Bachelorarbeit

Übersichtliches Zeichnen von literarischen Netzwerken mittels des Sugiyama-Frameworks

Jonas Barth

Abgabedatum: 13. Juli 2022
Betreuer: Prof. Dr. Alexander Wolff
Prof. Dr. Fotis Jannidis
Johannes Zink

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Zusammenfassung

Das Sugiyama-Framework beschreibt einen algorithmischen Leitfaden, mit der eine hierarchische Zeichnung für einen beliebigen Graphen $G = (V, E)$ erstellt werden kann. In der Forschung zum Thema Graphzeichnen ist es sehr populär. Jedoch lässt die Effizienz zu Wünschen übrig, denn im Worst-Case werden $O(|V||E| \log |E|)$ Rechenschritte benötigt, sowie Speicherplatz in der Größenordnung $O(|V||E|)$. Für diese Arbeit habe ich das Sugiyama-Framework implementiert, um literarische Netzwerke als lagenbasierte Graphen mit besonderen visuellen Merkmalen darzustellen. Dabei habe ich zudem veränderte Vorgehensweisen, im Vergleich zur klassischen Umsetzung genutzt und so die Laufzeit auf $O((l \cdot |E| + |V|) \log |E|)$ reduziert, wobei l der Anzahl von Lagen im Graph entspricht. Der benötigte Speicherplatz konnte auf $O(|V| + |E|)$ verringert werden. Letztlich bin ich auf weitere Ansätze zur Minderung von Kantenkreuzungen gestoßen, mit welchen das Endergebnis noch übersichtlicher wird. Darunter ist ein Nachbearbeitungsschritt, mit dem sich die Anzahl von Kreuzungen in kleinen und mittleren Graphen um durchschnittlich 8% im Vergleich zur normalen Umsetzung reduzieren lässt.

Abstract

The Sugiyama framework is a popular scheme for creating a hierarchical drawing of an arbitrary graph $G = (V, E)$. It is very common in the field of graph drawing. But the original approach can be regarded as inefficient. The wort-case running time is $O(|V||E| \log |E|)$, while $O(|V||E|)$ space is used. For this paper I have implemented Sugiyama's algorithm to construct a drawing of specific $n \times n$ matrices with the use of distinctive visual features. While doing this I used newer approaches than the original paper suggests and therefore reduced the runtime complexity to $O((l \cdot |E| + |V|) \log |E|)$, with l being the number of layers in the graph. The required memory is now only $O(|V| + |E|)$. Furthermore I came up with alternative methods for the reduction of edge crossings, which raise the readability of the final hierarchical graph. Most notably a postprocessing step, that decreased the number of crossings in small and medium sized graphs by about 8% on average, when compared to the normal implementation.

Inhaltsverzeichnis

1 Einleitung **4**
1.1 Problemstellung 4
1.1.1 Datensatz an literarischen Werken 4
1.1.2 Ziele der Visualisierung 5
1.2 Eigener Beitrag 5

2 Sugiyama-Framework **7**
2.1 Dummy-Knoten 8
2.2 Kreuzungsreduzierung 8
2.2.1 Zählen der Kreuzungen 9
2.2.2 Einseitge Kreuzungsminimierung 10
2.2.3 Barycenter-Methode 11
2.3 Mehrlagige Kreuzungsreduzierung 11
2.4 Brandes-Köpf-Algorithmus 12
2.5 Reduzierung der Dummy-Knoten 13

3 Eigene Implementierung **17**
3.1 Merkmale der Visualisierung 17
3.1.1 Kantendicke 17
3.1.2 Kontrahieren von Kanten 18
3.1.3 Horizontale Verschiebung 19
3.2 Anwendung 20
3.2.1 Filter und sonstige Einstellungen 21
3.2.2 Beispiele 22

4 Verbesserungsvorschläge zur Kreuzungsreduzierung **24**
4.1 Gewichtete Kreuzungsreduzierung 24
4.2 Postprocessing 25
4.3 Test der Algorithmen 29

5 Diskussion **32**

6 Fazit **34**

Literaturverzeichnis **35**

1 Einleitung

In heutigen Forschungsarbeiten müssen häufig größere Datensätze gesammelt und analysiert werden. Es ist meist schwierig in diesen Sammlungen schnell und übersichtlich interessante Ergebnisse zu entdecken und deutlich zu machen. Je nach der Art von Daten können unterschiedliche Zeichnungen hilfreich sein, wie z.B. Balken- oder Kuchendiagramme. Aber auch ein Graph G, bestehend aus Knoten V und Kanten E, kann dem Betrachter bei der Aufnahme der Resultate helfen. Daraus stellt sich die Frage, wie man algorithmisch einen Graphen am besten zeichnen kann, um möglichst viele Informationen gleichzeitig darzustellen und wichtige Ergebnisse hervorzuheben. Jedoch möchte man den Betrachter auch nicht überfordern und unübersichtlich werden. Mit diesem Problem beschäftigt man sich im Forschungsbereich des Graphzeichnens, in welchem passende Algorithmen für unterschiedliche Anwendungsfälle gesucht werden.

In dieser Arbeit geht es speziell um das Zeichnen eines lagenbasierten Graphen. Das bedeutet, dass die Knoten auf vorgegebene, horizontal oder vertikal angeordnete Geraden verteilt und innerhalb diesen sortiert werden. Das im Gebiet des Graphzeichnens langjährig bekannte Sugiyama-Framework, gibt genau für diese Art von Graphen eine Orientierung. In seiner originalen Form ist es aber sehr speicherintensiv. Genauer gesagt hat es eine Laufzeit von $O(|V||E|\log|E|)$ und benötigt $O(|V||E|)$ an Platz im Worst-Case. Außerdem führt es nicht zwingend zu einer Zeichnung, die auf die Problemstellung zugeschnitten ist. Deshalb wird in dieser Arbeit eine effizientere Umsetzung, sowie zusätzliche optische Ideen vorgestellt, um einen bestimmten Datensatz zu visualisieren.

1.1 Problemstellung

Das Thema dieser Arbeit ist aufgrund einer speziellen Problemstellung gewählt und befasst sich vor allem mit einer konkreten Lösung dafür. Im folgenden Abschnitt wird das Problem und die Zielsetzung genauer beleuchtet.

1.1.1 Datensatz an literarischen Werken

Für die Arbeit wurde mir ein Datensatz vom Lehrstuhl für Computerphilologie der Universität in Würzburg zur Verfügung gestellt. Darin wurden 2390 literarische Werke aus den Jahren 1825 bis 1915, nach Form, Stil, Emotion und Inhalt analysiert. Als Ergebnis wurden für jedes der Kriterien euklidische Distanzen der Texte zueinander berechnet. Somit besteht der Datensatz aus einer $n \times n$-Matrix je Eigenschaft, in denen eine Zelle der Ähnlichkeit zwischen zwei Texten entspricht. Außerdem wurde eine weitere Matrix angelegt, in der alle Resultate kombiniert wurden. Des Weiteren konnten viele der Werke in eines der Genres: Elegie, Ballade, Lied und Sonett eingeteilt werden. Diese Einteilung

ist in einem Metafile vermerkt, das zusätzlich Titel, Autor und Erscheinungsjahr eines Textes enthält.

1.1.2 Ziele der Visualisierung

Bei der Darstellung des Datensatzes sollten folgende Informationen klar werden. Generell ist die Stärke der Ähnlichkeit wichtig. Diese muss deutlich sichtbar sein, damit der Betrachter die Daten einordnen kann. Wir wollen explizit Texte mit besonders vielen Übereinstimmungen zu anderen kenntlich machen. Um sozusagen die Durchschnittswerke dieser Zeit zu finden. Aber auch zwei einzelne Texte, die über eine ungewöhnlich hohe Ähnlichkeit verfügen, stechen so heraus. Des Weiteren wird ein besonderer Fokus darauf gelegt, Werke aus unterschiedlichen Zeitperioden im Vergleich zu sehen, denn eine hohe Ähnlichkeit zu zeitlich nahen Texten ist vorauszusehen und ist deshalb weniger relevant. Weitere wichtige Ziele mit Hinblick auf den Datensatz:

- zeige ob ein Werk insgesamt zukunftsprägend ist, oder sich mehr an früheren Texten orientiert hat
- verdeutliche Abhängigkeiten von einzelnen Zeitperioden zueinander
- veranschauliche Zusammenhang zwischen den verschiedenen Genres

All das sollte für den Betrachter der fertigen Zeichnung ablesbar sein, ohne ihn zu verwirren. Dafür muss die Visualisierung übersichtlich sein, das heißt vor allem möglichst wenige Kantenkreuzungen in der fertigen Zeichnung. Außerdem muss der Algorithmus der das Ergebnis liefert, effizient umsetzbar sein, denn es handelt sich um einen relativ großen Datensatz für einen Graphen.

