Momentaufnahme und ist abhängig vom betrachteten Zeitpunkt. Ein dynamisches Layout besteht also aus einer zeitlichen Abfolge von Konfigurationen. In dieser Arbeit wird ein dynamisches Layout als $\{L_n \mid n \in \mathbb{N}\}$ dargestellt, wobei n die jeweils aktuelle Konfiguration beschreibt. Ein Konfigurationswechsel von L_n zur Nachfolgekonfiguration wird dann als $L_n \rightarrow L_{n+1}$ dargestellt. Im Laufe dieser Arbeit wird auch oft von einem inkrementellen Layout gesprochen. Dabei handelt es sich um ein dynamisches Layout mit besonderer Eigenschaft. In einem inkrementellen Layout werden bei einem Konfigurationswechsel $L_n \rightarrow L_{n+1}$ neue Knoten in die Konfiguration eingefügt. Die Anzahl der Knoten im Layout wächst mit der Zeit an.

Fokus dieser Arbeit sind kräftebasierte Layoutalgorithmen, die verschiedene Kräfte zwischen Knoten simulieren, dadurch Knoten bewegen und auf diese Weise ein Layout erzeugen. An Stelle von kräftebasierten Layoutalgorithmen wird deshalb auch die Bezeichnung Simulation verwendet.

3.3. Verwendete Datensätze

In der Entwicklung des Prototypen habe ich zwei Graphen verwendet, mit denen ich die Exploration getestet und bewertet habe.

3.3.1. Zufallsgraph

Am Anfang der Entwicklung habe ich ein Zufallsgraph $G = \{(V, E), \quad |V| = |E| = 100\}$ verwendet, dessen Kanten zufällig generiert werden. Der Graph hat also ein 1:1 Verhältnis von Knoten und Kanten und hat damit eine sehr geringe Dichte. Der Graph muss nicht zwingend zusammenhängend sein. Jeder Knoten hat einen von vier Typen: *Objekt*, *Ort*, *Person* oder *Institution*. Der Graph ist ungerichtet. Wenn in dieser Arbeit von einem Zufallsgraphen gesprochen wird, dann meine ich einen beliebigen Graphen, der die genannten Eigenschaften aufweist.

3.3.2. Beispielgraph

Im späteren Entwicklungsprozess wird ein Beispielgraph verwendet, der dem späteren Anwendungszweck des Prototypen entspricht. Der Beispielgraph wird mittels Daten von WikiData.org[1] generiert und umfasst drei verschiedene Entitätstypen. Dies sind Entitäten vom Typ *Person*, *Objekt* oder *Ort*, sie werden als Knoten im Graphen dargestellt. Die Kanten im Graphen sind die Relationen zwischen den einzelnen Entitäten und sie sind gerichtet. Die Art der Relation ist als Label der Kante gespeichert. Ein Beispiel wäre die Beziehung *liegt in* zwischen den zwei Entitäten des Typus *Ort*: *Berlin* und *Deutschland*. Außerdem gilt für jeden Knoten, dass er maximal fünf Hops vom Startknoten *Wolfgang Amadeus Mozart* entfernt ist. Damit umfasst der Graph 7416 Knoten und 19364 Kanten.

[1] https://www.wikidata.org/wiki/Wikidata:Main$_{Page}$

4. Layoutentwurf

Jedes Layout ist ein zentraler Bestandteil für die visuelle Exploration eines Graphen. Wie bereits erwähnt, gibt es verschiedene Möglichkeiten, ein Layout für einen Graphen zu generieren. Im Folgendem bewerte ich Techniken und Algorithmen nur in Bezug auf die inkrementelle Layoutentwicklung, da dies Ziel der Arbeit ist. Das heißt, Knoten und Kanten werden nach und nach dem Graphen hinzugefügt. Ein Wissen über den gesamten Graphen wird für den Algorithmus nicht vorausgesetzt. Diese Einschränkung bringt einige Nachteile mit sich. Denn die Layoutberechnung bezieht nur vorhandene und neu hinzukommende Knoten und Kanten mit ein. So kann es vorkommen, dass Knoten keine Verbindung haben und somit im Layout räumlich weit auseinander platziert werden, aber in der nächsten Erweiterung des Graphen eine Verbindung über einen neuen Knoten entsteht. Dies führt dazu, dass sich diese Knoten dann wieder annähern müssen. Diese Annäherung ist besonders schwer, wenn ein anderer Teil des Graphen dazwischen liegt. Außerdem werden die Probleme des vorherigen Layouts übernommen. Wenn die Anordnung der Knoten im Layout L_n nicht optimal ist, dann wird das Problem ins nächste Layout L_{n+1} übertragen. In diesem Kapitel stelle ich verschiedene Techniken aus verwandten Arbeiten vor und diskutiere, wie man die geschilderten Probleme minimieren kann.

4.1. Eigenschaften eines guten Layouts

Um die folgenden Techniken beurteilen zu können, muss festgelegt werden, welche Ansprüche der zu entwickelnde LA erfüllen soll. Da es sich um einen dynamischen LA handelt, betrachte ich hauptsächlich die Veränderung der Konfiguration L_n zur nachfolgenden Konfiguration L_{n+1}. Die Knoten der Konfiguration L_n sind nicht dieselben, wie in L_{n+1}. Grundsätzlich kann man die Knoten aus L_{n+1} in drei Kategorien unterteilen. Einmal gibt es die Knoten, die sowohl in L_n als auch in L_{n+1} enthalten sind. Dann gibt es die Knoten, die in L_{n+1} nicht mehr enthalten sind. Das heißt diese Knoten wurden aus dem Layout entfernt. Als letztes gibt es die neuen Knoten, welche erst in L_{n+1} hinzugefügt werden und vorher nicht im Layout sind. Für Knoten, die im Layout verweilen, ist die Aufgabe klar. Diese Knoten sollen sich in ihrer Position möglichst wenig ändern, um das mentale Bild, welches bei den Nutzer*innen entstanden ist, zu erhalten. Misue et al. [MELS95] leisten eine wertvolle Vorarbeit, weil sie die ungenaue Anforderung präzise definieren: Um das mentale Bild zu erhalten, ist es von Vorteil, dass Knoten relativ zueinander ihre Position beibehalten. Logischerweise ist das bei Knoten, die das Layout verlassen, oder neu hinzukommen, nicht möglich. Deshalb sollten diese Knoten so hinzugefügt oder entfernt werden, dass Knoten, welche im Layout bleiben, nicht zu stark positionell beeinflusst werden. Zusätzlich sind die grundlegenden Anforderungen an ein Layout zu beachten: Knoten sollen sich nicht überlappen und Kanten sollen sich möglichst wenig schneiden. Außerdem sollen Knoten, die eine Kante teilen, eine räumliche Nähe aufweisen. Diese Kriterien tragen zur Übersichtlichkeit des Layouts bei und unterstützen damit die Orientierung der Betrachtenden. Die folgenden Techniken sollen nun daran gemessen werden, inwiefern sie die hier erörterten Anforderungen an ein gutes Layout erfüllen.

4.2. Vorladen von Knoten

Ein wichtiger Bestandteil des Layoutentwurfs liegt in der Vorlade-Strategie. Das bedeutet, dass Nutzer*innen nur einen Teil des Layouts angezeigt bekommen. Für jeden Knoten, welcher von Nutzer*innen hinzufügt wird, befinden sich Nachbarknoten in der Simulation, die nicht Teil der Visualisierung sind. Dadurch werden Knoten in die Layoutberechnung mit einbezogen, die mit einer hohen Wahrscheinlichkeit von Nutzer*innen zum sichtbaren Graphen hinzugefügt werden könnten. Der Einfachheit halber werden die Knoten, die sowohl für Nutzer*innen sichtbar sind, als auch in der Simulation sind, als *sichtbare* Knoten bezeichnet und die Knoten, welche nur in der Simulation sind, als *unsichtbare* Knoten betitelt.

Sichtbare Knoten sind durch das Vorladen nicht mehr dem direkten Einfluss von neuen Knoten ausgesetzt, da sichtbare Knoten keine Kante mit neuen Knoten teilen. Diese neuen Knoten können sich in der Simulation auch freier bewegen, da sie unsichtbar hinzugefügt werden und somit nicht das mentale Bild der Nutzenden beeinflussen. Allerdings gilt das nur bis zu einem gewissen Grad, da Knoten, die nicht direkt miteinander verbunden sind, trotzdem Einfluss aufeinander nehmen. Diesen Spielraum, den man dadurch erlangt, kann man aber nutzen, um den Einfluss auf sichtbare Knoten abzufedern. Das liegt daran, dass das direkte Umfeld der sichtbaren Knoten bereits mit all seinen Verbindungen geladen ist. Die sichtbaren Knoten sind also viel näher an ihrer optimalen Position, als es der Fall wäre, wenn das direkte Umfeld der sichtbaren Knoten nicht mit einbezogen wird.

Es ist allerdings anzumerken, dass die Struktur des Graphen auch einen erheblichen Einfluss darauf hat, wie effektiv das Vorladen von Knoten ist. So profitiert ein Graph mit baumähnlicher Struktur, deutlich mehr von der Vorlade-Strategie, als ein Graph mit zyklischer Struktur. Bei der baumähnlichen Struktur kann es nicht vorkommen, dass ein Knoten plötzlich eine Verbindung zu einem anderen Teil des Graphen aufweist, der räumlich nicht in der Nähe ist. Tatsächlich beeinflusst dieser Umstand die Umsetzung der Vorlade-Strategie maßgeblich.

Ein weiterer Aspekt, den man beachten muss, ist die Anzahl der Knoten, die man im Layout vorlädt. Nicht selten können einzelne Knoten in einem Graphen mehrere hundert Nachbarknoten aufweisen. All diese Knoten im Layout vorzuladen führt schnell zu einem Platzproblem. Steiger et al. [SMK13] machen sich die Vorlade-Technik auch zu nutze und lösen das Problem, indem sie eine DOI-Funktion einsetzen, die nur die wichtigsten Knoten dem Layout hinzufügt.

Die Vorladestrategie ist wie gesagt stark vom verwendeten Graphen abhängig. Der Beispielgraph, der für diesen Prototypen benutzt wurde, weist keine baumähnliche Charakteristik auf. Die Anzahl der möglichen Nachbarknoten eines sichtbaren Knotens ist in diesem Beispielgraphen relativ gering. Um die Anzahl der unsichtbaren Knoten noch geringer zu halten, wurden in der finalen Umsetzung nur Knoten hinzugefügt, die durch eine ausgehende gerichtete Kante verbunden sind. In dieser Arbeit untersuche ich nun zwei Vorlade-Strategien.

4.2.1. Zwei-Hop-Strategie

Die erste Variante, die zum Einsatz kommt, lädt alle Nachbarknoten in einem zwei-Hop-Radius ins Layout. Das heißt, jeder unsichtbare Knoten ist maximal über zwei Kanten von einem beliebigen sichtbaren Knoten erreichbar, wie in Abbildung 4.1 zu sehen ist. Die Überlegung, Knoten vorzuladen,

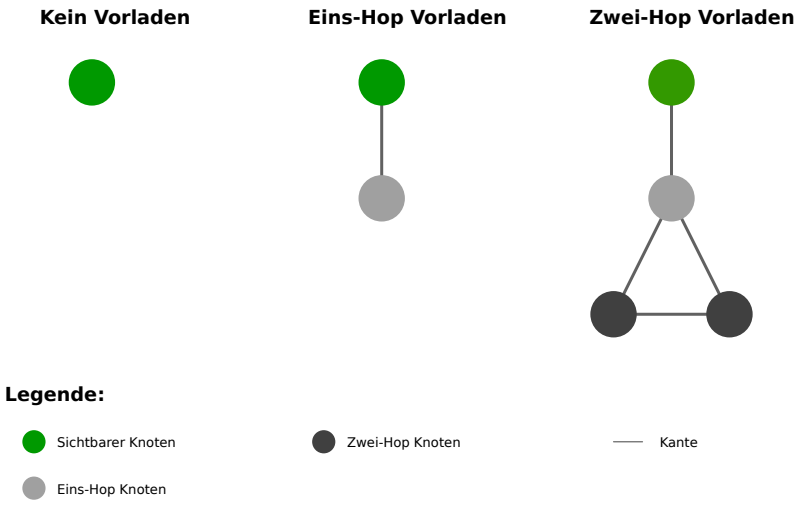


Abbildung 4.1.: Der selbe Graph in drei verschiedenen Vorlade-Szenarien.

ist bereits sehr früh in der Entwicklung des Prototypen aufgekommen, leider stand der später verwendete Beispielgraph zu diesem Zeitpunkt noch nicht zur Verfügung. Deshalb griff ich zunächst auf zufällig generierte Graphen zurück.

Die Idee hinter dieser Strategie ist es, je mehr Knoten bereits vorgeladen sind, desto geringer ist der Einfluss auf die bereits sichtbaren Knoten im Layout, wenn neue Knoten in die Simulation gegeben werden. Mit dem Zufallsgraphen konnte auch ein positiver Effekt im Vergleich zu dem Nicht-Vorladen der Knoten festgestellt werden. Die sichtbaren Knoten haben keine drastischen Positionswechsel durchgeführt. Mit dem später verfügbaren Beispielgraph hat sich die zwei-Hop-Strategie nicht bewährt. Das liegt daran, dass beide Graphen eine unterschiedliche Struktur aufweisen. Während jeder Knoten im Zufallsgraphen durchschnittlich zwei Nachbarknoten hat, sind es im Beispielgraphen ca. 5,2 Nachbarknoten. In der Praxis hat der Zufallsgraph somit eine signifikant geringere Dichte als der Beispielgraph. Das führt dazu, dass die vorgeladenen, unsichtbaren Knoten im Beispielgraphen die sichtbaren Knoten negativ beeinflussen. Eine hohe Anzahl verbundener Knoten führt deutlich wahrscheinlicher zu einem Layout mit vielen schneidenden Kanten. In der Praxis ist das ein großes Problem, welches die positiven Effekte der vorgeladenen Knoten zerstört.

4.2.2. Eins-Hop-Strategie

Da die zwei-Hop-Strategie nicht die gewünschten Resultate hervorgerufen hat, wurde zusätzlich mit nur eins-Hop vorgeladenen Nachbarknoten getestet. Zu viele stark verbundene Knoten sind nicht vorteilhaft für das Layout und die unsichtbaren Knoten nehmen eine zu starke Gewichtung im Layout ein. Daher ist es das Ziel, den Einfluss der unsichtbaren Knoten abzuschwächen und das Layout auf diese Weise von zu vielen Kanten zu entlasten. Nun ist jeder unsichtbare Knoten maximal über eine Kante von einem beliebigen sichtbaren Knoten erreichbar. In Abbildung 4.1 wird der Vergleich zwischen der eins-Hop und der zwei-Hop-Vorladestrategie verdeutlicht. Ein weiterer Vorteil der eins-Hop-Strategie ist, dass nun nur noch Kanten in das Layout geladen werden,

die mit mindestens einem sichtbaren Knoten verbunden sind. Das heißt, Kanten zwischen zwei unsichtbaren Knoten, werden weder ins Layout noch in die Simulation (bzw. den Algorithmus) geladen. Dadurch wird erreicht, dass die Kantenanzahl möglichst gering gehalten wird und nur diese im Layout sind, die aktiv auf sichtbare Knoten einwirken. Die eins-Hop-Strategie liefert im Vergleich zur zwei-Hop-Strategie bessere Resultate und sie ist auch beim gänzlichen Verzicht auf das Vorladen von Knoten, überlegen. Daher wird in dem hier vorgestellten Prototypen die eins-Hop-Strategie zum Vorladen von Knoten verwendet.

4.3. Platzieren neuer Knoten

Ein wichtiger Bestandteil für die Umsetzung eines inkrementellen Layouts beginnt bereits bei der initialen Platzierung der Knoten im Graphen. Eine gute initiale Platzierung hat die Möglichkeit die Simulation von L_n nach L_{n+1} zu unterstützen, so dass der Einfluss auf die bereits im Layout befindlichen Knoten, minimal bleibt. Dazu werden im Folgendem zwei Platzierungs-Möglichkeiten betrachtet.

4.3.1. Platzierungtechnik I: Zufällig

Die naheliegende Möglichkeit, welche auch oft bei statischen Layoutalgorithmen angewandt wird Fruchterman et al. [FR91], ist die zufällige Platzierung der Knoten im Simulationsbereich. Bei einem statischen LA ist das nicht so relevant, da meist alle Knoten gleichzeitig platziert werden und die Simulation der Knoten dann ein resultierendes Layout erzeugt. Das Ziel ist es in diesem Fall, eine optimale Position der Knoten zu finden. Im Falle eines dynamischen LA haben wir aber bereits ein Graphlayout, welches bereits optimale Positionen für die Knoten berechnet hat. Die Schwierigkeit liegt nun vielmehr darin, das alte Layout möglichst zu erhalten, während die neuen Knoten sich in dieses alte Graphlayout einfügen. Eine zufällige Platzierung von Knoten ist dann möglicherweise störend für das Graphlayout. Der Vorteil ist aber, dass wir durch das vorhandene Graphlayout auch Informationen haben, die wir für das Platzieren der Knoten verwenden können. Es ist für neue Knoten bekannt, mit welchen Knoten sie im Layout eine Kantenverbindung teilen. Die im Prototypen benutzte Platzierungstechnik II macht sich diese Informationen zu nutze.

4.3.2. Platzierungtechnik II: Geometrisches Zentrum

Eine andere Herangehensweise, die neuen Knoten im Layout zu platzieren, besteht darin die Positionen von Nachbarknoten zu betrachten und dadurch eine initiale Position des neuen Knoten zu ermitteln. Die Arbeit von Crnovrsanin et al. [CCM15] stellt eine Platzierungstechnik vor, in der neue Knoten im geometrischen Mittelpunkt der Nachbarknoten platziert werden. Knoten mit nur einem Nachbarn haben diesen Mittelpunkt natürlich nicht und deshalb werden diese zufällig, in einem festgelegten Radius, um den einzigen Nachbarknoten platziert.

Die Arbeit von Steiger et al. [SMK13] wählt einen einen anderen Ansatz. Hier wird für neue Knoten mit mehr als zwei Nachbarn das Gravitationszentrum dieser berechnet. Da jeder Knoten eine Masse zugewiesen bekommt, ist dies auch möglich. Anschließend werden die Knoten auf einem Radius der gewünschten Kantenlänge des expandierten Knotens platziert, so dass der Knoten möglichst

nahe des berechneten Gravitationszentrums liegt. Für Knoten mit nur einem Nachbarn wird der Knoten wie in Crnovrsanin et al. [CCM15] zufällig platziert. Knoten mit zwei Nachbarn werden zudem auf einem Kreis platziert, allerdings nicht mehr zufällig, sondern so, dass beide Kanten die gewünschte Länge haben. Auf einem Kreis sind das genau zwei mögliche Positionen [SMK13].

Nun wird im Folgendem die Platzierungstechnik II erläutert, welche im Prototypen implementiert wird. Dabei wird auch hier zwischen Knoten mit mehreren Nachbarn und Knoten mit nur einem Nachbarn unterschieden.

Knoten mit mehreren Nachbarn

Bei neuen Knoten, die bereits mehrere Verbindungen zu den im Layout befindlichen Knoten aufweisen, wird jeder einzelne Knoten im geometrischen Zentrum zu seinen Nachbarknoten platziert. Die hier angewendete Platzierungstechnik von Crnovrsanin et al. [CCM15] hat sich in der Praxis als hilfreich erwiesen, den Einfluss auf das Layout minimal zu halten. Die Platzierungstechnik aus der Arbeit von Steiger et al. [SMK13] wurde nicht getestet, da der in dieser Arbeit eingesetzte LA keine Massen nutzt.

Knoten mit nur einem Nachbarn

Knoten mit nur einem Nachbarn werden zufällig in einem Radius um den einzigen Nachbarknoten platziert. Der Unterschied zu der Arbeit von Crnovrsanin et al. [CCM15] ist, dass hier der Radius des Kreises abhängig vom platzierten Knoten ist. Wenn der neue platzierte Knoten einen Knotengrad von eins hat, dann ist der Radius bei 60 Pixeln und wenn der platzierte Knoten mehrere Nachbarn hat, dann liegt er bei 100 Pixeln. Da Knoten mit einem Knotengrad von eins weniger Platz brauchen als Knoten mit einem höheren Knotengrad, werden erstere näher am Nachbarknoten platziert, als solche mit einem höheren Knotengrad.

4.4. Löschen von Knoten

Die Graph-Exploration beinhaltet auch das Löschen von Knoten. Hierbei wird zwischen zwei Fällen unterschieden. Erstens geht es um das aktive Löschen von Knoten, das heißt, Nutzer*innen wählen einen Knoten zum Entfernen aus, wenn sie ihn nicht mehr im Layout sehen möchten. Der zweite Fall ist das passive Löschen von Knoten. Das passive Löschen tritt auf, wenn Nutzer*innen dem Layout neue Knoten hinzufügen. Das passive Löschen soll den Graphen von Knoten befreien, die für Nutzer*innen nicht relevant sind, und somit das Layout durch zu viele Knoten und Kanten unübersichtlich machen. In beiden Fällen werden die Knoten aus dem sichtbaren Teil des Layouts entfernt und zu unsichtbaren eins-Hop-Knoten. Für Nutzer*innen sind diese Knoten jetzt nicht mehr sichtbar. Ein vorher sichtbarer Knoten, der durch die Vorladestrategie in jedem Fall einen sichtbaren Nachbarknoten hat, verbleibt in der Simulation. Unsichtbare Knoten hingegen, die keine sichtbaren Nachbarn mehr haben, werden ganz entfernt.

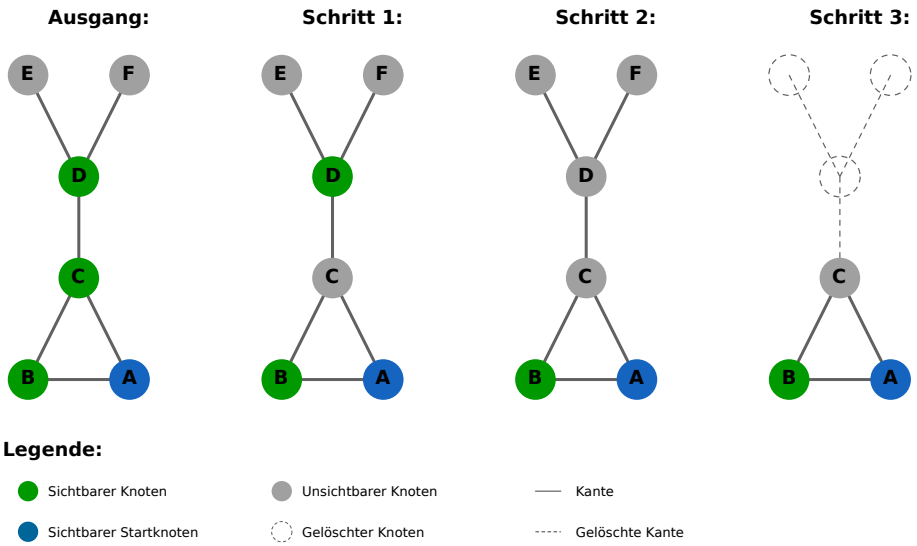


Abbildung 4.2.: Aus dem angegebenen Graphen wird in Schritt 1 der Knoten C entfernt und zu den unsichtbaren Knoten hinzugefügt. In Schritt 2 wird Knoten D automatisch den unsichtbaren Knoten hinzugefügt, da dieser nun keine sichtbare Verbindung mehr zu Startknoten A hat. In Schritt 3 werden alle unsichtbaren Knoten gelöscht, die keine direkte Verbindung zu sichtbaren Knoten haben.

4.4.1. Aktives Löschen

In Abbildung 4.2 wird der nachfolgende Vorgang des aktiven Löschens in drei Schritten aufgezeigt. In der Ausgangslage wird der Graph gezeigt, bevor Nutzer*innen einen Knoten entfernt haben. In Schritt eins wählen Nutzer*innen einen Knoten aus, den sie aus der Simulation entfernen und dieser Knoten wird dann den unsichtbaren Knoten hinzugefügt. Daraus kann sich jetzt ein Graph ergeben, der für Nutzer*innen aus zwei unabhängigen Komponenten besteht. Da es sich aber um eine Ego-basierte Exploration handelt und der Startknoten für Nutzer*innen einen sichtlichen Zusammenhang zum restlichen Graphen haben soll, werden in Schritt zwei alle sichtbaren Knoten, die keine direkte Verbindung über andere sichtbare Knoten zum Startknoten mehr haben, auch unsichtbar gesetzt. Im dritten Schritt werden alle Knoten, die unsichtbar sind und nicht Nachbarn eines sichtbaren Knotens sind, entfernt. Außerdem werden alle Kanten entfernt, die nicht mit einem sichtbaren Knoten verbunden sind. Durch dieses Vorgehen wird gewährleistet, dass der sichtbare Graph nicht in Komponenten zergliedert wird und außerdem ungewünschte Knoten aus der Simulation entfernt werden.

4.4.2. Passives Löschen

Auch beim passiven Löschen von Knoten aus dem sichtbaren Layout ist das Vorgehen gleich. Der einzige Unterschied liegt in Schritt eins. Hier wählen nicht Nutzer*innen den Knoten aus, den sie verschwinden lassen möchten, sondern das Programm wählt diese Knoten aus. Die Auswahl der zu entfernenden Knoten wird durch die Distanz zum gerade explorierten Knoten getroffen. Es werden

also alle Knoten entfernt, die von dem gerade explorierten Knoten vier Hops entfernt sind, diese Distanz wurde gewählt, da hier in der Praxis die besten Erfolge erzielt wurden. Allerdings gibt es auch eine Ausnahme, denn einige Knoten werden nicht entfernt. Der Startknoten wird nicht gelöscht, da er einen besonderen Stellenwert im Layout hat. Von ihm geht die Exploration aus und damit ist er immer Teil des Layouts. Außerdem werden Knoten auch nicht gelöscht, wenn sie die Verbindung zwischen dem Startknoten und dem gerade explorierten Knoten unterbrechen würden. Getestet wurde außerdem das Entfernen von Knoten in einer drei-Hop- oder fünf-Hop-Distanz. Die drei-Hop-Distanz wird als zu strikt angesehen, da aufgedeckte Knoten sehr schnell entfernt werden. Und die fünf-Hop-Distanz hat nicht den gewünschten Effekt gebracht, dass das Layout übersichtlich bleibt. Mit der vier-Hop-Distanz ist ein akzeptabler Mittelweg implementiert.

4.5. Layoutalgorithmus

Nachdem nun über Techniken gesprochen wurde, die den LA bei seiner Arbeit unterstützen, wird dieser jetzt selber in den Fokus genommen. Der LA soll genau die Eigenschaften haben, die in Sektion 4.1 bereits erläutert wurden. Hier möchte ich erwähne, dass einige Anforderungen des Layouts durchaus im Widerspruch stehen. Ein dynamisches Layout, mit möglichst wenig schneidenden Kanten und gleichzeitig minimaler Änderung der Anordnung der Knoten, ist eine Herausforderung. Ein optimales Layout L_n kann eine ganz andere Anordnung von Knoten als in L_{n+1} haben, selbst wenn in Layout L_{n+1} nur ein einzelner Knoten dazu kommt. Beide Layouts wären optimal, wenn man sie einzeln betrachtet. Als dynamisches Layout soll sich die Anordnung der Knoten allerdings möglichst wenig ändern. Es stehen somit zwei Anforderungen im Konflikt und es muss ein Mittelweg gefunden werden. Basierend auf dieser Ausgangslage werden nun verschiedene Ansätze zur Umsetzung eines LA verfolgt.

In den letzten Abschnitten wurden Techniken vorgestellt, mit denen Layoutalgorithmen unterstützt werden können, damit sich Knoten im Layout möglichst wenig neu positionieren müssen. In diesem Abschnitt werden Layoutalgorithmen untersucht, welche dafür eingesetzt werden, um Knoten im Layout zu bewegen.

Zu erwähnen ist, dass alle Ansätze dieser Arbeit mit Hilfe von D3 umgesetzt werden, das Funktionalitäten für kräftebasierte Layoutalgorithmen bereit stellt. Die grundlegende Funktionsweise der D3-Kräfte Simulation sieht man anhand der *Tick*-Funktion in Listing A.1. Die *Tick*-Funktion ist für die Ausführung eines Schrittes in der Simulation verantwortlich. Wie oft die *Tick*-Funktion aufgerufen wird ist vom α-Wert abhängig. In Zeile 7 der *Tick*-Funktion kann man sehen, wie der α-Wert bei jedem Aufruf angepasst wird. In Gleichung 4.1 ist die Gleichung noch einmal angegeben. Der α-Wert ist zu Beginn der Simulation auf eins gesetzt. Die Variable $alphaTarget = 0$ steht für den angestrebten α-Wert, welcher im Rahmen dieser Arbeit nicht verändert wird. Vor jeder Ausführung der *Tick*-Funktion wird geprüft, ob der α-Wert unter einen Minimalwert gefallen ist, der als $alphaMin = 0.001$ festgelegt ist. Wenn dieser Minimalwert unterschritten wird, stoppt die Simulation. Solange die Simulation aber noch läuft, werden in den Zeilen 9-11 die Kräfte aufgerufen, die man der Simulation hinzugefügt hat. Den Kräften wird der aktuelle α-Wert übergeben, der in die Berechnung der Kräfte mit einfließt. Da der α-Wert immer geringer wird, werden die resultierenden Kräfte auch immer geringer und die Simulation kühlt ab. Die Kräfte können die x und y-Koordinaten sowie vx und vy-Geschwindigkeiten eines Knotens anpassen. Diese Werte sind als Attribute eines Knotens gespeichert. Ein Knoten hat außerdem die Attribute fx und fy. Diese sind

standardmäßig auf *null* gesetzt. fx und fy werden nicht von Kräften angepasst, allerdings können sie dafür genutzt werden, um Positionen eines Knotens zu fixieren, sodass x und y-Koordinaten sowie vx und vy-Geschwindigkeiten keinen Einfluss mehr auf die Position des Knotens haben. Für den Startknoten des Graphen wird genau das getan. Der Startknoten ist in der Mitte des Bildschirms fixiert. In Zeile 16 und 18 ist zu sehen, wie fx und fy die Koordinaten bestimmen, wenn sie einen Wert haben. In Zeile 15 und 17 sieht man, dass im umgekehrten Fall die x und y-Koordinaten in jedem *Tick* angepasst werden. Die vx und vy-Geschwindigkeiten werden zu den aktuellen Koordinaten addiert um die neuen Knotenpositionen zu berechnen. Außerdem wird diese Veränderung noch mit dem Wert *velocityDecay* = 0.4 multipliziert, das soll Reibung simulieren. Mit dem Wert *alphaDecay* kann festgelegt werden, wie schnell der α-Wert fällt. Der Standardwert für *alphaDecay* ist in Gleichung 4.2 definiert. Daraus ergibt sich, dass die *Tick*-Funktion in jedem Simulationsvorgang 300 Mal aufgerufen wird.

$$alpha+ = (alphaTarget - alpha) * alphaDecay; \tag{4.1}$$

$$1 - alphaMin^{\frac{1}{300}} \tag{4.2}$$

D3 stellt mehrere Kräfte zur Verfügung, die in der Simulation verwendet werden können. Im Rahmen der Arbeit wurden zwei dieser Kräfte als Basis für die Umsetzung der Layoutalgorithmen verwendet. Die erste Kraft ist eine anziehende Kraft zwischen Knoten, die sich eine Kante teilen. In Listing A.2 wird die Berechnung dieser Kraft abgebildet. In Zeile 2 kann man wie in der *Tick*-Funktion die Anzahl der Durchläufe festlegen. Der Standardwert liegt bei einem einzigen Durchlauf. Eine Zeile weiter werden alle Kanten in einer Schleife durchlaufen. Die Kanten sind hier im Array *links* gespeichert. Ein *link* hat die Referenzen des Ausgangsknoten *source* und des Zielknotens *target* gespeichert. In den Zeilen 5-7 wird die Distanz l der beiden Knoten zueinander bestimmt. In Zeile 8 wird für die Distanz l mit der gewünschten Distanz für diese Kante aus dem Array *distances* der relative Unterschied berechnet. In *distances* sind die gewünschten Kantenlängen aller Kanten gespeichert. Diese können vor der Simulation festgelegt werden. Als Standard ist eine Länge von 30 Pixeln festgelegt.