1.2 Eigener Beitrag

Mit Hinblick auf diese Arbeit habe ich mich zunächst mit dem Sugiyama-Framework und weiteren Ansätzen zum Thema Graphzeichnen genauer beschäftigt. Dieses Wissen möchte ich in Kapitel 2 der Arbeit wiedergeben. Zusätzlich habe ich es aber genutzt, um eine Anwendung für das beschriebene Problem zu entwickeln. In dieser kann ein Nutzer einen Datensatz, bestehend aus einer $n \times n$-Matrix, sowie einem Metafile, eingeben. Durch verschiedene Filter und Einstellungen kann er sich daraufhin eine individuelle Visualisierung, in Form eines hierarchischen Graphen, ausgeben lassen. Die Darstellung erfüllt dabei die zuvor definierten Ziele. Es wird bei Ausführung der Anwendung ein Graph $G = (V, E)$ angelegt, wofür $O(n^2)$ Zeit gebraucht wird, um die gegebene $n \times n$-Matrix auszulesen. Für die Umwandlung in einen hierarchischen Graphen nach dem Sugiyama-Framework wird $O((l \cdot |E| + |V|) \log |V|)$ Zeit benötigt. Wobei l die Anzahl an Lagen darstellt. Der Speicherverbrauch beträgt $O(|V| + |E|)$.

In Kapitel 3 werde ich genauer auf die Implementierung der Anwendung eingehen. Dabei geht es speziell um besondere visuelle Merkmale, mit denen sie sich von anderen Umsetzungen unterscheidet, um die geforderten Ziele zu erreichen. Außerdem stelle ich

einige Entdeckungen aus dem oben genannten Datensatz vor, die durch die Visualisierung gemacht wurden.

Zusätzlich bin ich auf weitere Ansätze zur Reduzierung von Kantenkreuzungen in hierarchischen Graphen gestoßen, die meines Wissens noch nicht dokumentiert sind. Diese werde ich in Kapitel 4 vorstellen und deren Relevanz, anhand von durchgeführten Tests, einordnen.

Das dient dazu die gefundenen Ergebnisse einzuordnen und deren Relevanz zu diskutieren. Während in Kapitel 6 die Arbeit abschließend zusammengefasst wird.

2 Sugiyama-Framework

Zunächst ist das Ziel den gegebenen Datensatz in einen Graph $G = (V, E)$ umwandeln, durch den die Resultate leichter abzulesen sind. Dafür werden die literarischen Werke als Knoten interpretiert und die Ähnlichkeiten untereinander durch Kanten mit entsprechendem Gewicht dargestellt. Es stellt sich jedoch noch die Frage, was für eine Art von Graph bzw. Algorithmus zur Erstellung hier sinnvoll ist. Ein wichtiges Ziel bei der Visualisierung ist es, Abhängigkeiten zwischen verschiedenen Zeitperioden zu zeigen. Es ist sinnvoll, die zeitliche Zuordnung eines Knotens durch dessen x-Koordinate auszudrücken. Dadurch wäre diese für den Betrachter intuitiv ablesbar. Außerdem sind, wie bereits erwähnt, Übereinstimmungen zwischen Werken, die nur wenige Jahre auseinander erschienen sind, für unsere Zeichnung unwichtig. Diese beiden Bedingungen machen klar, dass eine hierarchische Zeichnung des Graphen hier passend ist. Bei so einer Zeichnung werden Knoten auf eine von mehreren aufeinander folgenden vertikalen, oder horizontalen Linien bzw. Lagen verteilt. Knoten haben keine Kanten zu Knoten in der eigenen Lage. In unserem Fall würde eine Lage einer Zeitspanne von beispielsweise 5 Jahren entsprechen. Ein einfaches Beispiel für so einen Graphen ist in Abbildung 2.1 zu sehen.

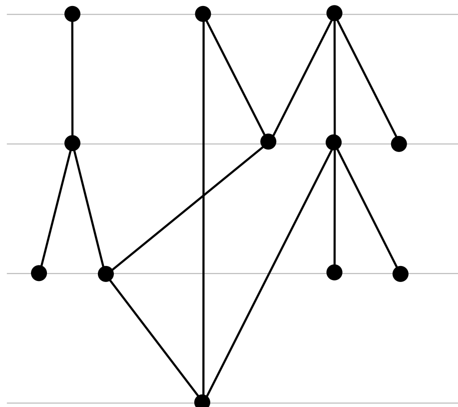

Abb. 2.1: Ein hierarchischer Graph mit 4 Lagen

Der Vorteil an einem hierarchischem Graphen ist, dass sich Kantenkreuzungen effektiv und effizient vermindern lassen und dass durch das bereits 1981 vorgestellte Sugiyama-Framework [STT81], ein Leitfaden existiert, wie ein beliebiger Graph Schritt für Schritt zu einem geschichteten Graph transformiert wird.

Die erste Aufgabe ist es, den Graph kreisfrei zu machen, um im folgenden die Knoten so auf die Lagen zuzuordnen, dass die wenigsten Überschneidungen auftreten. Diese

Arbeiten muss in unserem Fall nicht erledigt werden, denn durch die zeitliche Anordnung der Daten, haben wir bereits eine logische Verteilung der Knoten auf unterschiedliche Lagen und müssen damit arbeiten. Die restlichen Schritte des Sugiyama-Framework sind für uns relevanter: Einfügen von Dummy-Knoten, Verminderung der Kreuzungen und Glätten der Kanten. Diese Teilschritte werden im folgenden geklärt. In Abbildung 2.2 werden diese Schritte vereinfacht von links nach rechts dargestellt. Schwarze Vierecke stellen Dummy-Knoten dar.

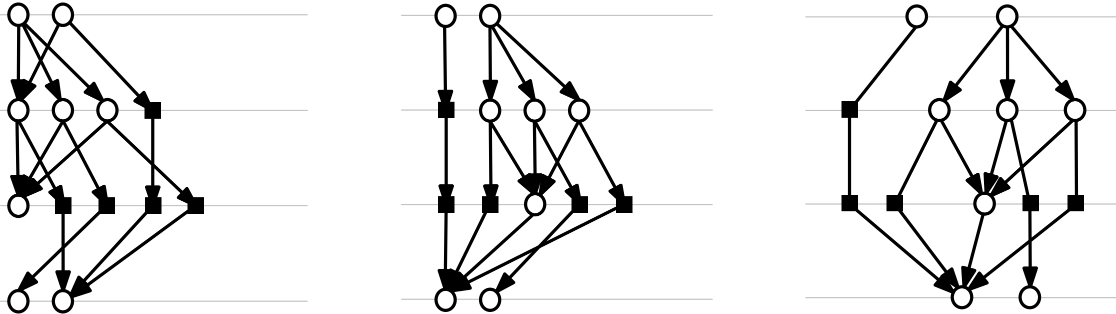

Abb. 2.2: Relevante Teilschritte des Sugiyama-Frameworks

2.1 Dummy-Knoten

Das Zeichnen von hierarchischen Graphen bringt zwei hauptsächliche Vorteile mit sich. Zum einen eine übersichtliche, stufenweise Unterteilung der Knoten. Zum anderen das erleichterte Vermindern von Kantenkreuzungen. Letzteres ist ein NP-schweres Problem und kann ohne besondere Einschränkungen des Grads oder der Knotenmenge schlecht approximiert werden, siehe [Tam13]. Hier kann aber eine solche Einschränkung gefunden werden, mit der das Problem vereinfacht wird. Und zwar müssen für lange, sich über mehrere Lagen erstreckende Kanten, sogenannte Dummy-Knoten auf jede Lage zwischen Start- und Endpunkt platziert werden. Daraufhin werden sie mit kürzeren Kanten untereinander verbunden. Nun verläuft jede Kante zwischen zwei direkt benachbarten Lagen, was für die gewünschte Einschränkung sorgt, wie wir später sehen werden. Das Hinzufügen der Dummy-Knoten kann jedoch sehr speicherintensiv sein. Tatsächlich liegt die Anzahl der Dummy-Knoten in $O(|V||E|)$, vgl. [Tam13]. An dieser Stelle wollen wir später in der Implementierung eine Verbesserung erzielen.

2.2 Kreuzungsreduzierung

Dank hinzugefügter Dummy-Knoten laufen alle Kanten des Graphen genau von einer zur nächsten benachbarten Lage. Zudem haben wir im Sugiyama-Framework keine Kanten zwischen Knoten der gleichen Stufe. Im Folgenden betrachten wir genau zwei aufeinanderfolgende Lagen L_1, L_2. Dabei können wir sehen, dass es sich hier um einen bipartiten Teilgraph handelt. Die beiden Knotenmengen L_1, L_2 haben nur Kanten $\{v, u\} \in E$

zwischen Knoten $v \in L_1, u \in L_2$. Durch diese Einschränkung im Teilgraph wird das Problem der Kreuzungsreduzierung jetzt einfacher und wird *einseitige Kreuzungsminimierung* genannt. Genauer gesagt versucht man hier die Kreuzungen zwischen zwei Lagen zu verringern, indem die Sortierung von L_2 verändert werden kann. Die Anordnung der Knoten in L_1 ist fest. Leider ist es weiterhin NP-schwer, siehe [Tam13], aber es gibt einige Heuristiken, mit denen sich gute Approximationen erzielen lassen.