Anschließend wird das Ergebnis der vorherigen Berechnung mit dem aktuellen α-Wert und der festgelegten Stärke der Kante aus *strengths* multipliziert. Die Stärke der Kante kann wie die Länge der Kante vor der Simulation festgelegt werden. Der Standardwert ist hier durch die Gleichung 4.3 für jede Kante gegeben. Je höher der Knotengrad, desto schwächer ist also die anziehende Kraft, die zwischen den Knoten der Kante wirkt.

$$\frac{1}{\min(\text{Knotengrad source}, \text{Knotengrad target})} \tag{4.3}$$

Anschließend wird die berechnete relative Änderung der Position mit der aktuellen Position multipliziert und in den Zeilen 10-13 mit den aktuellen Geschwindigkeiten vx und vy verrechnet. Hier ist zu beachten, dass in die Berechnung noch ein *bias* mit einbezogen wird. Beide Knoten werden

nicht gleichermaßen von der anziehenden Kraft zwischen ihnen beeinflusst, denn der Knotengrad wird hier mit eingerechnet. Die Gleichung 4.4 zeigt wie der *bias* für den Ausgangsknoten berechnet wird. Für den Zielknoten wird der Wert $1 - bias$ genommen.

$$bias = \frac{\text{Knotengrad source}}{\text{Knotengrad source + Knotengrad target}} \tag{4.4}$$

Die abstoßende Kraft ist in D3 mit Hilfe von Quadtrees umgesetzt, sodass die Barnes-Hut-Approximation [BH86] implementiert werden kann. In Listing A.3 sieht man die Implementierung dieser Kraft. In Zeile 2 wird der Quadtree erzeugt. Danach wird für jeden Quad, von klein nach groß, im Quadtree die *accumulate*-Funktion ausgeführt. Die *accumulate*-Funktion ist in Listing A.4 beschrieben. Hier werden die Kraft und der Kräfteschwerpunkt der einzelnen Quads berechnet. Ein Knoten hat standardmäßig die negative Kraft von -30. Dieser Wert kann vor der Simulation ebenfalls angepasst werden. Durch einen bottom-up Ansatz werden diese Kräfte immer gröber in größeren Quads zusammengefasst.

Bis dahin wurde nur der Quadtree erzeugt der zur Berechnung der Kräfte notwendig ist. Jetzt werden die resultierenden Kräfte berechnet und dies geschieht, indem für jeden Knoten der Quadtree durchgegangen und für jeden Quad die *apply*-Funktion (A.5) ausgeführt wird. Durch dieses Vorgehen muss nicht für jeden Knoten einzeln berechnet werden welche Kraft er auf jeden anderen Knoten ausübt. Vielmehr reicht es, die Kraft von zusammengefassten Knoten eines Quads gegenüber einzelnen Knoten zu berechnen. Dadurch ergibt sich eine bessere Laufzeit von $O(|V| \log |V|)$ anstatt von $O(|V|^2)$, wie in Barnes et al. [BH86] beschrieben.

Im Folgendem werden verschiedene Ansätze diskutiert, die diese eben vorgestellten abstoßenden und anziehenden Kräfte verwenden. Diese werden an gegebenen Stellen angepasst, wenn es für den vorgestellten Ansatz notwendig ist. Um die verschiedenen Ansätze hinsichtlich ihrer Effektivität zu beurteilen, wurden, wenn nicht anders erwähnt, die Standardwerte für die Simulation benutzt.

4.5.1. Force Scan Layout

Der erste Algorithmus ist die Umsetzung des von Misue et al. [MELS95] vorgestellten *Force Scan* Algorithmus. Der Force Scan Algorithmus ist der einzige betrachtete LA, der nicht auf einer kräftebasierten Berechnung basiert. Entworfen wurde der Force Scan Algorithmus, um Änderungen in bereits existierenden Graphlayouts anzupassen. In 4.1 ist die Implementierung des Force Scan Algorithmus in Typescript zu sehen, wie er in dieser Entwicklung verwendet wurde.

Eine wesentliche Forderung an ein dynamischen LA ist es, dass das mentale Bild von Nutzer*innen erhalten bleibt. Der Force Scan Algorithmus schafft es, indem er die relativen Positionen von Knoten erhält. Allerdings können die Knoten sich nicht neu positionieren, wenn es notwendig wäre. Sie bleiben relativ gesehen immer an der selben Stelle. Gerade bei einem inkrementellen Layout, in dem immer mehr Knoten nach und nach hinzugefügt werden, ist die Neupositionierung im Layout allerdings notwendig, um auf neue Kantenverbindungen zu reagieren. Die vorgestellte Kraft zwischen Kanten in Listing A.2 ermöglicht es, dass Knoten, die sich eine Kante teilen, räumlich angenähert werden. Mit dem Force Scan Algorithmus wird diese Art der Annäherung jedoch nicht angestrebt. Deshalb lässt sich für die Ego-basierte Graphexploration der Force Scan

Algorithmus nicht gut anwenden. Einige seiner Eigenschaften sind allerdings von Relevanz für die weitere Entwicklung des LA. Zum einen ist hier zu nennen, dass mit dem Force Scan Algorithmus Überlappungen von Knoten verhindert werden. Zum anderen bleibt das mentale Bild des Layouts sehr gut erhalten, auch wenn die Exploration im Graphen erschwert ist.

```
function force() {
  const spacing = 10;
  const xSort = [...Array(graph.nodes.length).keys()];
  const ySort = [...Array(graph.nodes.length).keys()];
  xSort.sort((a, b) => graph.nodes[a].x - graph.nodes[b].x);
  ySort.sort((a, b) => graph.nodes[a].y - graph.nodes[b].y);
  const length = graph.nodes.length;

  for (let i= 0; i < length; i++) {
    let k = i;
    while (length > k + 1 && graph.nodes[xSort[i]].x == graph.nodes[xSort[k + 1]].x) {
      k++;
    }
    let delta = 0;
    if (length > k + 1) {
      delta = force_scan(graph.nodes[xSort[i]].x, graph.nodes[xSort[k + 1]].x, spacing);
    }
    for (let j = k + 1; j < length; j++) {
      graph.nodes[xSort[j]].x += delta;
    }
    i = k;
  }

  for (let i = 0; i < length; i++) {
    let k = i;
    while (length > k + 1 && graph.nodes[ySort[i]].y == graph.nodes[ySort[k + 1]].y) {
      k++;
    }
    let delta = 0;
    if (length > k + 1) {
      delta = force_scan(graph.nodes[ySort[i]].y, graph.nodes[ySort[k + 1]].y, spacing);
    }
    for (let j = k + 1; j < length; j++) {
      graph.nodes[ySort[j]].y += delta;
    }
    i = k;
  }
}
```

Listing 4.1: Implementierung des Force Scan Algorithmus in TypeScript

4.5.2. Community Layout

Am Force Scan Algorithmus ist störend, dass sich die relativen Positionen der Knoten nicht mehr verändern, nachdem sie dem Layout hinzugefügt wurden. Das hat zur Folge, dass schlecht platzierte Knoten in ihrer Position bleiben. Um dem entgegen zu wirken, werden kräftebasierte Layoutalgorithmen in den Fokus gestellt, die diese Einschränkung nicht haben. Die Positionierung von Knoten ist nämlich genau die Problematik, für die kräftebasierte Layoutalgorithmen eingesetzt werden. Mit der Entscheidung, dass man Knoten im Layout neu positionieren möchte, kommt nun

die Frage auf, welche Knoten man neu positionieren muss. Denn am Force Scan Algorithmus kann man gut sehen, dass ein Beibehalten der relativen Positionen von Teilen des Graphen einen positiven Effekt auf das mentale Bild hat. In diesem Ansatz ist also das Ziel Knoten in zwei Gruppen aufzuteilen. Erstens die Knoten, die ihre relative Position beibehalten sollen und zweitens die Knoten, die sich neu positionieren sollen. Die Idee ist es den Graphen in abgegrenzte Communities aufzuteilen. So kann man jede Community einzeln betrachten. Sollten Änderungen im Graphen auftreten, die keine Knoten in einer Community betreffen, dann müssen diese Knoten sich nicht bewegen. Betroffene Communities dürfen ihre Knoten im LA bewegen. Die Idee dieser Herangehensweise ist durch die Arbeit von Frishman et al. [FT04] entstanden, da sie ein dynamisches Layout vorstellen, welches den Erhalt von Cluster-Strukturen anstrebt. Allerdings wird von Frishman et al. vorausgesetzt, dass der Graph eindeutige Cluster hat und das diese bekannt sind. Dieses Wissen kann hier in dieser Arbeit nicht vorausgesetzt werden. Deshalb müssen Communities (die oft nicht von Clustern unterschieden werden) erst mal gefunden werden.

Das Finden von Communities ist eine relevante Forschungsfrage für sich. In dieser Arbeit wurde nicht versucht einen eigenen Community-Algorithmus zu entwerfen. Stattdessen wird ein existierender Algorithmus implementiert. Es gibt eine Vielzahl von Algorithmen, mit denen sich Communities auffinden lassen. Newman [New18] gibt einen Überblick der wichtigsten Algorithmen und Konzepte um Communities zu finden. In dieser Arbeit wird der Louvain-Algorithmus von Blondel et al. [BGLL08] zum Finden von Communities verwendet. Bei der Implementation hat die Angabe eines Louvain-Algorithmus-Pseudocodes, aus der Arbeit von Nguyen [Ngu21], geholfen den Algorithmus umzusetzen. Die Wahl ist auf den Louvain-Algorithmus gefallen, da die Laufzeitkomplexität vergleichsweise gut ist im Gegensatz zu anderen Community-Algorithmen [New18].

Mit Hilfe des Louvain-Algorithmus ist es möglich die vorhandenen Knoten im Layout in Communities aufzuteilen. Der Ablauf im Programm sieht so aus, dass Nutzer*innen neue Knoten auswählen, die dem Layout hinzugefügt werden sollen. Diese werden dann anhand der vorgestellten Technik aus Kapitel 4.3 im Layout platziert. Bevor nun der Layoutalgorithmus gestartet wird, werden die Communities mit dem Louvain-Algorithmus berechnet. Danach bekommt jeder Knoten die Community zugeteilt, in der er sich befindet.

Die Idee ist, dass ein LA aus zwei Ebenen besteht. In der ersten Ebene wird für jede Community einzeln ein Layout generiert. Das heißt jede Community wird als einzelne Graphkomponente gehandhabt und besitzt ihr eigenes lokales Layout. Der benutzte LA ist für dieses lokale Layout zunächst nicht relevant. In der zweiten Ebene werden diese einzelnen Graphlayouts der Communities zu einem globalen Layout hinzugefügt. Das kann durch einen weiteren Layoutalgorithmus umgesetzt werden. Das globale Layout kann mit dem Force Scan Algorithmus umgesetzt werden, um die relativen Positionen der Communities beizubehalten. Die Abbildung 4.3 verbildlicht die eben geschilderte Vorgehensweise.

Es zeigt sich: Erstens kann dadurch die Berechnungszeit des Gesamtlayouts verkürzt werden, da das Aufteilen der Knoten in mehrere kleinere Simulationen die Gesamtlaufzeit verkürzt. Gerade die Kräfteberechnung zwischen allen Knoten ist mit $O(|V| \log |V|)$ bei großem V sehr aufwendig. Außerdem lassen sich einzelne Simulationen parallel berechnen. Zweitens ist es nicht immer notwendig jedes Graphlayout einer Community neu zu berechnen, wenn sich in ihr die Knotenanzahl durch Hinzufügen oder Löschen nicht ändert. Wenn in Community A ein Knoten hinzugefügt wird, dann kann das Layout von Community B bestehen bleiben, da in dieser Community sich nichts

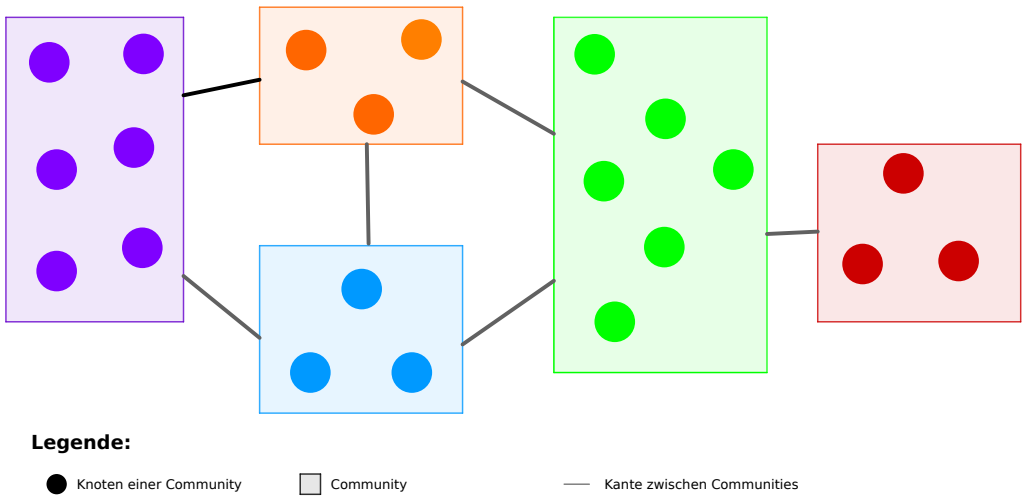


Abbildung 4.3.: In dieser Abbildung ist ein Graph zu sehen, der in fünf verschiedene Communities unterteilt ist. Diese sind farblich voneinander abgegrenzt. Die Kanten zwischen den Communities stehen für Kanten die die Knoten aus zwei verschiedenen Communities verbinden. Jede einzelne Community hat ihr eigenes lokales Layout, welche hier vereinfacht ohne Kanten dargestellt sind. Alle Communities ergeben zusammen ein globales Layout, welches mit einem LA angeordnet werden kann.

verändert hat. Drittens unterstützt es das mentale Bild des Graphen, wenn lediglich die Layouts der Communities angepasst werden, bei denen Änderungen vorgenommen wurden. Dadurch fallen die Änderungen am Gesamtlayout geringer aus.