Bevor wir uns diese ansehen, brauchen wir zunächst eine Methode um die Anzahl der Kreuzungen zwischen zwei Lagen zu zählen. Denn sonst kann in der späteren Anwendung nicht bestimmt werden, ob eine veränderte Sortierung von L_2 überhaupt empfehlenswert ist.

2.2.1 Zählen der Kreuzungen

Um die Anzahl von Kantenkreuzungen bei der einseitigen Kreuzungsminimierung zu ermitteln, gibt es bereits einige Algorithmen. Eine naive Implementierung wäre eine Kreuzungsmatrix in $O(|E|^2)$ Zeit zu erstellen und daraus das Ergebnis abzulesen. Sander [San94] entwickelte dafür eine effizientere Methode, welche in $O(|L_1|+|L_2|+|E|+C)$ liegt, mit C als Anzahl der Kreuzungen. Für unser Ergebnis ist aber keine Kreuzungsmatrix nötig und wir können deshalb schnellere Algorithmen nutzen, die nur die Anzahl der Kreuzungen als Ergebnis ausgeben. Nagamochi and Yamada [NY04] haben beispielsweise zwei Algorithmen, mit Laufzeiten von $O(|L_1||L_2|)$ und $O(\min\{|L_1||L_2|, E \log |L_s|\})$, vorgestellt, wobei L_s der kleineren der beiden Knotenmengen aus L_1, L_2 entspricht. Das Gegenstück zu L_s nennen wir L_b. Wir wollen einen etwas leichteren und ähnlich schnellen Algorithmus benutzen und zwar den von Barth et al. [BJM02].

Um den Algorithmus durchführen zu können, muss zuvor für jeden Knoten die Position in seiner Lage gespeichert werden. Nun werden die Kanten sortiert und zwar erst nach der Reihenfolge ihres Startknoten $v \in L_b$ und dann, bei Gleichheit, nach der Position der Endknoten $u \in L_s$. Dies ist mit dem bekannten Algorithmus Radixsort in $O(|E|)$ machbar. Für diese sortierte Liste betrachte man jetzt nur die Positionen der jeweiligen Endknoten in L_s und sortiert nach diesen mittels des Algorithmus Insertionsort. Jedoch wird für jede Vertauschung beim Sortieren, ein Zähler für die Anzahl an Kreuzungen erhöht. Das ist korrekt, denn jedes mal, wenn Kanten e_1, e_2 getauscht werden, hat e_1 einen Endknoten, der früher in der Lage L_s vorkommt, als der von e_2. Die Kante e_2 war aber zunächst vor e_1 in der ursprünglichen Anordnung, weil e_2 einen weiter vorne befindlichen Startknoten in L_b hat. Genau dann existiert eine Kreuzung zwischen den beiden.

Leider läuft Insertionsort in $O(|E|^2)$, aber das Verfahren wird in ähnlicher Art und Weise mit einem Binärbaum, auf dessen unterster Ebene alle Knoten der kleineren Knotenmenge L_s Platz finden, angewendet. Die Kanten werden, wie bereits zuvor, beschrieben sortiert. Aufsteigend werden Kanten an den Blättern des Baums eingefügt, die ihrem Knoten aus L_s entsprechen. Von dort wird für eine Kante jeder Elternknoten bis zur Wurzel im Binärbaum besucht und jeweils ein Zähler für den Knoten um 1 erhöht. Falls wir uns außerdem in einem linken Kindknoten befinden, addieren wir die Zahl im rechten Geschwisterknoten zur Anzahl der gefundenen Kreuzungen auf. Denn auch hier

bedeutet das, es wurden bereits Kanten eingefügt, deren Zielknoten später in L_s liegen, da sie einen rechten Geschwisterknoten besucht haben. Gleichzeitig haben diese Kanten Startknoten in L_b, die vor unserem Startknoten sind. All diese Kanten werden gekreuzt.

Das folgende Beispiel aus Abbildung 2.3 macht das Vorgehen klarer. Auf der linken Seite ist eine beliebige Sortierung von Knoten auf zwei Lagen zu sehen, darunter ist die, durch Radixsort angeordnete Liste, mit den jeweiligen Endpunkten. Rechts davon ist, in Schwarz, der Binärbaum abgebildet, wie er nach den ersten vier eingefügten Kanten aussieht. In Rot sind die Änderungen vermerkt, die sich bei Betrachten der fünften Kante ergeben. Die beiden unterstrichenen Werte, werden in diesem Durchlauf auf die Kreuzungszahl aufaddiert.

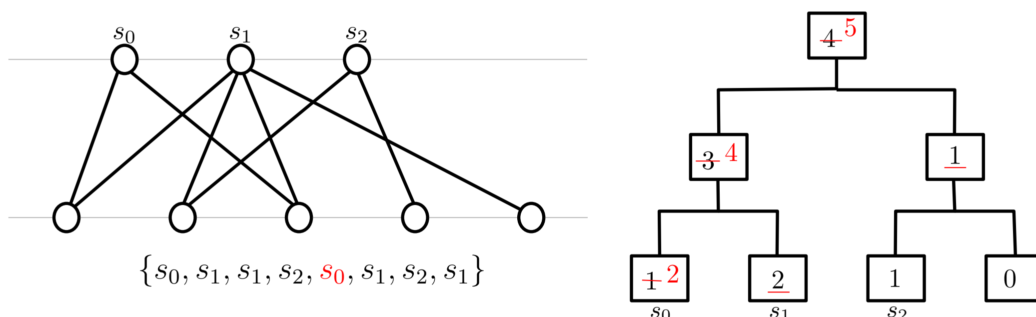

Abb. 2.3: Beispiel für das Kreuzungszählen mit dem Algorithmus von Barth et al. [BJM02]

Da der Binärbaum so konstruiert wird, dass die Anzahl der Blätter der untersten Ebene gerade so größer ist als $|L_s|$, haben wir eine Höhe $O(\log |L_s|)$. Wir haben schon festgestellt, dass für das Vorsortieren nur $O(|E|)$ gebraucht wird. Der Algorithmus von Barth et al. hat also eine Laufzeit von $O(|E| \log |V_s|)$, denn es müssen maximal $|E|$-Kanten im Baum eingefügt werden.

2.2.2 Einseitge Kreuzungsminimierung

Beim Problem der einseitigen Kreuzungsminierung betrachten wir also zwei Lagen L_1 und L_2. Die Lage L_1 hat eine feste Sortierung, während wir die Knoten aus L_2 frei verschieben können, um die Kreuzungen zu reduzieren.

Es wurde bereits erwähnt, dass das Problem eine optimale Lösung zu finden, NP-schwer ist. Zwar gibt es auch exakte Algorithmen, wie beispielsweise von Jünger und Mutzel [JM02], basierend auf der Relaxierung eines linearen Programms. Es kann aber nur auf Lagen angewendet werden, bei denen L_s höchstens 60 Knoten besitzt, bei größeren Instanzen wird die exakte Methode ineffizient. Es gibt jedoch einige Heuristiken, mit denen sich gute Ergebnisse erzielen lassen und akzeptable Laufzeiten vorweisen.

Eine einfache Herangehensweise ist die Greedy-Switch-Methode von Eades und Kelly [EK86], in der aufsteigend jeder benachbarte Knoten $v_i, v_{i+1} \in L_2$ betrachtet wird. Die Knoten werden nur dann vertauscht, wenn sich die Anzahl der Kreuzungen dadurch reduziert. Dieser Algorithmus ist im Worst-Case um einen Faktor in $O(|L_2|)$ schlechter als das Optimum und eignet sich deshalb nur als Nachbearbeitung für eine andere Heuristik.

Mäkinen und Sieranta [MS94] schlagen einen genetischen Algorithmus vor. Es werden einige Permutationen von Knoten aus L_2 als Startpopulation angenommen und diese werden Schrittweise nach ihrer Kreuzungszahl evaluiert, mutiert und kombiniert. Dieser Vorgang wird wiederholt ausgeführt, bis über 50 Iterationen keine Verbesserung erzielt wird. Die Methode liefert laut den Autoren gute Ergebnisse. Es konnte aber bisher keine Approximationsschranke festgestellt werden, zudem verfügt sie über eine deutlich höhere Laufzeit als die folgenden Heuristiken.

Bei der Median-Heuristik von Eades et al. [EW94] berechenen wir für jeden Knoten $v \in L_2$ den Wert $m(v)$. Genauer gesagt ist $m(v)$ gleich dem Median der Positionen seiner Nachbarn aus L_1. Anhand ihrer $m(v)$-Werte werden die Knoten aus L_2 umsortiert. Dieses Vorgehen bildet tatsächlich eine 3-Approximation. Es hat außerdem die Eigenschaft, dass es für Instanzen bei denen Kreuzungen komplett vermieden werden können, diese Darstellung auch immer gefunden wird.

2.2.3 Barycenter-Methode

In der vorgestellten Implementierung habe ich mich für die, von Sugiyma et al. [STT81] selbst eingeführte Barycenter-Heuristik entschieden. Die Heuristik ist sehr ähnlich zur zuvor gezeigten Median-Methode, denn auch hier ordnen wir jedem Knoten v durch eine Funktion $b(v)$ einen Wert zu. Hier entspricht $b(v)$ dem arithmetischen Mittel der Positionen p seiner Nachbarn aus L_1, und nicht mehr dem Median. Es wird also das sogenannte Barycenter, bzw. der Schwerpunkt berechnet. Die Idee dabei ist ganz einfach, wenn man Knoten nahe zu ihren Nachbarn platziert, werden wohl wenige Kreuzungen verursacht.

$$\forall v \in L_2\colon \quad b(v) = \frac{1}{|N(v)|} \sum_{u \in N(v)} p_u$$

Falls wir für einen Knoten v, den Schwerpunkt $b(v)$ bestimmen, aber ein u in der gleichen Lage existiert mit: $b(u) = b(v)$, verschieben wir $b(v)$ um ein kleines δ. Daraufhin werden die Knoten aus L_2 anhand der berechneten Werte sortiert.

Die Barycenter-Heuristik liefert eine $O(\sqrt{n})$-Approximation, was erst einmal schlechter ist als bei der Median-Heuristik. Tatsächlich schneidet die Barycenter-Methode jedoch in Tests für große Graphen von allen bekannten Heuristiken am besten ab, siehe [JM02]. Zudem hat auch sie den Vorteil, dass falls eine Sortierung ohne Kreuzungen möglich ist, diese immer gefunden wird. In Abbildung 2.4 wurde die Barycenter-Heuristik auf den Graphen aus Abbildung 2.3 angewendet. Die errechneten $b(v)$-Werte sind in den Knoten von L_2 zu sehen.

2.3 Mehrlagige Kreuzungsreduzierung

Mit dem Wissen, wie wir Kreuzungen zwischen zwei Lagen reduzieren können, wollen wir nun die Strategie für den gesamten Graphen erklären.

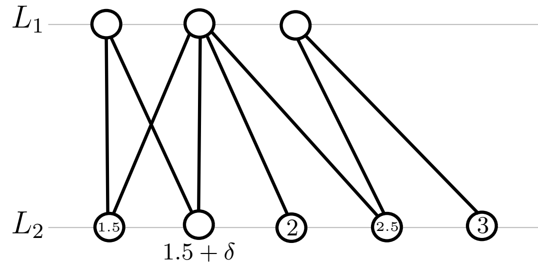

Abb. 2.4: Barycenter-Heuristik angewendet auf den Graphen aus Abbildung 2.3

Alle Lagen erhalten zu Beginn eine zufällige Sortierung der Knoten. Daraufhin werden zunächst für Knoten aus L_2 die Barycenter mit Hinblick auf die Positionen der Knoten aus L_1 ausgerechnet und dadurch neu angeordnet. Wenn die Veränderung zu einer kleineren Kreuzungszahl zwischen den beiden Lagen führt, wird die neue Sortierung übernommen. Im anderen Fall wird die alte Anordnung wiederhergestellt. Die ausgewählte Sortierung wird benutzt, um die Barycenter-Werte für L_3 zu kalkulieren und auch hier geprüft, ob es zu einer Verbesserung führt. So wird weiter vorgegangen, bis alle Lagen betrachtet wurden. Dann ist ein sogenannter *Sweep* vollständig abgeschlossen.

Wir führen nun mehrfach Sweeps vor- und rückwärts durch den Graphen durch, bis sich keine Verbesserung mehr erzielen lässt. Dabei merkt man sich das bisher beste gefundene Ergebnis. Natürlich ist das Vorgehen nicht perfekt, denn eine bestimmte Sortierung in einer vorherigen Lage beeinflusst die mögliche Anzahl der Kreuzungen auf später untersuchten Lagen. Der Algorithmus terminiert wenn periodisch die gleichen Ergebnisse, in Form von Kreuzungszahlen, auftreten. Dann wird das beste Resultat wiederhergestellt.

In der Arbeit von Sugiyama et al. [STT81] wird außerdem noch ein Nachbearbeitungsschritt vorgestellt. Bei diesem wird die gefundene Anordnung der Knoten mit den wenigsten Kreuzungen betrachtet, und ein weiteres Mal vor- und rückwärts durch den Graphen gegangen. Besonderes Augenmerk wird auf Knoten einer Lage gelegt, für die sich zunächst ein identischer $b(v)$-Wert errechnen lässt. Solche Knoten werden in ihrer Sortierung gespiegelt, im Vergleich zu vorher. Falls sich durch einen dieser Sweeps ein besseres Ergebnis erzielen lässt, wird es übernommen. Dieser Schritt wird auch in meiner Implementierung angewendet.

2.4 Brandes-Köpf-Algorithmus

Im letzten Schritt des Sugiyama-Frameworks steht das Glätten von Kanten an. Denn lange Kanten verfügen über mehrere Dummy-Knoten an unterschiedlichen Positionen. Um vor allem diese Kanten geradliniger darzustellen und die Verteilung der Knoten generell ausgewogen zu gestalten, müssen wir für alle normalen und Dummy-Knoten nun passende x-Koordinaten finden. Denn bisher wurde nur eine Sortierung der Knoten pro Lage erreicht, die daraus resultierende Zeichnung hat zwar nur noch wenige Kantenkreuzungen, aber ist noch sehr kompakt und unbalanciert.

Auch für diesen Schritt haben Sugiyama et al. [STT81] selbst einen Algorithmus vorgestellt. In einem quadratischen Programm sollen die Dummy-Knoten so angeordnet werden, dass eine gerade Linie zwischen den Start- und Zielknoten entsteht. Das entstehende Programm ist aber laufzeittechnisch sehr teuer. Deshalb gibt es auch hier wieder einige Heuristiken, wie von Gansner et al. [GKNV93], um eine gute Lösung zu approximieren. Der momentan beliebteste Algorithmus auf diesem Gebiet ist wohl von Brandes und Köpf [BK01]. Dieser wurde in meiner Implementierung ebenfalls umgesetzt. Darum wollen wir ihn im Folgenden etwas genauer besprechen.

Betrachtet man einen hierarchischen Graphen mit vertikal angeordneten Lagen, geht man einmal von oben nach unten und einmal in entgegengesetzter Richtung vor. Wir gehen im weiteren Verlauf vom ersten Fall aus. Zunächst werden alle Knoten in Blöcke sortiert. Ein Knoten v auf L_i wird dabei in den Block geordnet, in der bereits sein Median-Nachbarn m auf der vorherigen Lage L_{i-1} ist. Sollte es keinen Nachbarn geben, entsteht für diesen Knoten ein neuer Block. Auch in den beiden folgenden Fällen muss für v ein neuer Block hinzugefügt werden. Zum einen falls der ausgewählte Nachbar bereits einen anderen Knoten $w \in L_i$ zu sich in den Block geholt hat. Oder die Kante vm zum Median-Nachbarn geschnitten wird, und bereits anhand der schneidenden Kante ein Block erweitert wurde. Die genauere Fallunterscheidung hierzu will ich an dieser Stelle aber nicht ausführen. Wichtig ist, dass einem Block auf jeder Lage höchstens ein Knoten zugeordnet ist und sich Blöcke nicht mit anderen kreuzen.

Nun existieren sich über die Lagen spannende Blöcke. Pro Block erhalten alle Knoten die Koordinate ihres am weitesten oben liegenden Mitglieds. Sie werden also in vertikale Linien geformt. Als nächstes werden die Blöcke einmal soweit wie möglich am linken und einmal am rechten Rand kompakt angeordnet. Für jede dieser Positionierungen merken wir uns die x-Koordinate. Da der Algorithmus auch von unten nach oben ausgeführt wird, erhalten wir so vier verschiedene Koordinaten. Im letzten Schritt wird der Median, der vier errechneten Koordinaten, für jeden Knoten bestimmt und dieser wird für die finale Zeichnung verwendet. Ein Beispiel für alle vier Teilergebnisse, sowie die endgültige Zeichnung ist in Abbildung 2.5 zu sehen.