Bei der Umsetzung des Prototyps sind Probleme aufgetreten, für die im Rahmen dieser Arbeit keine Lösung gefunden wurden. Bei einem dynamischen Layout ist es wichtig, dass die Communities sich nicht dramatisch in ihrer Zusammensetzung ändern. Ein Beispiel wäre der Fall, dass zwei Communities im Layout L_n zu einer Community in Layout L_{n+1} zusammengeführt werden, da die Konnektivität zwischen den Knoten stark zugenommen hat. Dann müssten beide Layouts, die durch dieses Communities repräsentiert werden, zu einem Layout zusammengefügt werden. Im benutzten Beispielgraphen haben sich die Zusammensetzungen der einzelnen Communities von Layout zu Layout stark unterschieden. Der Entwurf wurde hier also nicht weiterverfolgt, weil die erhofften Vorteile den Nachteilen überwiegen.

4.5.3. Knoten unterscheidende Layouts

Nachdem die Einteilung der Knoten in Communities nicht den erhofften Erfolg gebracht hat, liefern Crnovrsanin et al. [CCM15] einen weiteren Ansatz für einen LA. Crnovrsanin et al. unterscheiden zwischen Knoten, die sich im Layout bewegen dürfen und Knoten, die es nicht dürfen. Zu den Knoten, die sich bewegen dürfen, gehören alle neu hinzugefügten Knoten. Außerdem dürfen sich alle Knoten bewegen, die eine Kante mit einem neuen Knoten teilen, wenn der neue Knoten mit mehreren Knoten verbunden ist. Alle anderen Knoten dürfen sich nicht bewegen. Es werden auch

noch Regeln für neue und gelöschte Kanten aufgestellt, aber auf die wird in dieser Arbeit nicht eingegangen, da das Löschen und Hinzufügen von einzelnen Kanten nicht Teil des Prototypen ist. Diese Art, die Knoten zu unterscheiden, habe ich in den nächsten Ansätzen benutzt.

Unbewegliche Knoten

Zunächst habe ich es, wie von Crnovrsanin et al. [CCM15] vorgeschlagen, so umgesetzt, dass sich nur ein Teil der Knoten bewegen dürfen. Allerdings werden in unserer Variante des Layouts unsichtbare Knoten vorgeladen. Neue Knoten werden unsichtbar ins Layout geladen und teilen sich anfänglich nur Kanten mit anderen unsichtbaren Knoten (Hier wurde noch die zwei-Hop-Vorladestrategie benutzt). Dieser Vorteil wird direkt bei der Implementierung ausgenutzt und alle unsichtbaren Knoten dürfen sich im LA bewegen. Sichtbare Knoten haben in jedem Fall keinen Kontakt zu neuen Knoten und dürfen sich somit nicht im LA bewegen. Im Allgemeinen dürfen sich in dieser Umsetzung mehr Knoten bewegen, als es ihnen in der Implementierung von Crnovrsanin et al. [CCM15] erlaubt ist. Mehr Bewegungsfreiheit für Knoten die sogar keinen Einfluss auf das mentale Bild der Nutzer*innen haben, haben keinen Nachteil. Als Basis für den LA wurden die kantenbasierte Anziehungskraft und die abstoßende Kraft aus D3 verwendet. Für die Knoten, die sich nicht bewegen sollen, wurden keine Kräfte berechnet. Sie wurden an den gegebenen Stellen übersprungen.

Mit dem zufälligen Graphen konnten dadurch gute dynamische Layouts erzeugt werden, allerdings ist auch hier das Ergebnis stark von der Struktur des Graphen abhängig. Mit dem Beispielgraphen entstanden dadurch sehr unleserliche Layouts. Dies ist darauf zurückzuführen, dass im Beispielgraphen die Graphdichte höher ist. Dadurch sind Knoten stärker vernetzt. Sichtbare Knoten haben aber ihre Position im Layout festgelegt und bewegen sich nicht mehr. Wenn nun neue Knoten dazukommen, dann müsste für die Knoten Platz gemacht werden. Dies geschieht nicht. Stattdessen wird das Layout immer unleserlicher, da Kanten und Knoten nicht ihre optimale Position im Layout erreichen können. Ein ähnliches Problem, welches auch im Force Scan Algorithmus aufgetreten ist. Deswegen wurde auch diese Art der Implementierung verworfen.

Geschwindigkeitsreduzierte Knoten

Die Umsetzung hier ist ähnlich wie in der ersten Variante, allerdings dürfen sich die sichtbaren Knoten in diesem LA bewegen. Die sichtbaren Knoten bekommen stattdessen nur eine Einschränkung in ihrer Bewegungsfreiheit. Während sich die unsichtbaren Knoten, wie im vorherigen Ansatz mit normaler Geschwindigkeit bewegen können, wird die Geschwindigkeit der sichtbaren Knoten um einen bestimmten Prozentsatz gedrosselt. Umgesetzt wurde dies, indem die vx und vy-Geschwindigkeiten der Knoten nach der Kräfteberechnung, mit einem Faktor multipliziert wurden. Für eine Reduzierung der Geschwindigkeit um beispielsweise 20%, wurde sie mit dem Faktor $0,8$ multipliziert. Für die sichtbaren Knoten wurden mehrere Werte zwischen 20% und 80% getestet. Mit einer starken Reduzierung, näher an den 80%, konnten im Vergleich zu den vorherigen Layouts die besten Ergebnisse erzielt werden. Alle Knoten können sich bewegen und somit aus einer suboptimalen Position in eine bessere Position gelangen. Allerdings müssen viel stärkere Kräfte auf diese sichtbaren Knoten wirken, damit diese große Distanzen zurücklegen.

Alpha gesperrte Knoten

Eine andere Variante den Bewegungsfreiraum von sichtbaren Knoten einzuschränken, ist sie in der Simulation zu ignorieren, bis ein bestimmter α-Wert erreicht ist. Wie bereits bekannt, startet die Simulation mit einem α-Wert von eins und wird in jeder Iteration der Simulation verringert, bis er unter einen gewissen Schwellenwert *AlphaMin* fällt und die Simulation beendet wird. Je höher der α-Wert, desto stärker sind die Kräfte, die auf die Knoten wirken. Wenn man nun bei einem hohen α-Wert die sichtbaren Knoten nicht aktiv an der Simulation teilnehmen lässt, sondern erst, wenn der α-Wert geringer ist, dann kann man so auch die starken Positionsänderungen der sichtbaren Knoten verhindern. Es wurde mit zwei Grenzen für die α-Werte gearbeitet. Einmal durften die sichtbaren Knoten aktiv an der Simulation teilnehmen, wenn der α-Wert unter $0,9$ gefallen ist und ein anderes Mal wurde ein Grenzwert von $0,8$ getestet. Allgemein hat diese Variante nicht so gute Erfolge erzielt, wie die vorherige Variante mit der Geschwindigkeitsreduktion. Deshalb wurde dieser Ansatz nicht weiter verfolgt.

4.5.4. Masse Layout

Eine weitere Umsetzung eines LA, ist die von Steiger et al. [SMK13] vorgestellte Variante, in der den einzelnen Knoten eine Masse in die Berechnung mit eingefügt wird, so dass Knoten mit höherer Masse weniger von den Kräften beeinflusst werden, als solche mit niedriger Masse. Die Positionsänderung der Knoten zum vorherigen Layout lässt sich mit dieser Variante beeinflussen, in dem man den Knoten, die sich schon länger im Layout befinden, eine höhere Masse zuschreibt. Steiger et al. [SMK13] machen dies, in dem sie zählen, wie viele Iterationen ein Knoten sich bereits in der Simulation befindet. Zusammen mit der gesamten Anzahl an Iterationen, die die Simulation schon durchlaufen hat, wird dann eine normalisierte Masse für die Knoten berechnet. Ziel ist es, dass sich unsichtbare Knoten langsam an die Masse der sichtbaren Knoten annähern. Die Kraftverteilung zweier Knoten mit Masse m_1 und m_2 wird von Steiger et al. [SMK13] dann folgendermaßen definiert:

$$f_1 = \frac{m_2}{m_1 + m_2} \tag{4.5}$$

$$f_2 = \frac{m_1}{m_1 + m_2} \tag{4.6}$$

Die Faktoren f_1 und f_2 bestimmen, wie stark die beiden Knoten sich gegenseitig beeinflussen. So hat beispielsweise ein Knoten der gerade im Layout platziert wurde, fast gar keinen Einfluss auf einen Knoten der sich schon seit 1000 Iterationen in der Simulation befindet.

Dieses Vorgehen von Steiger et al. [SMK13] wurde in diesem Layoutalgorithmus übernommen. Umgesetzt wurde dies, indem in jedem Datenpunkt eines Knotens die Masse gespeichert wird. Wenn ein Knoten neu im Layout platziert wird, dann hat dieser die initiale Masse von eins. Nach jeder Ausführung der *Tick*-Funktion wird die Masse aller im Layout befindlichen Knoten um eins erhöht. Insbesondere wird hier nicht zwischen sichtbaren und unsichtbaren Knoten unterschieden. Die Masse eines Knotens ist in dieser Implementierung nur von der Dauer in der Simulation abhängig.

Sobald ein Knoten aus dem Layout entfernt ist, wird auch hier die Masse zurückgesetzt. Die Faktoren f_1 und f_2 werden in die Simulation folgendermaßen mit eingerechnet: In der anziehenden Kraft wird der *bias* abgeändert, sodass er nun nicht mehr vom Knotengrad abhängt, sondern von der Masse der Knoten. Der *bias* wird mit f_1 und f_2 berechnet, indem die Massen des Ausgangs- und Zielknotens eingesetzt werden. Für die abstoßende Kraft muss die Masse anders einberechnet werden, da hier die Kraft nicht zwischen einzelnen Knoten berechnet wird, sondern zwischen einem Knoten und einem Quad. Ein Quad ist eine Aggregation mehrere Knoten um die Berechnung zu beschleunigen. Deshalb muss schon in der *accumulate*-Funktion die Masse der aggregierten Knoten addiert werden. Das wird äquivalent zur Berechnung der Stärke eines Quads durchgeführt. Die Masse der Kindelemente wird jeweils aufsummiert. Anschließend wird in der *apply*-Funktion (Zeile 16 und 17) die akkumulierte Masse eines Quads mit der Masse des aktuellen Knotens eingerechnet. In Zeile 33 wird wird das gleiche für einzelne Knoten getan, wenn die Barnes-Hut Approximation [BH86] nicht angewendet werden kann.

Das Ergebnis des resultierenden Layouts ist eine deutliche Verbesserung zu den vorherigen Layoutalgorithmen. Knoten die bereits länger im Layout verweilen bewegen sich im Vergleich zu neuen Knoten deutlich weniger. Gleichzeitig bleibt das Layout im Vergleich übersichtlich. Der Einfluss neuer Knoten auf das Layout ist außerdem abgeschwächt, womit drastische Positionswechsel alter Knoten reduziert werden. Das Platzproblem bleibt aber weitestgehendes bestehen. Auch hier kommt es zu Überlappungen von Knoten.

4.5.5. Platz reservierendes Layout

Die vorherigen kräfte-basierten Algorithmen zur Berechnung eines Layouts haben alle eine Gemeinsamkeit. Sie versuchen Knoten in Kategorien zu unterteilen, die dann unterschiedliche Bewegungsfreiheiten zugewiesen bekommen. Allerdings ist gerade im Beispielgraphen ein häufiges Problem der Platzmangel, der beim Hinzufügen von neuen Knoten in das Layout entsteht. Dadurch werden zwangsläufig auch die bereits vorhandenen Knoten dazu bewegt Platz zu machen. Starke Kräfte wirken in diesen Momenten auf die Knoten ein. Hier hilft es nicht, wenn sich nahe Knoten gar nicht oder nur sehr beschränkt bewegen können, da das Platzproblem so bestehen bleibt und das Graphlayout unleserlich wird. Praktisch wäre es, wenn Knoten, die hinzugefügt werden, bereits über ausreichend Platz verfügen.

Dafür wird in diesem Ansatz noch eine weitere Kraft, zusätzlich zu den beiden oben genannten Kräften (abstoßende und anziehende Kraft), eingeführt. Diese neue Kraft ist eine Kollisionskraft, die wieder auf der Basis von D3 implementiert wurde. Diese Kollisionskraft behandelt Knoten nicht als Punkte sondern als Kreise. Dadurch soll verhindert werden, dass sich Knoten überlappen.

Die Kollisionskraft aus Listing A.6 hat einen ähnlichen Aufbau wie die abstoßende Kraft und arbeitet ebenfalls mit Quadtrees. In Zeile 2 wird hier auch wieder ein Quadtree erzeugt und für jeden Quad wird die *Prepare*-Funktion aus Listing A.7 aufgerufen. Hier wird jedem Quad der Radius des größten in ihm befindlichen Kreises zugewiesen.

Die *Apply*-Funktion aus Listing A.8 berechnet widerum die Kräfte zwischen zwei kollidierenden Knoten. Zuerst wird geprüft, ob die beiden Knoten sich überlappen (Zeile 5-7). Wenn dies der Fall ist, dann werden diese Knoten in gegen gesetzte Richtungen auseinander gestoßen, in dem die

Geschwindigkeitswerte der Knoten angepasst werden (Zeile 10-15). Falls beide Knoten zufällig auf dem selben Punkt fallen, werden diese mit einem Zufallswert auseinander gebracht. Hier umgesetzt mit der *Jiggle*-Funktion in Zeilen 8 und 9, die den x und y-Werten einen zufälligen Wert zuweist.

Der entscheidende Punkt ist, dass diese Kollisionskraft dafür genutzt wird, um neuen Knoten Platz im Layout zu garantieren. Umgesetzt wird dies, indem jedem Knoten ein eigener Radius zugewiesen wird, der abhängig vom ausgehenden Knotengrad des Knotens ist. In der Formel 4.7 wird die Berechnung der Radien gezeigt. Die Variable r ist der einheitliche Radius der Knoten, wie er auch in der Visualisierung zu sehen ist. Und die Variable d_i ist der Knotengrad des Knoten i, für ausgehende Kanten, die mit unsichbaren Knoten verbunden sind. Wie man sehen kann wird auf der rechten Seite der *max*-Funktion die Fläche dieser Nachbarknoten berechnet und mit einem Faktor von zwei multipliziert um dann anschließend aus der resultierenden Kreisfläche den neuen Radius zu berechnen. Der Faktor zwei ist ein festgelegter Wert, der in der Praxis gut funktioniert hat. Auf der linken Seite der *max*-Funktion wird der tatsächliche Radius r mal drei genommen, da das der Mindestabstand zu anderen Knoten sein soll. Je größer also d_i, desto größer ist auch der Kollisionsradius in der Simulation. Eine weitere Änderung ist, dass diese Kollisionskraft nur auf sichtbare Knoten angewendet wird. Unsichtbare Knoten werden also nicht von sichtbaren Knoten verdrängt, sondern können die Lücken zwischen den sichtbaren Knoten füllen. Zusätzlich zur Kollisionskraft werden auch die abstoßenden und anziehenden Kräfte eingesetzt. Bei diesen beiden Kräften werden allerdings alle Knoten im Layout berücksichtigt.

$$r_i = \max(r * 3, \sqrt{r^2 * d_i * 2}) \tag{4.7}$$

Außerdem wird die anziehende Kraft, die zwischen zwei mit Kanten verbundenen Knoten wirkt, angepasst. Die gewünschten Längen der Kanten, welche im *distance*-Array gespeichert werden, wurden von der Standardlänge von 30 Pixeln auf 100 Pixeln erhöht. Mit Ausnahme von Knoten mit Knotengrad von eins. Hier wird die Länge auf 60 Pixeln gesetzt, da diese weniger Platz brauchen. Bei der abstoßenden Kraft wurden die Knotenstärken von -30 auf -60 gesetzt. Die Knoten stoßen sich also doppelt so stark ab. Außerdem wurde ein maximales Limit gesetzt für die die abstoßende Kraft zwischen zwei Knoten berechnet wird. Der Standardwert liegt bei keiner Begrenzung also unendlich. Hier wurde er auf den 40 Fachen Knotenradius gesetzt um zu weites auseinander driften des Graphen zu vermeiden.