Insgesamt liefert der Algorithmus von Brandes und Köpf einen kompakten, aber dennoch balancierten Graphen und erhöht somit die Übersichtlichkeit sehr. Kleinere Fehler im Algorithmus in der originalen Fassung wurden von Brandes et al. [BWZ20] ausgebessert. Die Bestimmung der Medians ist in linearer Zeit möglich, und gleiches gilt für die Anordnung der Blöcke. Aufgrund der geringen Laufzeit habe ich diesen Algorithmus in meiner Anwendung umgesetzt.

2.5 Reduzierung der Dummy-Knoten

Wie bereits erwähnt, wird für Kanten, die sich über mehrere Lagen erstrecken, auf jeder Stufe zwischen Start- und Endknoten ein Dummy-Knoten eingefügt. Das führt zu einem hohen Speicherverbrauch, der den Algorithmus bei sehr großen Graphen sehr ineffizient macht. Das wollen wir durch die folgende Beobachtung ändern.

Jeder Dummy-Knoten wird durch die Barycenter-Heuristik gleichauf zu seinem vor-

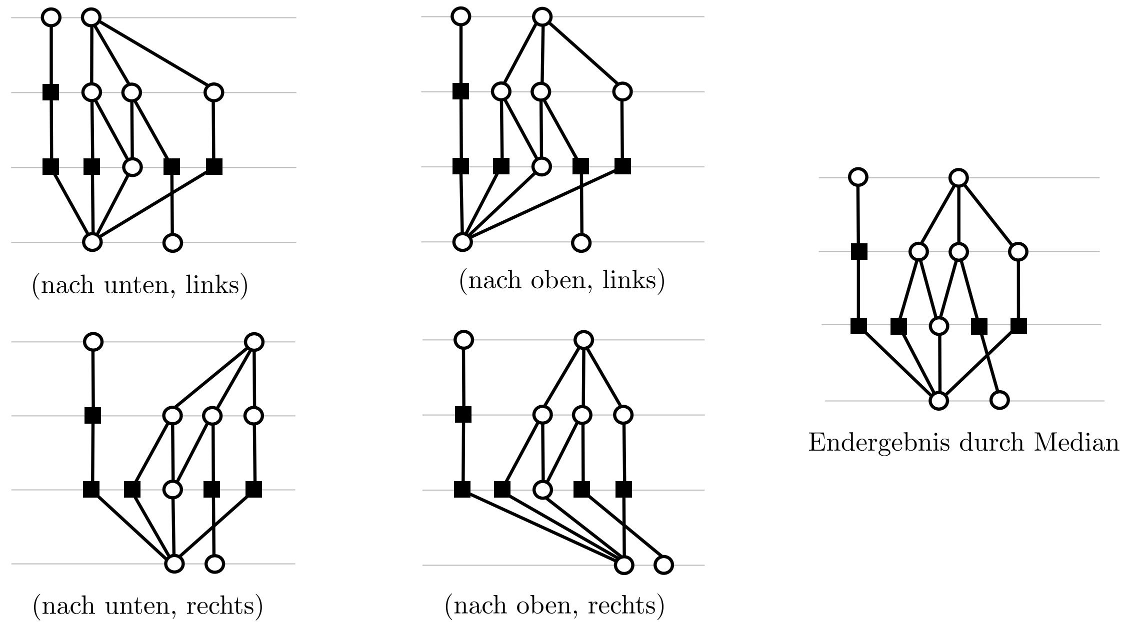

Abb. 2.5: Entstehende Zeichnungen beim Durchlauf des Brandes-Köpf-Algorithmus, Information in Klammern: (Ausführrichtung, horizontale Ordnung)

ausgegangenen Dummy-Knoten platziert, denn er hat nur ihn als Nachbarn in der vorherigen Lage. Im Brandes und Köpf Algorithmus werden Kanten zwischen Dummy-Knoten bevorzugt und immer zu Blöcken zusammengefasst. Deshalb entsteht in der klassischen Herangehensweise stets eine gerade Kante zwischen den Dummy-Knoten.

Eiglsperger et al. [ESK04] haben diese Beobachtung genutzt und das Sugiyama-Framework effizienter in $O(|E| + |V| \log |V|)$ implementiert, ohne eine Verschlechterung der entstehenden Kreuzungszahl. In ihrem Ansatz wird für sich über genau drei Lagen erstreckende Kanten ein Dummy-Knoten eingefügt, wie im klassischen Ansatz. Bei noch längeren Kanten werden aber nur zwei Dummy-Knoten ergänzt. Einen p- und einen q-Knoten, die ein sogenanntes *Segment* bilden. Das Segment wird letztendlich geradlinig gezeichnet. Der Knoten p wird auf die Lage gesetzt, die auf den Startknoten folgt, und q wird eine Stufe vor dem Endknoten platziert. Diese Idee habe ich verwendet, habe mich bei der Umsetzung jedoch für eine leichtere Variante, mit etwas höherer Laufzeit entschieden.

Bei einem Sweep durch den Graphen müssen die Segmente in einer Datenstruktur gehalten werden, die durch die momentane Lage verlaufen. So wird, wenn ein p-Knoten auf einer Lage gefunden wird, ein Segment hinzugefügt, um auch auf der nächsten Stufe die Sortierung und Kreuzungszahl korrekt zu berechnen. Damit das Segment in darauffolgenden Lagen richtig einsortiert wird, entspricht der Barycenter-Wert $b(s)$ des Segments s seiner Position in der vorherigen Lage. Sobald ein q-Knoten entdeckt wird, löschen wir das dazugehörige Segment aus der Liste heraus.

Insgesamt macht das die Implementierung um einiges komplizierter. Der Durchlauf eines vorwärts laufenden Sweeps in meiner Anwendung wird im folgenden Pseudocode

beschrieben. Für einen Rückwärtssweep ist der Code sehr ähnlich, man muss aber beachten, dass p-Knoten, dann wie q-Knoten und andersherum zu behandeln sind. Wir haben vor Beginn des Codes eine Liste aller Lagen in *layers* gespeichert. Außerdem hält jeder Knoten einen Wert *v.value*, den er von der Barycenter-Heuristik zugewiesen bekommt. Die p- und q-Knoten haben jeweils eine Referenz auf ihren eindeutigen Dummy-Nachbarn, darauf kann über die Variable *partner* zugegriffen werden. In dieser Implementierung werden die p-Knoten gleichzeitig als Segment verwendet und in der zugehörigen Liste mitgeführt. Die Funktion *sortToPattern* erhält als Parameter eine Liste mit Knoten einer Lage und die Segmente die durch diese verlaufen und gibt eine gültige Sortierung der Lage zurück.

Algorithmus 1: Durchlauf eines Vorwärtssweeps

```
1  sumCrossings = 0
2  segments = [ ]
3  for i = 2 to l do
4      patternPredeccessor = sortToPattern(layers[i − 1], segments)
5      sort layers[i] into pverts, qverts, others
                                   // others sind normale und p-Knoten
6      patternHere = sortToPattern(others, segments)
7      crossingsOld = calcCrossings(patternHere, patternPredecessor)
8      calcBarycenter(others)
9      patternHere = sortToPattern(others, segments)
10     crossingsNew = calcCrossings(patternHere, patternPredecessor)
                                   // prüfe ob neue Sortierung besser ist
11     if crossingsOld < crossingsNew then
12         resetValues(patternHere)
13         sumCrossings += crossingsOld
14     else
15         sumCrossings += crossingsNew
16     segments = (segments \ qverts.partner) ∪ pverts
17     foreach p ∈ pverts do
18         p.partner.value = p.value
                                   // speichere gegebenenfalls beste Instanz
19 if sumCrossings < crossingsLastSweep then
20     saveValues(patternHere)
```

Die Frage die sich nun stellt ist, was für eine Worst-Case Laufzeit fällt für den Schritt der Kreuzungsminimierung an. Dafür müssen wir zunächst die Laufzeit eines einzelnen Schleifendurchlaufs, des Sweeps abschätzen.

Die Anzahl der Segmente kann nicht größer sein als $|E|$. Auf einer Lage betrachten wir $|L_i|$-Knoten. Das Einsortieren von Knoten mit Segmenten ist durch einen einfachen Sortieralgorithmus machbar. Dieser benötigt somit $O((|E| + |L_i|) \log(|E| + |L_i|))$ Zeit.

Das Kalkulieren der Barycenter, sowie auch das Aufrechterhalten der Liste von Segmenten, ist in linearer Zeit machbar. Um die Anzahl der Kreuzungen auszurechnen, werden $O(|E| \log |V_i|)$ Rechenschritte gebraucht. Das Sortieren hat somit asymptotisch die schlechteste Worst-Case Laufzeit aller Berechnungsschritte eines Schleifendurchlauf. Für die Laufzeit eines gesamten Sweeps ergibt sich:

$$\sum_{i=1}^{l} O((|E| + |L_i|) \log(|E| + |L_i|))$$

Dabei kann das im Logarithmus vorkommende $|E|$ mit $|V^2|$ abgeschätzt werden. Und es gilt allgemein: $O(\log |V|^2) = O(\log |V|)$

Da zudem über alle Lagen iteriert wird und $L_1 \cup L_2 \cup \cdots \cup L_l = V$ gilt, entsteht nach Auflösen der Summe der Term:

$$O((l \cdot |E| + |V|) \log |V|)$$

Man erhält also durch diese Modifikation eine Laufzeit von $O((l \cdot |E| + |V|) \log |V|)$ für einen Sweep in der Kreuzungsminimierung. Zudem erreichen wir durch das Weglassen der Dummy-Knoten auf allen Lagen zwischen p- und q-Knoten, ein deutliches Ersparnis an erforderlichem Speicher. Statt $O(|V||E|)$ wie im klassischen Ansatz von Sugiyama, benötigt man so nur $O(|V| + |E|)$. Dadurch ist der Algorithmus auch auf sehr große Graphen anwendbar.

3 Eigene Implementierung

Da wir bereits das nötige Vorwissen für das Zeichnen von hierarchischen Graphen geklärt haben, gehe ich in diesem Kapitel genauer auf meine Implementierung ein. Umgesetzt wurde die Anwendung in der Programmiersprache Python und als Framework für die Visualisierung wurde das bekannte Modul Matplotlib verwendet. Als Input soll der in der Einleitung erwähnte Datensatz genutzt werden. Daraus wird dann ein Graph gezeichnet, der die angeführten Ziele erfüllt. Dafür wurden die zuvor erwähnten Teilschritte aus dem Sugiyama-Framework umgesetzt, um so einen horizontal angeordneten, lagenbasierten Graphen zu erhalten. Zusätzlich werden aber weitere visuelle Besonderheiten in der Zeichnung umgesetzt, die in klassischen Graphen nicht vorkommen. Für diese Merkmale sind vor allem die Kantengewichte wichtig. Diese sind hier über die Ähnlichkeiten, der literarischen Werke zueinander gegeben. Ein Kantengewicht wird über die folgende Funktion w zugeordnet.

$$w\colon E \to \mathbb{R}$$

3.1 Merkmale der Visualisierung

3.1.1 Kantendicke

Das erste besondere Merkmal ist die Dicke, bzw. Zeichenintensität der Kanten. Die Idee hier ist ganz einfach. Die Stärke der Ähnlichkeit, soll in der Visualisierung ersichtlich sein. Damit kann der Betrachter einordnen, wie sehr ein Werk oder Zeitperiode zu anderen abhängig ist. Das ist beispielsweise in Abbildung 3.1 sehr gut möglich. Hier kann man erkennen, dass der schwarze Knoten ganz links wenige Gemeinsamkeiten mit den beiden türkisen Knoten besitzt. Die jeweiligen Kanten zu diesen Knoten sind sehr schwach gezeichnet. Deutlich auffälliger ist die Verbindung zum schwarzen Knoten auf der rechten Seite.

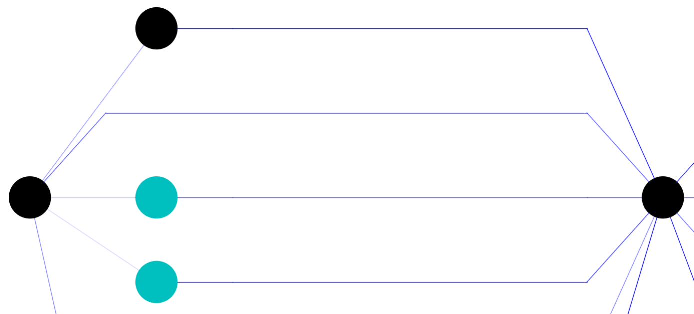

Abb. 3.1: Beispiel aus der Anwendung zur Kantendicke

Um dies im Code umzusetzen, nutzen wir den α-Wert. Dieser bestimmt für ein gezeichnetes Objekt in Matplotlib die Zeichenintensität und lässt es so auch dicker oder dünner wirken. Eine Kante wird durch ein Linien-Objekt dargestellt. Dessen α-Wert wird abhängig von der Ähnlichkeit der verbundenen Punkte gesetzt. Für eine Kante e berechnet er sich wie folgt.

$$\alpha_e = \frac{w(e)}{\max_{e' \in E} w(e')}$$

Wir teilen das Gewicht der Kante, durch das Gewicht der Kante im Graph, die am schwersten ist. Somit entspricht er einem Wert zwischen 0 und 1. Da aber für $\alpha < 0.2$ die resultierende Linie kaum mehr sichtbar ist, wird in diesem Fall $\alpha = 0.2$ gesetzt. Durch diese Minimalschranke kann der Betrachter auch weniger große Ähnlichkeiten nachverfolgen, ohne dass diese besonders auffällig sind.

Insgesamt ist dieses Merkmal für die Erfüllung unserer gesetzten Ziele sehr wichtig, denn es ist so schnell und leicht abzulesen, welche Werke in besonders hoher Abhängigkeit zueinander stehen.

3.1.2 Kontrahieren von Kanten

Eine weitere Besonderheit der Zeichnung, stellt das Kontrahieren von Kanten da. Das bedeutet generell, dass mehrere Kanten zusammengezogen werden und als eine einzige Kante abgebildet werden. In dieser Implementierung wird es folgendermaßen umgesetzt.

Betrachten wir eine Lage L_i. Die Knoten darin, können viele Kanten besitzen, die sich über mehr als nur die direkt benachbarten Lagen L_{i-1}, L_{i+1} erstrecken. Das führt in der Visualisierung dann zu vielen langen Kanten, die viel Platz einnehmen. Deshalb können in meiner Implementierung nach vorne verlaufende Kanten einer Lage kontrahiert werden, deren Endpunkte in der gleichen Lage liegen. Ebenso können eingehende Kanten zusammengezogen werden, wenn die Startpunkte der gleichen Zeitperiode zugordnet sind.

Der Ersteller der Visualisierung kann auswählen, ob er diese Möglichkeiten gemeinsam nutzen möchte, in eine der beiden Richtungen, oder gar nicht. Bei besonders großen Graphen ist das Kontrahieren sehr empfehlenswert, denn es erhöht die Übersichtlichkeit erheblich. Außerdem hat es den großen Vorteil, dass vor allem durch beidseitig kontrahierte Kanten, Abhängigkeiten zwischen unterschiedlichen Zeitspannen deutlich besser ersichtlich werden. So erfüllen wir eine weitere Zielsetzung für unsere Darstellung. Die unterschiedlichen Optionen Kanten zu Kontrahieren sind in Abbildung 3.2 veranschaulicht.

Für die Umsetzung werden bereits beim Einlesen der Daten, abhängig von den ausgewählten Einstellungen des Nutzers, die nötigen Rechenschritte vollzogen. Der erste und der letzte Dummy-Knoten, die eigentlich für eine einzige Kante gedacht sind, können jetzt mehrere Kanten zu tatsächlichen Knoten besitzen. Außerdem werden für die Kanten zwischen Dummy-Knoten, die Gewichte, der dafür zusammengezogenen Kanten, aufaddiert. Das führt zu einer proportional dickeren Kante, die dem Betrachter

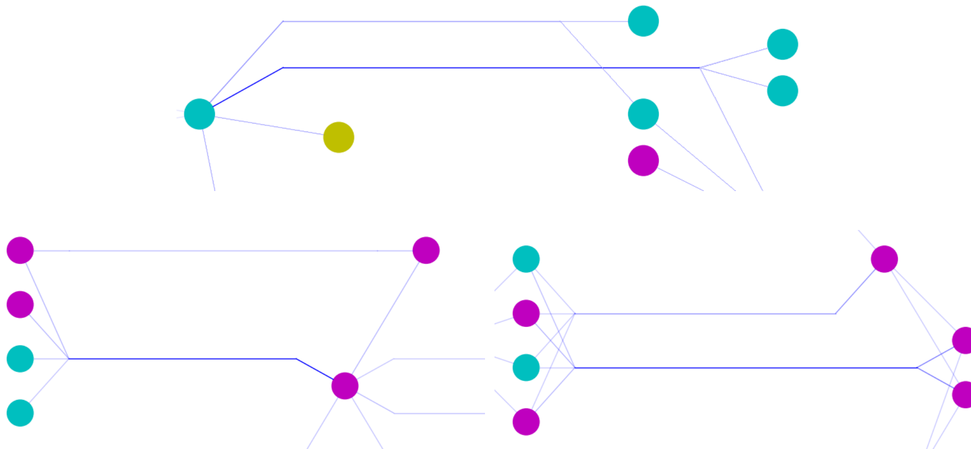

Abb. 3.2: Unterschiedliche Kontrahierungsmöglichkeiten

zeigt, dass an dieser Stelle viele Ähnlichkeiten gebündelt wurden und so eine größere Abhängigkeit zwischen den Zeitperioden andeutet.