Zusätzlich sind die Simulationswerte auch angepasst. Der Wert *alphaDecay* wurde auf 0.02 gesetzt und *alphaMin* auf 0.1. Dadurch wird die *Tick*-Funktion nur noch 114 mal ausgeführt in stelle der 300 mal bei den Standardwerten. Das wird aus zwei Gründen getan. Zum einen wird die Simulation stark verkürzt, was die Exploration des Graphen angenehmer macht. Zum anderen wird der *alphaMin*-Wert von 0.001 auf 0.1 gesetzt und dadurch werden Simulations *Ticks* mit sehr niedrigen *alpha*-Werten unter 0.1 nicht mehr ausgeführt. Da mit niedrigem *alpha*-Werten die Simulation fast abgekühlt ist, sind die Positionsänderungen der Knoten bereits minimal. So kann die Simulation nochmal verkürzt werden, ohne dass ein Unterschied im Ergebnis festgestellt werden kann. Durch diese Änderungen konnte die Simulations-Dauer auf fast ein Drittel reduziert werden.

Diese Umsetzung der Layoutberechnung wird auch in der finalen Version des Prototypen verwendet, da hiermit sehr gute Erfolge erzielt wurden. Zum einen hat das „reservieren“ von Platz für unsichtbare Knoten, geholfen das Platzproblem zu reduzieren. Und zum anderen hat sich der Einfluss neu hinzugefügter Knoten auf die bestehenden Knoten auch verringert. Durch mehr Platz müssen sich bestehende Knoten weniger bewegen.

4.6. Vorschausimulation

Neben dem nun vorgestellten kräftebasierten LA der die Änderungen im Graphen simuliert und ein Nachfolgelayout erzeugt, wird noch ein weiterer LA eingesetzt. Dieser zweite LA wird im Folgendem als Vorschausimulation bezeichnet, während der im letzten Abschnitt besprochene LA als Hauptsimulation bezeichnet wird, da beide Layoutalgorithmen auf einer kräftebasierten Simulation basieren. Die Vorschausimulation ist eine stark verkürzte Version der Hauptsimulation und soll unsichtbare Knoten, die Nutzer*innen ausgewählt haben, im Layout angemessen platzieren. Das heißt: Genügend Abstand zu anderen Knoten. Grundsätzlich werden die selben Kräfte, wie in der Hauptsimulation verwendet, allerdings werden diese nur auf die von Nutzer*innen ausgewählten Knoten angewandt. Alle anderen Knoten im Layout bewegen sich also nicht. Dies ist das selbe Prinzip, wie in Kapitel 4.5.3, wo nur ausgewählte Knoten an der Simulation teilnehmen. Außerdem wird für die Kollisionskraft nur der doppelte Radius verwendet, da hier nur Überlappungen verhindert werden sollen, und nicht wie in der Hauptsimulation, Platz reserviert werden soll. Wenn Nutzer*innen sich entscheiden, die ausgewählten Knoten dem sichtbaren Layout endgültig hinzuzufügen, dann werden die Änderungen aus der Vorschausimulation übernommen und die Hauptsimulation wird gestartet. Falls Nutzer*innen die Knoten nicht hinzufügen wollen, dann werden die Positionen zurückgesetzt.

Für die Vorschausimulation wurde der *alphaDecay*-Wert auf 0.01 gesetzt und der *alphaMin*-Wert auf 0.2. Dadurch wird die *Tick*-Funktion nur noch 16 mal ausgeführt an Stelle der 114 Mal in der Hauptsimulation. Dadurch terminiert die Vorschausimulation so schnell, dass Nutzer*innen das Ergebnis mit fast keiner Verzögerung sehen können und so nicht warten müssen.

5. Visualisierung des Graphen

Im vorherigen Kapitel wurde ausführlich darüber gesprochen, wie der LA unter Berücksichtigung des mentalen Bildes von Nutzer*innen umgesetzt wird. Allerdings ist das Berechnen der Knotenpositionen im Layout nicht das einzige, was man in einem Graphlayout beachten muss. Es ist auch wichtig, das Layout angemessen visuell zu repräsentieren. In diesem Kapitel wird die visuelle Umsetzung des Graphlayouts erläutert und beschrieben. Außerdem wird ein kurzer Überblick der Interaktionsmöglichkeiten mit dem Graphen gegeben.

Die Visualisierung des Graphen findet im Browser statt und wird mit Hilfe von D3 umgesetzt. D3 ist eine Bibliothek von Bostock et al. [BOH11], die es ermöglicht Daten mit Document Object Model (DOM)-Elementen zu verknüpfen, um so DOM-Elemente zu erstellen und zu modifizieren. Im entwickelten Prototypen nutzen wir diese Eigenschaft von D3 um Kanten und Knoten des Graphen als Scaleable vector grafics (SVG)-Elemente zu repräsentieren.

5.1. Knoten

Die visuelle Repräsentation eines Knoten besteht aus einem farbigem Kreis mit einem Radius von zehn Pixeln. Die Farbe des Kreises ist abhängig von der Entitätsklasse des Knotens. Die drei Entitätsklassen werden durch *Person* in rot, *Objekt* in gelb und *Ort* in blau repräsentiert. Diese Farben wurden ausgewählt, da es sich bei drei Entitätsklassen angeboten hat, die Grundfarben zu wählen. Außerdem ist es möglich den Kreis mit dem Mauszeiger zu fokussieren, um ein Label mit der durch diesen Kreis repräsentierten Namen der Entität einzublenden. In Abbildung 5.1 wird der Unterschied an einem Beispiel gezeigt.

(a) unfokussiert **(b)** fokussiert

Abbildung 5.1.: In Bild a) wird der Knoten unfokussiert mit weißem Rand gezeigt. In Bild b) wird der Knoten mit dem Mausszeiger fokussiert und der Rand wird schwarz. Außerdem wird das Label des Knotens eingeblendet.

5.2. Kanten

Kanten werden im Layout als farbige Bézier-Kurven dargestellt. Da es sich um gerichtete Kanten handelt, übernimmt die Kante die Farbe des Ziel-Knotens. Außerdem sind Kanten zu 60% durchsichtig um die Knoten hervorzuheben. Kanten werden im Layout als Bézier-Kurven mit jeweils vier Punkten dargestellt:

Punkt 1: $[x_1, y_1]$

Punkt 2: $[\frac{x_1+x_2}{2}, y_1]$

Punkt 3: $[\frac{x_1+x_2}{2}, y_2]$

Punkt 4: $[x_2, y_2]$

Der erste und vierte Punkt sind jeweils der Start- und Zielpunkt der Kurve. Die Punkte ergeben sich aus den Koordinaten des Start- und Zielknotens. Punkt zwei und drei sind die Kontrollpunkte der Bézier-Kurve.

Im Verlauf der Entwicklung wurden auch gerade Linien getestet, die im Gegensatz zu den Bézier-Kurven, nicht in einer geschwungenen Bahn verlaufen. Die Umsetzung von Bézier-Kurven über geraden Linien hat keinen besonderen Hintergrund. Allerdings ist die Frage, welchen Einfluss die Darstellung einer Kante auf Nutzer*innen hat, in der Forschung aufgegriffen worden. So wird in der Arbeit von Xu et al. [XRP+12] die aufgestellte Hypothese widerlegt, dass Nutzer*innen eine Kurve als Kante bevorzugen. Die Arbeit von Huang et al. [HEHB16] wiederum kann dieses Ergebnis nicht replizieren. Hier werden verschiedene Darstellungsweisen von Kanten in einer Nutzerstudie evaluiert. Die Darstellung einer Kante als Kurve wird hier von den Nutzern*innen präferiert, obgleich die Unterschiede nicht groß sind. Allerdings ist hier hervorzuheben, dass die Nutzung von Kurven, um Kantenkreuzungen zu minimieren, deutlich bevorzugt wurden. Ich sehe hier also Potential zur Verbesserung der Kantendarstellung.

In dieser Arbeit wird zwischen ausgehender und eingehender Kante nicht unterschieden. Es kommt im Beispielgraphen oft vor, dass eine beidseitige Verbindung existiert. Die beiden resultierenden Kanten werden im Layout übereinander gezeichnet, da beide Bézier-Kurven mit den selben Punkten berechnet werden. Dadurch ist nur eine von beiden Kanten fokussierbar, da eine Kante unter der anderen liegt. Beim Fokussieren einer Kante, wird wie beim Knoten, ein Label eingeblendet, welches die Relation zwischen den beiden Knoten anzeigt. Bei mehreren Kanten, die übereinander gezeichnet werden, kann also nur die Relation der obersten Kante angezeigt werden. Hier besteht noch Verbesserungspotential.

5.3. Beschriftung

Zusätzlich zu den eingeblendeten Labels beim Fokussieren der einzelnen Knoten und Kanten, wurde für die Knoten eine dauerhafte Beschriftung hinzugefügt. Diese befindet sich über den einzelnen Knoten und soll den Nutzer*innen bei der Orientierung im Graphen helfen um das Kurzzeitgedächtnis zu entlasten. Für die Kanten im Graphen wurde diese Art der Beschriftung bewusst nicht umgesetzt, da das Layout sonst zu unübersichtlich wird. Falls Nutzer*innen sich für die genaue Art der Beziehung interessieren, können sie dies durch Fokussieren der Kante erfahren.

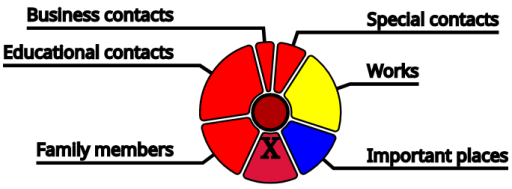


Abbildung 5.2.: Über die einzelnen Segmente des Menüs lassen sich Knoten verschiedener Kategorien hinzufügen, oder der ausgewählte Knoten kann gelöscht werden.

Für die Orientierung im Graphen ist eine dauerhafte Beschriftung der Knoten ausreichend, denn das dauerhafte Anzeigen der Labels bringt gleichzeitig Probleme mit sich. Ebenfalls kommt es mit den Labels der Knoten zu Überlappungen. Um diese Überlappungen zu vermeiden, wird vorab berechnet, ob sich zwei Labels überlappen würden und wenn ja, dann wird das weniger relevante Label ausgeblendet. Die Relevanz wird anhand des Knotengrades der möglichen ausgehenden Kanten berechnet. Der Knoten mit dem höheren Knotengrad behält sein Label. Bei gleichem Knotengrad behält der Knoten sein Label, der in der Sortierung als erstes kommt.

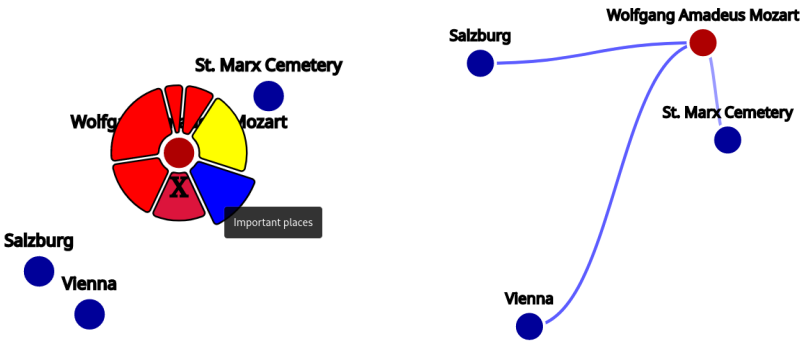
5.4. Funktionen des Prototypen

Nachdem nun die Darstellung des Graphlayouts abgeschlossen ist, werden jetzt die Interaktionsmöglichkeiten mit dem Layout genauer erklärt.

5.4.1. Knotenmenü

In Abbildung 5.2 sieht man das aufgeklappte Knotenmenü, welches sich beim Klicken des Knotens öffnet. Das Menü ist wie ein Donut-Diagramm um den Knoten angeordnet. Das Menü selber besteht aus mehreren Segmenten. Einmal ein Segment auf der unteren Seite des Menüs, welches mit einem *X* beschriftet ist. Beim Klicken dieses Segments, wird dieser Knoten gelöscht. Mit Hilfe der anderen Segmente können Knoten dem Layout hinzugefügt werden. Diese Segmente geben durch drei Merkmale Informationen darüber, welche Knoten dem Layout hinzugefügt werden können. Das erste Merkmal ist die Beschriftung der Segmente. Diese geben an, in welcher Beziehung die aggregierten Knoten zum ausgewählten Knoten stehen. Das zweite Merkmal ist die Farbe des Segments. Die Farbe gibt an, um welche Entitätsklasse es sich bei den aggregierten Knoten handelt. Diese haben die gleiche Farbkodierung wie die entsprechenden Knoten. Die Farbe rot steht also beispielsweise für Personen. Das dritte Merkmal ist die Größe der Segmente. Dieses Merkmal gibt den Hinweis, welchen Anteil die verschiedenen aggregierten Knoten am Gesamten haben.

In der Arbeit von Tu et al. [TS13] wird ein alternativer Ansatz vorgestellt. Dort werden Knoten und Kanten mittels einer Liste ausgewählt und können dann der Visualisierung hinzugefügt werden. Mit dieser Liste kann auch nach Knoten und Kanten gesucht werden. Außerdem lassen sich Eigenschaften von Knoten und Kanten in einer Liste sehr gut darstellen. Allerdings bietet das



(a) *Important Places* Vorschau **(b)** Neues Layout

Abbildung 5.3.: Wenn ein Segment zum Hinzufügen von Knoten fokussiert wird, dann werden die Knoten, die hinter der Aggregation stecken, wie in a) im Layout eingeblendet. Mit den hinzugefügten Knoten wird dann ein neues Layout b) erzeugt.

Knotenmenü den Vorteil, dass aggregierte Knoten, dynamisch durch eine Vorschau direkt im Layout, einsehbar sind. Hier liegt der Fokus auch mehr auf der Exploration, während eine Suchfunktion gut ist, um bestimmte Knoten zu finden.

5.4.2. Löschen von Knoten

Wenn die Nutzer*innen das Knotenmenü geöffnet haben, dann können sie den aktuellen Knoten aus dem sichtbaren Layout entfernen, in dem sie im Knotenmenü auf den unteren mit dem „X“ markierten Knopf drücken. Danach schließt sich das Menü wieder und der LA berechnet das neue Layout. Nach Berechnung des neuen Layouts wird das alte Layout angepasst. Wie das Entfernen des Knotens gehandhabt wird, ist in Kapitel 4.4.1 erklärt.

5.4.3. Vorschau von Knoten

Wenn Nutzer*innen ein Segment der aggregierten Knoten im Menü fokussieren, dann vergrößert sich das Segment etwas um deutlich zu machen, dass dieses Segment gerade fokussiert wird. Daraufhin werden die aggregierten Knoten im Layout eingeblendet, um eine Vorschau über die Knoten zu geben, die Nutzer*innen möglicherweise hinzufügen möchten. Dies ist in Abbildung 5.3 a) zu sehen. Wenn Nutzer*innen das Segment nicht mehr fokussieren, erhält es wieder seine Ursprungsgröße und die Knoten die in der Vorschau sind, werden wieder ausgeblendet.