3.1.3 Horizontale Verschiebung

Um für ein literarisches Werk zu zeigen, ob es insgesamt mehr zu älteren oder jüngeren Texten zugehörig ist, wird in der Visualisierung eine horizontale Verschiebung der Knoten vorgenommen. Dabei ist zu beachten, dass in unserer Zeichnung eine Lage nicht durch eine Linie, sondern durch ein zweidimensionales Rechteck dargestellt wird. Die vertikale Sortierung liefert die Barycenter-Heuristik und die genaue y-Koordinate wird durch den Brandes-Köpf Algorithmus ermittelt. Für die genaue x-Koordinate hingegen gehen wir von der Mitte der zugehörigen Lage aus und verschieben von dort den jeweiligen Knoten v. Für die Verschiebung werden folgende Informationen gebraucht: die durchschnittliche Ähnlichkeit $\overline{v}_{\text{in}}$ zu Texten aus älteren Lagen und die durchschnittliche Übereinstimmung $\overline{v}_{\text{out}}$ zu Werken, die in späteren Zeitperioden geschaffen wurden.

$$v_{\text{shift}} = \frac{\overline{v}_{\text{out}} - \overline{v}_{\text{in}}}{\overline{v}_{\text{out}} + \overline{v}_{\text{in}}}$$

Für die Verschiebung werwenden wir die Variable v_{shift}. Diese hat den Wertebereich $[-1, 1]$, denn der Zähler kann nie größer als der Nenner sein. Je größer der Unterschied zwischen den Mittelwerten, desto größer der Betrag von v_{shift}. Im nächsten Schritt wird v_{shift} mit der Hälfte der Breite einer Lage multipliziert und der errechnete Wert auf die x-Koordinate von v aufaddiert. Ein Knoten kann dabei nie über die Begrenzung seiner Lage hinaus verschoben werden.

In der fertigen Anwendung kann ein Threshold angelegt werden, der einen minimalen Wert beschreibt, den eine Kante haben muss, damit sie überhaupt gezeichnet wird. Deshalb werden teilweise Knoten und Kanten nicht dargestellt. Für die Berechnung der

Verschiebung werden jedoch alle Werte beachtet. Nur nicht zu Texten, die explizit vom Nutzer herausgefiltert wurden, siehe Unterabschnitt 3.2.1.

In Abbildung 3.3 ist ein Graph zu sehen, bei dem die horizontale Verschiebung vorgenommen wurde. Viele der Knoten sind eher zentral angeordnet. Jedoch gibt es auch ein paar Ausreißer, wie zum Beispiel in der Zeitperiode 1845–1850. Obwohl der türkise Knoten nur eine Kante nach vorne besitzt, ist er im hinteren Teil seiner Lage positioniert. Man kann also davon ausgehen, dass die abgebildete Ähnlichkeit die Ausnahme darstellt und das literarische Werk, hinter dem Knoten, ansonsten einen geringen Einfluss auf spätere Texte hatte.

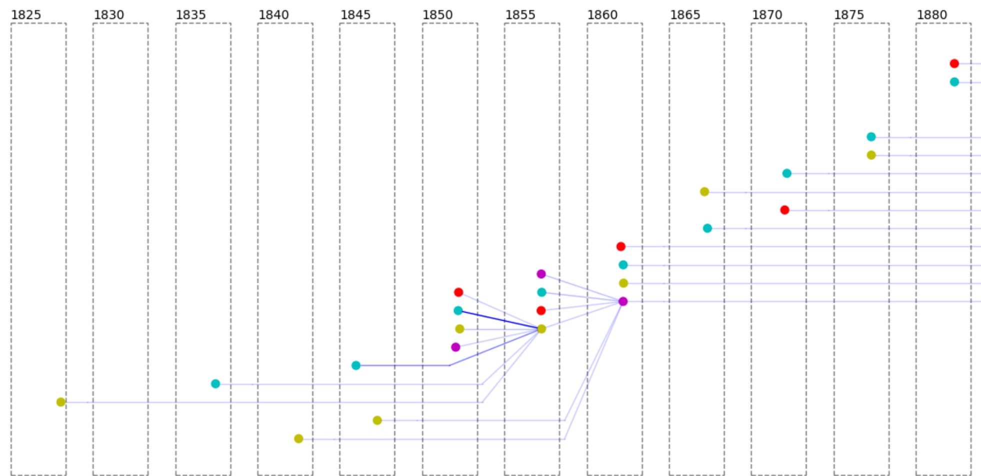

Abb. 3.3: Visualisierung mit horizontaler Verschiebung innerhalb einer Lage

Wichtig ist, dass für die Berechnung von v_{shift} Durchschnittswerte benutzt werden und keine absoluten. Das liegt daran, dass wir nur einen zeitlich begrenzten Datensatz zur Verfügung haben. Sollte man absolute Werte wählen, schieben sich offensichtlich alle alten Texte zu den neuen, denn sie haben kaum ältere Werke zu denen sie gezogen werden könnten. Gleiches gilt für neuere Texte. Deshalb ist der Durchschnitt hier passender.

3.2 Anwendung

Wie zu Beginn erwähnt, ist ein Datensatz von literarischen Werken gegeben und mehrere $n \times n$-Matrizen, die jeweils Ähnlichkeiten in Bezug auf Form, Stil, Inhalt und Emotion enthalten. All diese Informationen in einer Zeichnung verständlich unterzubringen, ist bei der Größe des Datensatzes kaum möglich. Deshalb wurden in der Anwendung zusätzlich einige verstellbare Parameter und Filtereinstellungen beigefügt. Diese ermöglichen dem Nutzer eine individuelle Zeichnung, je nachdem welche Teile des Datensatzes er sich gerade ansehen möchte. Außerdem wird eine Zeichnung so übersichtlicher und verfügt über einen klareren Ausdruck.

3.2.1 Filter und sonstige Einstellungen

Für einen möglichst benutzerfreundlichen Umgang mit der Anwendung wurde eine grafische Benutzeroberfläche angefertigt. Über diese kann der Nutzer das passende Dokument laden und die gewünschten Parameter per Point-and-Click setzen.

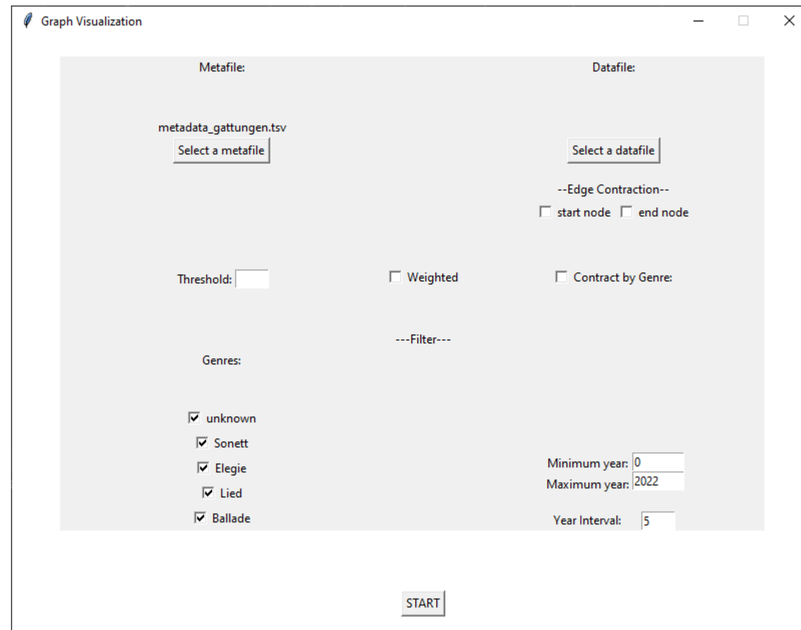

Abb. 3.4: Benutzerschnittstelle der Anwendung

Ein sehr wichtiger Parameter ist der *Threshold*. Der Nutzer kann hier einen beliebigen ganzzahligen Wert oder Gleitkommawert angeben. Dadurch werden beim Einlesen der $n \times n$-Matrix Ähnlichkeiten, die nicht größer oder gleich des Thresholds sind, aussortiert. Das führt dazu, dass die Zeichnung nicht mit Kanten überflutet wird und wir uns stattdessen auf größere Werte fokussieren.

Jedoch ist es schwierig für den Nutzer beim erstmaligen Laden einer Zeichnung direkt einen passenden Threshold zu bestimmen. Deshalb gibt es auch die Möglichkeit, das Feld leer zu lassen, woraufhin die Anwendung automatisch einen Threshold ansetzt, der in etwa so viele Kanten zulässt, wie es Knoten gibt. Der Nutzer bekommt den errechneten Wert zurück und kann sich an diesem orientieren, falls er mit dem Ergebnis noch nicht zufrieden ist.

Ein weiterer interessanter Parameter wird durch die Checkbox *Contract by genre* dargestellt. Wenn der Nutzer hier einen Haken setzt, werden nicht mehr die einzelnen Werke als Knoten angezeigt. Stattdessen werden pro Lage alle Knoten eines zusammengehörenden Genres zu einem einzigen Knoten kontrahiert. Das hilft, dem Betrachter besonders die Abhängigkeiten von Genres in verschiedenen Zeitintervallen zu zeigen, was ein weiteres Ziel der Visualisierung ist. Wobei angemerkt werden sollte, dass ein Knoten auch generell eine Färbung erhält, die auf das Genre des Texts hinweist. Damit kann auch

sonst das Genre eines Werks abgelesen werden.

Der Parameter *Weighted*, bezieht sich auf die spezielle Berechnungsmethode der Barycenter-Werte, auf die ich im folgenden Kapitel noch genauer eingehe. Außerdem gibt es einige Filtereinstellungen, um die Zeichnung beliebig zuzuschneiden. Zum einen werden die möglichen Genres nach Laden des Metafiles ausgelesen. Damit kann der Nutzer dann wählen, ob alle Genres berücksichtigt werden sollen, oder nur bestimmte. Es gibt noch einige Texte im Datensatz, die noch nicht genauer analysiert wurden. Diese sind anstatt eines richtigen Genres als *unknown* vermerkt. Somit ist es empfehlenswert vor allem diese zunächst auszusortieren. Oder man wählt nur ein einzelnes Genre aus und sieht sich speziell die Ähnlichkeiten von verwandten Texten an.

Letztlich kann noch die Größe der Jahresintervalle, standardmäßig fünf, angepasst werden, oder eine genaue Zeitperiode genauer betrachtet werden, indem ein minimales und maximales Jahr angegeben wird.

3.2.2 Beispiele

Im Folgendem werden wir uns ein paar Beispiele für Entdeckungen im Datensatz, die durch die Visualisierung gemacht werden können, ansehen.

In Abbildung 3.5 kann man sehr gut erkennen, dass beinahe alle der abgebildeten Texte eine Verbindung zu einem Lied aus der Zeitperiode 1895–1890 haben. Bei diesem auffäligen Knoten handelt es sich um das Werk *Die Reise* von *Richard Dehmel*, was durch einen Klick auf den entsprechenden Knoten auch aus der Visualisierung abzulesen ist. Dieser Text scheint besonders starke, stilistische Ähnlichkeiten zu Texten aus der Vergangenheit zu haben. Er hat aber gleichzeitig auch viel mit später erschienenen Werken gemeinsam, denn er ist horizontal sehr mittig in seiner Lage platziert.

In den Zeitperioden 1890 und 1895 sind weitere Lieder zu sehen, die ebenfalls hohe Ähnlichkeiten zu anderen Texten im Bereich Stil vorzuweisen haben. Dieses Zeitintervall und Genre scheint besonders interessant. Außerdem kann man anhand der Positionierung, der Knoten aus der Lage 1850, sehr gut erkennen, welche Werke hier eher prägend für zeitlich folgende Texte waren und welche sich mehr an dagewesenen orientiert haben.

Für die Erstellung von Abbildung 3.6 wurde nach Genres kontrahiert. Wir können somit zwar nicht mehr einzelne Werke nachverfolgen, haben dafür aber einen besseren Überblick über das große Ganze. Hier ist beispielsweise interessant zu sehen, dass Lieder aus den Jahren um 1860–1865 wohl einen besonders hohen Einfluss auf den Inhalt von nachfolgenden Werken verschiedenster Genres hatten. Dies wird durch die vielen ausgehenden Kanten an dieser Stelle deutlich und aufgrund der Tatsache, dass der zugehörige Knoten in seiner Lage weiter vorne angesiedelt ist. Die größte Ähnlichkeit besteht jedoch zu Liedern aus den Jahren 1895–1900.

Des Weiteren kann man erkennen, dass besonders die Elegien was den Inhalt betrifft sehr unter sich bleiben, denn es sind viele Kanten zwischen lila Knoten zu sehen. Dahingegen sind Sonetten insgesamt wenig vertreten und auch dann kaum Gemeinsamkeiten mit anderen Genres oder untereinander haben. Balladen sind ebenfalls etwas schwieriger zu entdecken. Bei ihnen ist aber auffällig, dass sie besonders häufig Kanten zu Elegien besitzen, wie beispielsweise, die beiden roten Knoten am unteren Bildrand.

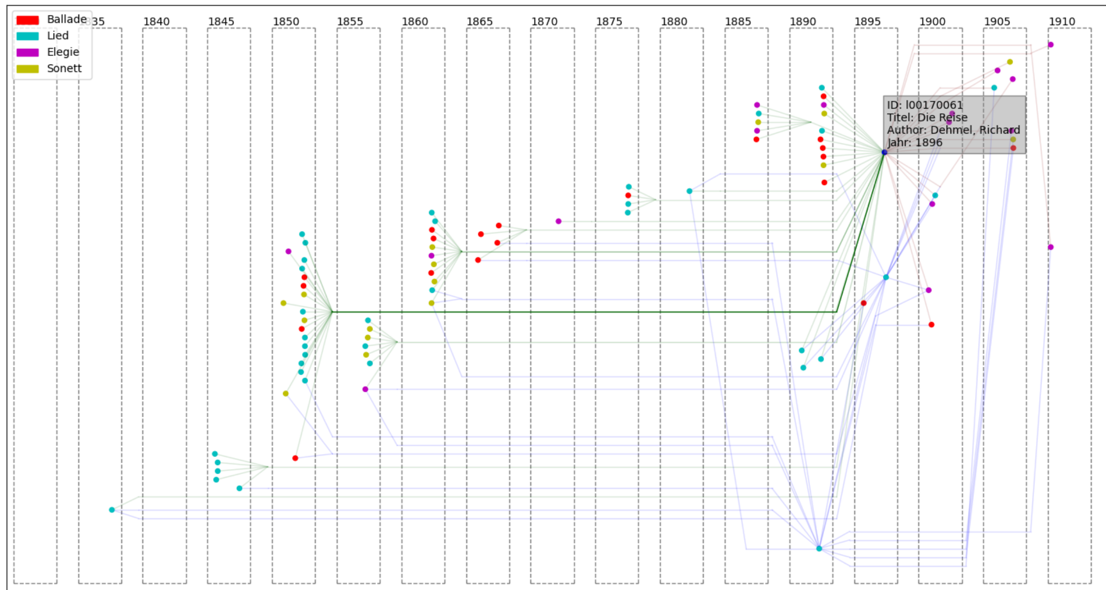

Abb. 3.5: Ähnlichkeiten Stil, mit Kantenthreshold 9

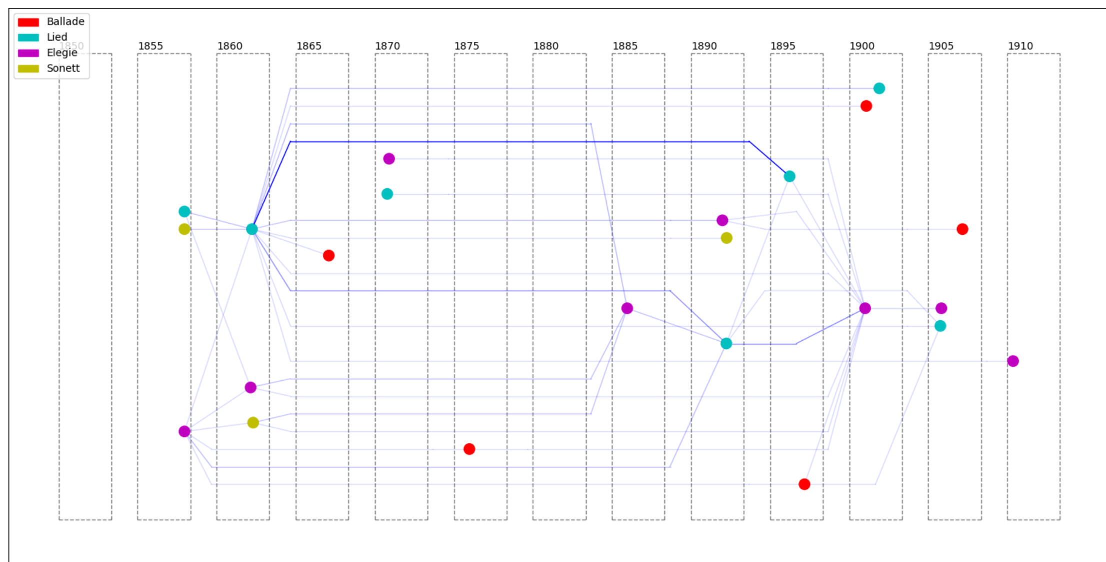

Abb. 3.6: Ähnlichkeiten Inhalt, nach Genres kontrahiert mit