5.4.4. Hinzufügen von Knoten

Die Nutzer*innen können sich aber auch dafür entschieden, die aggregierten Knoten dem Layout dauerhaft hinzuzufügen, indem sie das fokussierte Segment klicken. Dann schließt sich das Knotenmenü und der LA wird gestartet um ein Nachfolgelayout zu erzeugen. Wenn der LA das

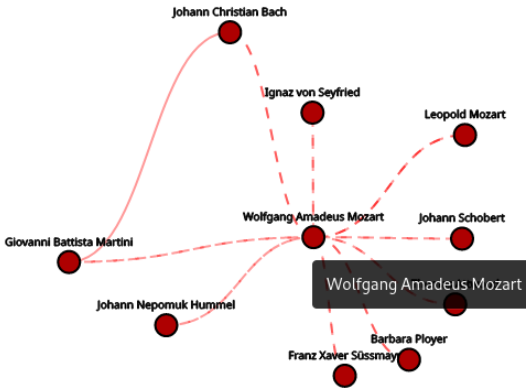


Abbildung 5.4.: Der *Wolfgang Amadeus Mozart* Knoten wird fokussiert. Er und seine Nachbarknoten bekommen eine schwarze Umrandung und die verbindenden Kanten werden gestrichelt, um Nachbarknoten hervorzuheben.

Nachfolgelayout erzeugt hat, dann wird das alte Layout ersetzt. In Abbildung 5.3 a) werden dem Startknoten *Wolfgang Amadeus Mozart* neue Knoten hinzugefügt, die dann in ein neues Layout in Abbildung 5.3 b) übergehen.

5.4.5. Hervorheben von Knoten und Kanten

Ein weiterer Bestandteil der Ego-Graph Exploration ist das visuelle Hervorheben einzelner Knoten und Kanten, wenn Nutzer*innen mit dem Mauszeiger über einen Knoten ihrer Wahl fokussieren. Der fokussierte Knoten bekommt nun eine schwarze Umrandung. Außerdem werden alle durch eine Kante direkt verbundenen Knoten gleichermaßen sichtbar gemacht. Die verbindenden Kanten werden ebenfalls durch einen Wechsel von einer durchgezogenen Linie zu einer gestrichelten Linie hervorgehoben (Abbildung 5.4). Wenn Nutzer*innen den Knoten nicht mehr fokussieren, dann werden alle visuellen Elemente wieder in ihre ursprüngliche Form gebracht.

5.4.6. Animationen

Um den Nutzer*innen Änderungen im Graphen möglichst schonend beizubringen, werden Animationen benutzt, um das mentale Bild der Betrachtenden vom Graphen zu erhalten. Auch hier stellt D3 Funktionen zur Verfügung, die es ermöglichen, Änderungen zu animieren.

Neue Knoten werden im Layout so animiert, dass sie nicht sichtbar mit einem Radius von 0 Pixeln im Layout platziert werden und dann in einer kurzen 750ms Animation auf eine Größe des Radius von 10 Pixeln anwachsen. Kanten haben keine Einführungs-Animation. Allerdings haben sowohl Knoten als auch Kanten eine Animation für Positionsänderungen. Nachdem für ein Layout L_n ein Nachfolgelayout L_{n+1} berechnet wurde, werden die Änderungen der Knoten und Kanten durch eine kurze 750ms Animation durchgeführt. Anstatt also, dass der Knoten an einer Position verschwindet und an einer neuen Position auftaucht, wird der Knoten von seiner alten Position zu seiner neuen Position hingeführt.

Alternativ könnte man auch nach jedem Simulationsschritt die Knoten- und Kantenpositionen anpassen, allerdings ist das sehr aufwändig und verlangsamt die Simulation deutlich, da diese beiden Prozesse nicht parallel laufen. Deswegen werden hier Änderungen nur einmalig nach der Simulation präsentiert.

5.4.7. Zoomen

Das Zoomen in der Anwendung wird auch unterstützt. Hierzu müssen Nutzer*innen mit dem Cursor den Hintergrund fokussieren und dann das Mausrad benutzen. Die Nutzer*innen können bis zur $0,2$-fachen Größe herauszoomen um einen Überblick über den gesamten Graphen zu bekommen, falls dieser zu groß sein sollte, um ihn in der normalen Größe vollständig zu erfassen. Dies ist besonders notwendig, da es keine direkten Beschränkungen gibt, wie weit der Graph sich in die x und y-Richtung ausbreiten kann. Wenn das Mausrad in die andere Richtung betätigt wird, dann zoomt man hinein. Man kann über den Ausgangswert hinaus auf eine 4-fache Größe heranzoomen. Das Zoomen ist uniform umgesetzt, somit werden alle Teile der Visualisierung beim Zoomen gleichermaßen vergrößert und verkleinert.

5.4.8. Panning

Da es sich um eine Ego-Graph-basierte Exploration handelt, ist der Startknoten, von der die Exploration ausgeht, standardmäßig immer im Mittelpunkt des Bildschirms platziert. Allerdings können Nutzer*innen sich frei im Graphen bewegen, in dem sie die linke Maustaste gedrückt halten und die Visualisierung in die gewünschte Richtung ziehen. Sie haben somit die Möglichkeit, Knoten von Interesse in den Mittelpunkt zu bringen.

5.4.9. Debug Button

Zurzeit befindet sich in der oberen linken Ecke des Browserfensters ein grauer Button mit der Aufschrift „Debug“. Dieser Button wurde in der Entwicklung des Prototypen benutzt und blendet beim Betätigen die *unsichtbaren* Knoten mit verbundenen Kanten im Layout ein. Bei einem erneuten Betätigen des Buttons, werden die *unsichtbaren* Knoten mit verbundenen Kanten wieder ausgeblendet.

6. Programmablauf

In den vorherigen Kapiteln habe ich die verschiedenen Techniken vorgestellt, welche im Prototypen zur Layouterstellung genutzt werden. Weiterhin bin ich auf die Möglichkeiten zur Interaktion mit dem Layout und die Umsetzung der Visualisierung eingegangen, jetzt greife ich diese beiden Aspekte des Prototypen wieder auf, um den Programmablauf zu erklären. In Abbildung 6.1 sind die wichtigsten Bestandteile des Programmes in einem Flussdiagramm veranschaulicht. Aus Gründen der Übersichtlichkeit werden Hervorhebe-, Zoom- und Panning-Effekte hier ausgelassen, da diese keinen Einfluss auf das dynamische Layout haben.

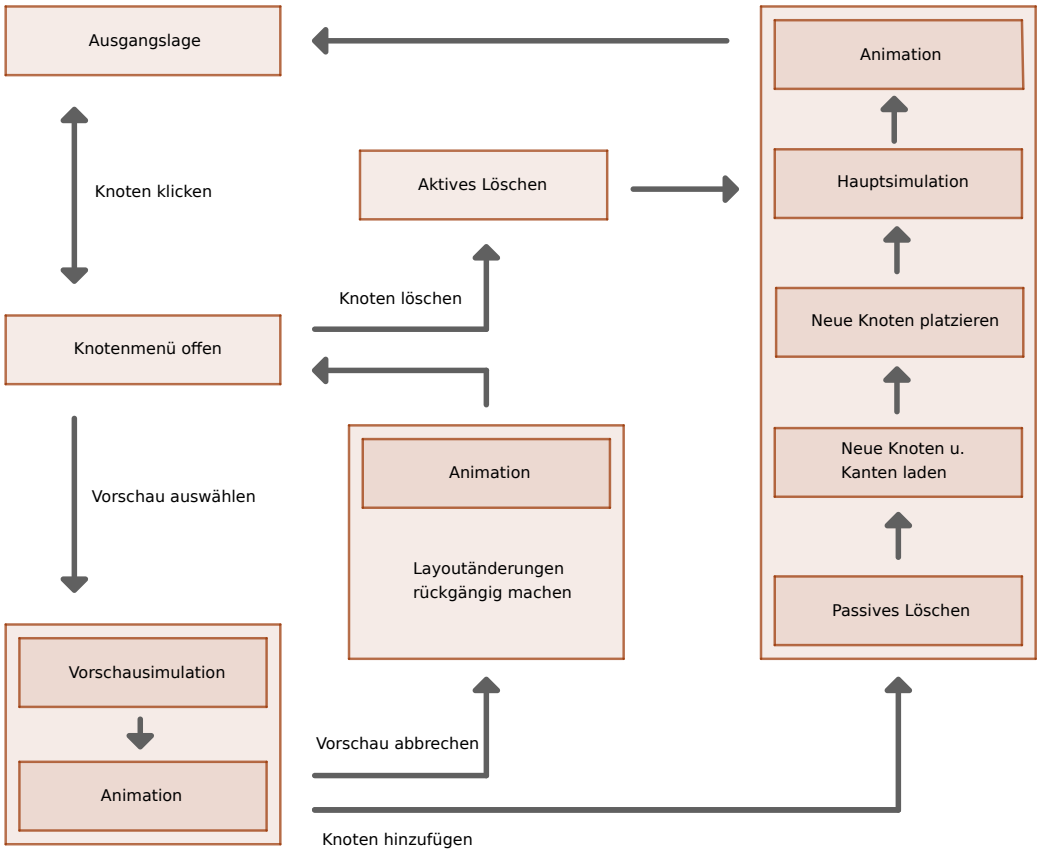


Abbildung 6.1.: Flussdiagramm des Programmablaufs.

In der Abbildung 6.1 links oben befindet sich der Graph in der Ausgangslage, das heißt es werden gerade keine Aktionen auf ihm ausgeführt. Wenn nun Nutzer*innen auf einen beliebigen Knoten klicken, dann öffnet sich das Knotenmenü. Hier können Nutzer*innen auswählen, den Knoten zu löschen. Dieser Vorgang ist bereits als aktives Löschen bekannt. Nachdem das aktive Löschen im Graphen stattgefunden hat, startet der LA um ein neues Layout zu erzeugen. Wenn der LA terminiert hat und die neuen Positionen der Knoten berechnet sind, dann werden die Änderungen im visuellen Layout durch eine Animation eingeführt. Dieser Ablauf geschieht immer dann, wenn Nutzer*innen einen Knoten über das Knotenmenü löschen.

Nutzer*innen können über das Knotenmenü auch eine Vorschau über aggregierte Knoten bekommen. Sobald sie das Knotenmenü öffnen, werden ihnen alle ausgehenden Kanten dieses Knotens, welche noch nicht sichtbar sind, zur Auswahl vorgeschlagen. Wie das aussieht, wird in Kapitel 5.4.1 gezeigt. Wenn Nutzer*innen nun mit dem Mauszeiger über eine aggregierte Gruppe fokussieren, dann wird die Vorschausimulation (Vergleiche aus Kapitel 4.6) auf diesen Knoten gestartet. Diese Simulation ist sehr kurz. Die entstandenen Änderungen der Vorschausimulation werden danach animiert und die aggregierten Knoten dadurch temporär sichtbar im Layout. Gleichzeitig werden andere Knoten unsichtbar; wenn die aggregierten Knoten endgültig expandiert werden, bleiben sie das. Nutzer*innen können auswählen, die Vorschau abzubrechen, indem sie die aggregierte Gruppe nicht mehr fokussieren, oder auf diese zu klicken und den Graphen zu expandieren. Im ersten Fall werden die Änderungen der Vorschausimulation rückgängig gemacht, indem Knoten auf ihre vorherige Position gesetzt werden. Außerdem wird die Sichtbarkeit der Knoten auf den Ausgangswert zurückgesetzt. Dies geschieht mit Hilfe einer Animation. Im zweiten Fall bleiben die Knoten sichtbar, da Nutzer*innen diese Knoten expandieren möchten. Jetzt wird das passive Löschen endgültig angewendet. Das heißt für Nutzer*innen, dass die Knoten, welche in der Vorschau unsichtbar geworden sind, das nun auch bleiben. Dadurch, dass nun die Knoten expandiert wurden, haben diese Knoten jetzt keine vorgeladenen Nachbarn, da sie selber die vorgeladenen Nachbarn waren. Das heißt, um die eins-Hop-Vorladestrategie aufrecht zu erhalten und gleichzeitig die weitere Expansion des Graphen zu ermöglichen, müssen die noch nicht geladenen Nachbarknoten dem unsichtbarem Layout hinzugefügt werden. Da diese fehlenden Knoten und Kanten nicht Teil des Graphen sind, müssen diese vom Server angefragt werden. An dieser Stelle werden nun also Anfragen an den Server gesendet, der die Nachbarknoten und Kanten kennt, da er den kompletten Graphen gespeichert hat. In der lokalen Anwendung, also dem Prototypen, wird nur der vorgeladene Teil des Graphen gespeichert.

Es werden jetzt also Serveranfragen gesendet, die für die gerade explorierten Knoten, nachfragen, ob diese Nachbarknoten haben. Die Serverantwort wird dann auf Nachbarknoten gefiltert, die durch ausgehende Kanten erreichbar sind und sich noch nicht im Layout befinden. Die Knoten, auf die das zutrifft, werden unsichtbar dem Layout hinzugefügt und sind nun die neuen eins-Hop Knoten. Als nächstes wird diesen neuen Knoten jeweils eine Position zugewiesen. Dies geschieht mit Hilfe der Platzierungstechnik aus Kapitel 4.3.2. Schließlich wird die Hauptsimulation gestartet und erzeugt ein Nachfolgelayout. Durch eine abschließende Animation werden die Änderungen am Layout veranschaulicht. Der Vorgang kann beliebig wiederholt werden.

Hier möchte ich anmerken, dass es während der Entwicklung des Prototypen keinen Server gab. Der Prototyp wurde aber so konzipiert, dass er für die Anwendung mit einem Server ausgelegt ist. Deshalb habe ich Serveranfragen und Serverantworten simuliert, um so den vorgesehenen Ablauf realitätsnah abzubilden. Der Prototyp enthält deshalb lokal auch den Servergraphen.

7. Workshop-Feedback

Im Rahmen der Bachelorarbeit wurde die Möglichkeit wahrgenommen, an einem Workshop teilzunehmen, um dort den entwickelten Prototypen vorzustellen. In einem Kurzvortrag habe ich die visuellen Elemente und Funktionen des Prototypen präsentiert. Anschließend hab ich eine kurze Demo des Prototypen vorgeführt. Die Teilnehmer des Workshops waren überwiegend Geisteswissenschaftler*innen aus verschiedenen Fachbereichen, die sich mit Linked (Open) Data und Cultural Heritage beschäftigen. Die Frage, wie man diese Daten visuell explorieren kann, ist in diesem Zusammenhang von Interesse. Aus diesem Grund habe ich diesen Workshop genutzt um Feedback für die Ego-Graph-basierte visuelle Exploration von semantischen Wissensgraphen zu bekommen.

Ein Teilnehmer hat die Anmerkung gebracht, dass es die Exploration erleichtern würde, wenn zusätzlich zu der Bezeichnung von Entitäten eine kurze Beschreibung angezeigt wird. Außerdem wurde gewünscht, dass in der Anwendung eine Legende für die verschiedenen visuellen Elemente und Konzepte hinzugefügt wird, um die Verständlichkeit zu erhöhen. Eine weitere Rückmeldung war, dass man zurzeit nicht sehen kann, ob sich hinter Knoten weitere Relationen verbergen. Dazu muss man das Knotenmenü öffnen. Es wäre hilfreich, wenn Knoten, die „Sackgassen“ sind, gekennzeichnet werden. Eine weitere Anmerkung war, dass die Anwendung momentan sehr Entitäts-lastig ist und nur Relationen zwischen Entitäten explorierbar sind. Die Teilnehmer des Workshops arbeiten in der Praxis mit Wissensgraphen, die auf Entität-Event Relationen basieren. Das heißt Entitäten sind nicht direkt, sondern nur indirekt über Events miteinander verbunden. Ein Event wäre dann eine Kante, die mehr als zwei Entitäten verbinden müsste. Diese Art der Relation ist nicht Gegenstand meiner Arbeit, kann aber als zukünftiger Anwendungsfall abgewogen werden.

Außerdem wurden Funktionen genannt, die man gerne in der Anwendung hätte, wenn diese Art der Graphenexploration weiter verfolgt wird. So wäre es hilfreich, wenn man für zwei Entitäten den kürzesten Pfad berechnen kann. Desweiteren wurde der Vorschlag gebracht, dass das Bearbeiten von Relationen zusätzlich zur Exploration, ebenfalls interessant sein kann. Als Beispiel wurde genannt, fehlende Relationen in das Knoten-Kanten-Diagramm einfügen zu können. Außerdem wäre es für die Nutzung hilfreich, wenn Kurzbefehle für die einzelnen Funktionen der Anwendung hinzugefügt werden.

Über die generelle Einsetzbarkeit des Prototypen wurden ebenfalls Meinungen abgegeben. Die angedachte explorative Nutzung konnten sich die Geisteswissenschaftler*innen gut vorstellen. Der Prototyp bietet eine Möglichkeit Relationen und Entitäten zu finden, nach denen nicht explizit gesucht wird. Für einen ebenfalls aufgekommenen Anwendungszweck, die Netzwerkanalyse, konnten sie sich das nicht vorstellen.

Das Feedback hat Aufschlüsse darüber gegeben, wie der Prototyp von potentiellen Nutzer*innen wahrgenommen wird. Über den Hauptteil der Arbeit, nämlich die Erstellung eines dynamischen Layoutalgorithmus, wurde sich auf Grund des fehlenden technischen Hintergrunds der Teilnehmer*inne, nicht ausgetauscht. Aus den erhaltenden Rückmeldung lassen sich hauptsächlich Features

ableiten, die in zukünftigen Iterationen der Entwicklung berücksichtigt werden können. Als Explorationswerkzeug für Entitäten in einem Wissensgraphen wurde der Prototyp vom Publikum als potentiell hilfreiche Möglichkeit angenommen.

8. Ergebnisse und Diskussionen

In dieser Bachelorarbeit wurde ein Prototyp für die Ego-Graph-basierte visuelle Exploration semantischer Wissensgraphen entwickelt. Dieser basiert auf der inkrementellen Expansion eines Ego-Graphen. Der Prototyp soll die folgenden Probleme berücksichtigen: Zum einen soll es möglich sein, dass die Relationen einer Entität in aggregierter Form für Nutzer*innen angezeigt werden, um eine Übersicht über verbundene Entitäten zu geben. Diese aggregierten Entitäten sollen außerdem intuitiv in einer Vorschau angezeigt werden können. Wenn Nutzer*innen sich entscheiden, einen Knoten zu expandieren und adjazente Knoten in das Layout hinzuzufügen, sollen diese sich ins vorhandene Layout einfügen, sodass dies mit minimalen Veränderungen geschieht, um das mentale Bild der Nutzer*innen möglichst wenig zu stören. Ob für diese Problemstellungen Lösungen gefunden wurden und wie diese umgesetzt wurden, wird in diesem Kapitel diskutiert. Außerdem wird auf eventuelle Schwächen und Limitationen des Prototypen eingegangen und diskutiert, wie diese einzuordnen sind.

Die Umsetzung des dynamischen LA hat in dieser Arbeit den größten Stellenwert eingenommen und damit den größten Teil der Arbeit beansprucht. Hier wurden vor allem Algorithmen und Techniken aus verwandten Arbeiten betrachtet. Diese wurden dann bestmöglich im Prototypen implementiert und untereinander verglichen. Der Vergleich erfolgte durch wiederholtes Ausführen der Anwendung mit verschiedenen Einstellungen. So wurden in der Anwendung vorübergehend Buttons hinzugefügt, mit denen zwischen den einzelnen Layoutalgorithmen gewechselt werden konnte. Mit den unterschiedlichen Layoutalgorithmen wurden nacheinander die Graphen exploriert. Dadurch, dass der selbe Graph mit unterschiedlichen Layoutalgorithmen exploriert wurde, konnte ein Unterschied festgemacht werden. Aus Zeitgründen konnte keine ausführliche Nutzerstudie durchgeführt werden, um die Qualität der Layoutalgorithmen zu vergleichen. Um trotzdem ein gewisses Maß an Aussagekraft über die Effektivität der einzelnen Layoutalgorithmen zu bekommen, wurden für den Vergleich objektive Kriterien herangezogen. Dazu gehören das Überlappen von Knoten und ob sich viele Kanten schneiden und wie stark sich Knoten relativ zueinander bewegen. Dabei sind in der Entwicklung vor allem die Unterschiede zwischen dem verwendeten Zufallsgraphen und dem Beispielgraphen aufgefallen. Layoutalgorithmen, die im Zufallsgraphen noch zu den Favoriten gehörten, sind im Beispielgraphen teilweise unbrauchbar gewesen. Dies ist auf die unterschiedliche Struktur und Kantendichte der Graphen zurückzuführen. Im Beispielgraphen wurde der größte Erfolg mit dem in Kapitel 4.5.5 vorgestellten LA erzielt. Hier wird mit Hilfe einer Kollisionskraft Platz für Knoten reserviert, die noch nicht dem sichtbaren Graphen hinzugefügt wurden. Das wird erreicht, in dem sichtbare Knoten eine sehr großen Kollisionsradius haben, der abhängig von der Anzahl der unsichtbaren Nachbarn ist.

Der im finalen Prototypen umgesetzte LA hat einige Limitationen, die bisher noch nicht angesprochen wurden. Im Kapitel 4.5.2 wird ein Entwurf vorgestellt, Communities in die Layoutberechnung mit einzubeziehen. Die Umsetzung dieses Entwurfs hat leider nicht den gewünschten Erfolg gehabt. Zusätzlich stellen diese stark vernetzten Strukturen ein Problem im Layout dar. Gerade wenn die Kantendichte sehr hoch ist, wird das Layout schnell unübersichtlich. In Abbildung 8.1 ist ein Beispiel

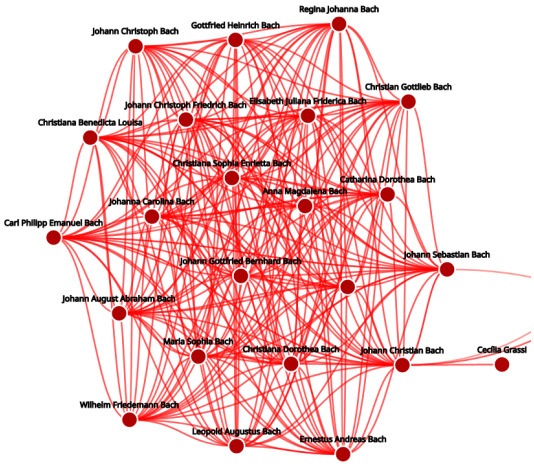
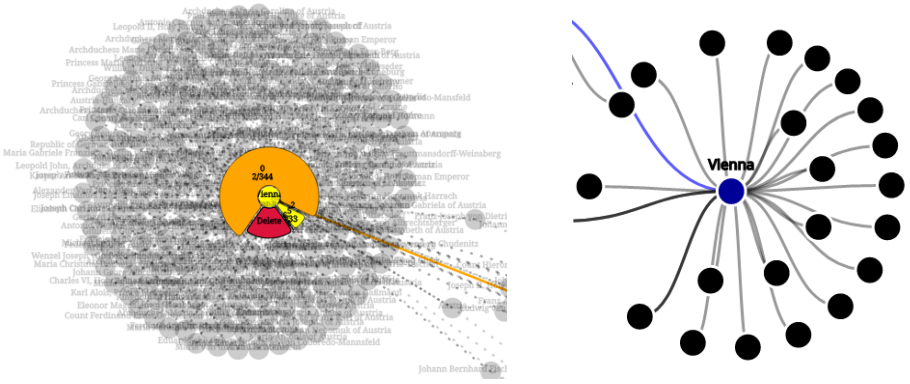


Abbildung 8.1.: Hier ist ein Ausschnitt aus der Graph-Exploration zu sehen. Der Knoten *Johann Sebastian Bach* und seine Nachbarn sind untereinander sehr stark mit Kanten verbunden.

für eine sehr stark vernetzte Community zu sehen. Dieses Problem ist nicht neu und wurde bereits in anderen Forschungsarbeiten thematisiert. Als Beispiel ist hier die Arbeit von Henry et al. [HFM07] zu nennen, die solche stark vernetzten Strukturen als Matrizen im Knoten-Kanten-Diagramm darstellt. Eine alternative Herangehensweise ist es, die Kanten zwischen den Knoten zu bündeln und damit übersichtlicher darzustellen. Holten et al. [HV09] bündeln Kanten mit einer kräftebasierten Simulation, indem sie entlang der Kanten Knoten platzieren, die sich gegenseitig anziehen. Dadurch wird die Unübersichtlichkeit im Layout reduziert. Solche Techniken bieten auf jeden Fall noch Potential zur Optimierung des Layouts, da sie in der Umsetzung des Prototypen nicht berücksichtigt wurden. Insgesamt ist die Darstellung der Relationen zwischen Entitäten in der Entwicklung zu kurz gekommen. Nicht nur die Darstellung der Kanten, wie eben beschrieben, ist suboptimal, auch die Informationen, die diese Kanten repräsentieren sollen, sind schwer für Nutzer*innen zugänglich. Die Beschreibung der Relation ist nur einsehbar, wenn man die zugehörige Kante mit dem Mauszeiger fokussiert, da diese dann eingeblendet wird. Wie bereits besprochen ist das aber nur möglich, wenn diese durch keine andere Kante verdeckt wird. Außerdem wird die Richtung der Kanten auch nicht visualisiert. Beide Probleme entstehen dadurch, dass Kanten durch ein einfaches SVG-path Element repräsentiert werden, die durch vorgegebene Punkte verlaufen. Hier bietet es sich an eine komplexere Darstellung zu wählen, die die Informationen einer Relation besser verdeutlichen kann.

Ein weiterer Bestandteil dieser Arbeit war die Aggregation und Vorschau von adjazenten Knoten, ausgehend von einem ausgewählten Knoten. Die Umsetzung dafür wurde sehr stark durch die beschriebene Vorladetechnik aus Kapitel 4.2 beeinflusst. Für Nutzer*innen soll ein Überblick explorierbarer Entitäten gegeben werden, die sie dann dem Layout hinzufügen können. In der praktischen Umsetzung ist das mit Herausforderungen verbunden, denn die Anzahl der aufdeckbaren Entitäten kann für manche Knoten in einem Graphen sehr hoch sein. In Abbildung 8.2 a) sieht man das Problem am Beispiel der Entität *Vienna*. Hier sind auch die vorgeladenen Knoten im Layout sichtbar gemacht, um das Problem besser zu verdeutlichen. Für die Entität *Vienna* sind alle adjazenten Knoten ins Layout vorgeladen, unabhängig davon, ob diese Knoten mit ausgehenden oder eingehenden Kanten verbunden sind. In diesem speziellen Beispiel sind es knapp unter 400 Knoten, die alleine für die Entität *Vienna* vorgeladen werden. Dass so eine hohe Menge an vorgeladenen



(a) **(b)**

Abbildung 8.2.: In Abbildung a) ist eine frühe Version des Prototypen zu sehen. Hier wurden alle, mit *Vienna* durch Kanten verbundene Knoten, im Layout vorgeladen, hier als transparente Kreise visualisiert. In Abbildung b) ist die finale Version des Prototypen zu sehen. Hier werden nur Knoten, die durch ausgehende Kanten mit *Vienna* verbunden sind, dem Layout hinzugefügt. Hier als schwarze Kreise visualisiert. Das reduziert die vorgeladenen Nachbarknoten des Knotens *Vienna* von fast 400 auf 22.

Knoten nicht wünschenswert ist, wurde in Kapitel 4.2 bereits festgestellt. Außerdem ergeben sich daraus auch Nachteile für die Aggregation und Vorschau der Knoten. Wenn Nutzer*innen die adjazenten Knoten für den Knoten *Vienna* hinzufügen möchten, dann wären es auf einen Schlag knapp unter 400 Knoten. Aus diesem Grund wurden nur Knoten vorgeladen, die durch ausgehende Kanten erreichbar sind. Dadurch konnte die Anzahl der vorgeladenen Knoten stark reduziert werden. Im finalen Prototypen sind nur noch 22 Knoten für die Entität *Vienna* vorgeladen. Zusätzlich dazu wurden die aggregierten Relationen in Kategorien unterteilt, um den Nutzer*innen Kontrolle darüber zu geben, welche Relationen sie aufdecken möchten. Die Vorschau dieser Relationen wird durch temporäres Einblenden der ausgewählten vorgeladenen Knoten erreicht. Das bietet sich insofern an, als damit diese Knoten bereits im Layout sind und damit nicht weit entfernt von ihrer späteren Position. Allerdings hat diese Art der Vorschau den Nachteil, dass nicht immer alle Labels dieser Knoten angezeigt werden, wenn diese überlappen würden. Für eine Vorschau über diese Knoten ist das nicht wünschenswert. Hier ist eine Verbesserung notwendig. Angehen könnte man das Problem, in dem man Knoten in der Vorschau noch mehr Platz verschafft, damit Labels nicht überlappen können.

Zusätzlich zum Hinzufügen von Knoten können auch Knoten aus dem Layout entfernt werden. Dies kann entweder durch aktives Löschen der Nutzer*innen geschehen, oder Knoten werden von der Anwendung passiv gelöscht. Im Prototypen werden die Knoten automatisch entfernt, die beim Expandieren von neuen Knoten mindestens vier Hops vom expandierten Knoten entfernt sind. Dieses passive Löschen von Knoten hilft dabei, die Anzahl der Knoten und Kantenkreuzungen gering zu halten, und das Layout bleibt übersichtlich. Allerdings werden hier Knoten aus dem Layout entfernt, unabhängig davon, wie lange diese im Layout sind, und ob Nutzer*innen diese entfernt haben möchten. Hier kann man Nutzer*innen noch mehr einbeziehen, indem man Knoten anhand ihrer Relevanz für Nutzer*innen löscht. Das lässt sich beispielsweise mit einer DOI-Funktion

umsetzten. Hier könnte man die Zeit die ein Knoten im Layout verbracht hat, in die DOI-Funktion mit einfließen lassen. Außerdem könnte man Nutzer*innen die Möglichkeit einräumen, Knoten zu markieren, damit sie vor dem automatischen Löschen geschützt sind.

9. Zusammenfassung und Ausblick

Das Ziel dieser Bachelorarbeit ist es einen Prototypen für die Ego-Graph-basierte visuelle Exploration semantischer Wissensgraphen basierend auf inkrementeller Expansion zu entwickeln. Diese Exploration soll auf einem Knoten-Kanten-Diagramm durchgeführt werden und mit Hilfe von modernen Webtechnologien wie TypeScript, D3 und WebPack umgesetzt werden. Der Prototyp soll die folgenden Probleme berücksichtigen: Es soll eine Übersicht von Relationen einer Entität geben, die in aggregierter Form angezeigt werden können. Außerdem sollen die Entitäten hinter den aggregierten Relationen in einer intuitiven Vorschau einsehbar sein. Ebenfalls soll bei der Expansion eines Knotens, bei der adjazente Knoten hinzugefügt werden, das Graphlayout sich nur minimal verändern.

Um diese Anforderungen umzusetzen habe ich mich zu Beginn der Arbeit in die Webtechnologien TypeScript und D3 eingearbeitet. Währenddessen habe ich eine ausführliche Recherche zum aktuellen Stand der Forschung bezüglich der Thematik der inkrementellen Layoutentwicklung durchgeführt. Dafür wurde nach ein einer Möglichkeit gesucht, wie man die Anforderung von minimaler Veränderung im Layout definieren kann. Die Arbeit von Misue et al. [MELS95] habe ich als Anhaltspunkt genommen, was man für eine minimale Veränderung im Layout beachten muss, und, wie das mentale Bild der Nutzer*innen erhalten werden kann. Anhand dieser Anforderung wurden verschiedene Layoutalgorithmen und Techniken bewertet, die im Prototypen umgesetzt wurden. Anschließend habe ich die am besten funktionierenden Layoutalgorithmen und Techniken als finale Implementierung im Prototypen ausgewählt. Diese Bachelorarbeit gibt also einen Überblick über verschiedene Techniken, mit denen ein dynamischer LA umgesetzt werden kann. Der ausgewählte LA im finalen Prototyp verfolgt einen anderen Ansatz, wie die vorgestellten Layoutalgorithmen aus anderen Arbeiten. Der ausgewählte LA legt den Fokus darauf, dass neu hinzugefügte Knoten genug Platz im Layout haben und dadurch weniger Einfluss auf vorhandene Knoten nehmen. Dies hat in der Praxis die besten Erfolge erzielt. Außerdem wurde ein Knotenmenü entworfen, welches es Nutzer*innen auf schnelle Weise ermöglicht Informationen über explorierbare Relationen zu bekommen. Sie können selber auswählen, welche Kategorie von Relationen exploriert werden soll. Die Knoten hinter den Relationen können mittels einer Vorschau im Layout eingeblendet werden. Zusätzlich können Nutzer*innen über das Knotenmenü Knoten aus dem Layout löschen. Knoten werden ebenfalls automatisch gelöscht, wenn Nutzer*innen einen Teil des Graphen exploriert haben, der vier Hops entfernt ist. Dadurch wird das Layout klein gehalten und die Übersichtlichkeit verbessert.

In dieser Arbeit wurde ein Lösungsansatz für die beschriebene Problemstellung ausgearbeitet, der letztendlich im finalen Prototypen umgesetzt ist. Der vorgestellte Lösungsansatz baut auf anderen Forschungsarbeiten auf, die als Grundlage für die Entwicklung genutzt wurden. Der finale Prototyp ist, auch wenn er die ego-basierte Exploration umsetzt, nur ein Ansatz für die Exploration von Wissensgraphen, die je nach Einsatzgebiet ganz andere Anforderungen erfüllen muss. Es lassen sich aber relevante Forschungsfragen ableiten, die in dieser Iteration der Entwicklung nicht umgesetzt werden konnten und Potential für weitere Forschung haben. Zum einen ist noch von Interesse, welche

Knoten automatisch aus dem Layout entfernt werden können, sodass das Layout übersichtlich bleibt und gleichzeitig die für Nutzer*innen relevanten Knoten erhalten bleiben. Wie man das mit Hilfe einer DOI-Funktion umsetzten kann ist ungeklärt. Außerdem ist es noch von Interesse inwiefern Communities in einem dynamischen Layout eingesetzt werden können, um die Übersichtlichkeit und die Berechnungsdauer eines Layouts zu verbessern. In Hinblick auf die Beständigkeit von solchen Communities, im Verlauf von Layoutanpassungen, ist noch Klärungsbedarf. Zu Letzt sind auch noch Fragen offen geblieben, wie die Visualisierung von Kanten umgesetzt werden kann, sodass die Informationen dieser Kanten intuitiv einsehbar sind, diese aber nicht die Übersichtlichkeit des Layouts mindern.

Literaturverzeichnis

[BGLL08] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre. „Fast unfolding of communities in large networks“. In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Okt. 2008), P10008. DOI: 10.1088/1742-5468/2008/10/p10008. URL: https://doi.org/10.1088/1742-5468/2008/10/p10008 (zitiert auf S. 31).

[BH86] J. Barnes, P. Hut. „A hierarchical O (N log N) force-calculation algorithm“. In: *Nature* 324.6096 (1986), S. 446–449. DOI: 10.1038/324446a0 (zitiert auf S. 13, 29, 35).

[BOH11] M. Bostock, V. Ogievetsky, J. Heer. „D³ Data-Driven Documents“. In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), S. 2301–2309. DOI: 10.1109/TVCG.2011.185 (zitiert auf S. 39).

[CCM15] T. Crnovrsanin, J. Chu, K.-L. Ma. „An Incremental Layout Method for Visualizing Online Dynamic Graphs“. In: *Graph Drawing and Network Visualization*. Hrsg. von E. Di Giacomo, A. Lubiw. Cham: Springer International Publishing, 2015, S. 16–29. ISBN: 978-3-319-27261-0 (zitiert auf S. 15, 24, 25, 32, 33).

[FR91] T. M. Fruchterman, E. M. Reingold. „Graph drawing by force-directed placement“. In: *Software: Practice and experience* 21.11 (1991), S. 1129–1164 (zitiert auf S. 13, 15, 24).

[FT04] Y. Frishman, A. Tal. „Dynamic Drawing of Clustered Graphs“. In: *IEEE Symposium on Information Visualization*. 2004, S. 191–198. DOI: 10.1109/INFVIS.2004.18 (zitiert auf S. 31).

[HEHB16] W. Huang, P. Eades, S.-H. Hong, H. Been-Lirn Duh. „Effects of curves on graph perception“. In: *IEEE Pacific Visualization Symposium (PacificVis)*. 2016, S. 199–203. DOI: 10.1109/PACIFICVIS.2016.7465270 (zitiert auf S. 40).

[HFM07] N. Henry, J.-D. Fekete, M. J. McGuffin. „NodeTrix: a Hybrid Visualization of Social Networks“. In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (2007), S. 1302–1309. DOI: 10.1109/TVCG.2007.70582 (zitiert auf S. 50).

[HP09] F. van Ham, A. Perer. „“Search, Show Context, Expand on Demand”: Supporting Large Graph Exploration with Degree-of-Interest“. In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (2009), S. 953–960. DOI: 10.1109/TVCG.2009.108 (zitiert auf S. 16).

[Hu05] Y. Hu. „Efficient, high-quality force-directed graph drawing“. In: *Mathematica Journal* 10.1 (2005), S. 37–71 (zitiert auf S. 13).

[HV09] D. Holten, J. J. Van Wijk. „Force-directed edge bundling for graph visualization“. In: *Computer Graphics Forum*. Bd. 28. 3. Wiley Online Library. 2009, S. 983–990 (zitiert auf S. 50).

[LLY11] C.-C. Lin, Y.-Y. Lee, H.-C. Yen. „Mental map preserving graph drawing using simulated annealing“. In: *Information Sciences* 181.19 (2011), S. 4253–4272. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2011.06.005. URL: https://www.sciencedirect.com/science/article/pii/S0020025511002799 (zitiert auf S. 15).

[LPP+06] B. Lee, C. Parr, C. Plaisant, B. Bederson, V. Veksler, W. Gray, C. Kotfila. „TreePlus: Interactive Exploration of Networks with Enhanced Tree Layouts“. In: *IEEE Transactions on Visualization and Computer Graphics* 12.6 (2006), S. 1414–1426. DOI: 10.1109/TVCG.2006.106 (zitiert auf S. 15).

[MELS95] K. Misue, P. Eades, W. Lai, K. Sugiyama. „Layout Adjustment and the Mental Map“. In: *Journal of Visual Languages & Computing* 6.2 (1995), S. 183–210. ISSN: 1045-926X. DOI: https://doi.org/10.1006/jvlc.1995.1010. URL: https://www.sciencedirect.com/science/article/pii/S1045926X85710105 (zitiert auf S. 15, 21, 29, 53).

[New18] M. Newman. *Networks*. Oxford University Press, 2018 (zitiert auf S. 13, 17, 31).

[Ngu21] F. Nguyen. „Leiden-Based Parallel Community Detection“. Bachelor's thesis. Karlsruhe Institute of Technology, 2021 (zitiert auf S. 31).

[PAKC15] R. Pienta, J. Abello, M. Kahng, D. H. Chau. „Scalable graph exploration and visualization: Sensemaking challenges and opportunities“. In: *International Conference on Big Data and Smart Computing (BIGCOMP)*. 2015, S. 271–278. DOI: 10.1109/35021BIGCOMP.2015.7072812 (zitiert auf S. 16).

[SMK13] M. Steiger, T. May, J. Kohlhammer. „Stable Incremental Layouts for Dynamic Graph Visualizations“. In: *IADIS International Journal on Computer Science & Information Systems* 8.2 (2013) (zitiert auf S. 15, 22, 24, 25, 34).

[TS13] Y. Tu, H.-W. Shen. „GraphCharter: Combining browsing with query to explore large semantic graphs“. In: *2013 IEEE Pacific Visualization Symposium (PacificVis)*. 2013, S. 49–56. DOI: 10.1109/PacificVis.2013.6596127 (zitiert auf S. 41).

[XRP+12] K. Xu, C. Rooney, P. Passmore, D.-H. Ham, P. H. Nguyen. „A User Study on Curved Edges in Graph Visualization“. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), S. 2449–2456. DOI: 10.1109/TVCG.2012.189 (zitiert auf S. 40).

Alle URLs wurden zuletzt am 23. 09. 2022 geprüft.

A. Quellcode aus D3

Die folgenden Listings dienen der Veranschaulichung von Ausschnitten der D3-Kräfte-Simulation und wurden dem GitHub Repository der D3-Bibliothek[1] entnommen.

```
function tick(iterations) {
    var i, n = nodes.length, node;

    if (iterations === undefined) iterations = 1;

    for (var k = 0; k < iterations; ++k) {
      alpha += (alphaTarget - alpha) * alphaDecay;

      forces.forEach(function(force) {
        force(alpha);
      });

      for (i = 0; i < n; ++i) {
        node = nodes[i];
        if (node.fx == null) node.x += node.vx *= velocityDecay;
        else node.x = node.fx, node.vx = 0;
        if (node.fy == null) node.y += node.vy *= velocityDecay;
        else node.y = node.fy, node.vy = 0;
      }
    }

    return simulation;
  }
```

Listing A.1: D3 Quellcode der *Tick*-Funktion

```
function force(alpha) {
    for (var k = 0, n = links.length; k < iterations; ++k) {
      for (var i = 0, link, source, target, x, y, l, b; i < n; ++i) {
        link = links[i], source = link.source, target = link.target;
        x = target.x + target.vx - source.x - source.vx || jiggle(random);
        y = target.y + target.vy - source.y - source.vy || jiggle(random);
        l = Math.sqrt(x * x + y * y);
        l = (l - distances[i]) / l * alpha * strengths[i];
        x *= l, y *= l;
        target.vx -= x * (b = bias[i]);
        target.vy -= y * b;
        source.vx += x * (b = 1 - b);
        source.vy += y * b;
      }
    }
```

[1] https://github.com/d3/d3-force

```
16    }
```

Listing A.2: D3 Quellcode der Anziehenden Kräfte

```
1  function force(_) {
2      var i, n = nodes.length, tree = quadtree(nodes, x, y).visitAfter(accumulate);
3      for (alpha = _, i = 0; i < n; ++i) node = nodes[i], tree.visit(apply);
4    }
```

Listing A.3: D3 Quellcode der Abstoßenden Kräfte

```
1  function accumulate(quad) {
2      var strength = 0, q, c, weight = 0, x, y, i;
3
4      // For internal nodes, accumulate forces from child quadrants.
5      if (quad.length) {
6        for (x = y = i = 0; i < 4; ++i) {
7          if ((q = quad[i]) && (c = Math.abs(q.value))) {
8            strength += q.value, weight += c, x += c * q.x, y += c * q.y;
9          }
10       }
11       quad.x = x / weight;
12       quad.y = y / weight;
13     }
14
15     // For leaf nodes, accumulate forces from coincident quadrants.
16     else {
17       q = quad;
18       q.x = q.data.x;
19       q.y = q.data.y;
20       do strength += strengths[q.data.index];
21       while (q = q.next);
22     }
23
24     quad.value = strength;
25   }
```

Listing A.4: D3 Quellcode der *Accumulate*-Funktion

```
1  function apply(quad, x1, _, x2) {
2      if (!quad.value) return true;
3
4      var x = quad.x - node.x,
5          y = quad.y - node.y,
6          w = x2 - x1,
7          l = x * x + y * y;
8
9      // Apply the Barnes-Hut approximation if possible.
10     // Limit forces for very close nodes; randomize direction if coincident.
11     if (w * w / theta2 < l) {
12       if (l < distanceMax2) {
13         if (x === 0) x = jiggle(random), l += x * x;
14         if (y === 0) y = jiggle(random), l += y * y;
15         if (l < distanceMin2) l = Math.sqrt(distanceMin2 * l);
16         node.vx += x * quad.value * alpha / l;
17         node.vy += y * quad.value * alpha / l;
18       }
19       return true;
```

```
    }

    // Otherwise, process points directly.
    else if (quad.length || l >= distanceMax2) return;

    // Limit forces for very close nodes; randomize direction if coincident.
    if (quad.data !== node || quad.next) {
      if (x === 0) x = jiggle(random), l += x * x;
      if (y === 0) y = jiggle(random), l += y * y;
      if (l < distanceMin2) l = Math.sqrt(distanceMin2 * l);
    }

    do if (quad.data !== node) {
      w = strengths[quad.data.index] * alpha / l;
      node.vx += x * w;
      node.vy += y * w;
    } while (quad = quad.next);
  }
```

Listing A.5: D3 Quellcode der *Apply*-Funktion

```
function force() {
    var i, n = nodes.length,
        tree,
        node,
        xi,
        yi,
        ri,
        ri2;

    for (var k = 0; k < iterations; ++k) {
      tree = quadtree(nodes, x, y).visitAfter(prepare);
      for (i = 0; i < n; ++i) {
        node = nodes[i];
        ri = radii[node.index], ri2 = ri * ri;
        xi = node.x + node.vx;
        yi = node.y + node.vy;
        tree.visit(apply);
      }
    }
```

Listing A.6: D3 Quellcode der Kollisionskraft

```
 function prepare(quad) {
    if (quad.data) return quad.r = radii[quad.data.index];
    for (var i = quad.r = 0; i < 4; ++i) {
      if (quad[i] && quad[i].r > quad.r) {
        quad.r = quad[i].r;
      }
    }
  }
```

Listing A.7: D3 Quellcode der *Prepare*-Funktion

```
   function apply(quad, x0, y0, x1, y1) {
      var data = quad.data, rj = quad.r, r = ri + rj;
      if (data) {
        if (data.index > node.index) {
```

```
5           var x = xi - data.x - data.vx,
6               y = yi - data.y - data.vy,
7               l = x * x + y * y;
8           if (l < r * r) {
9             if (x === 0) x = jiggle(random), l += x * x;
10            if (y === 0) y = jiggle(random), l += y * y;
11            l = (r - (l = Math.sqrt(l))) / l * strength;
12            node.vx += (x *= l) * (r = (rj *= rj) / (ri2 + rj));
13            node.vy += (y *= l) * r;
14            data.vx -= x * (r = 1 - r);
15            data.vy -= y * r;
16          }
17        }
18        return;
19      }
20      return x0 > xi + r || x1 < xi - r || y0 > yi + r || y1 < yi - r;
21    }
22  }
```

Listing A.8: D3 Quellcode der *Apply*-Funktion (Kollisionskraft)

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift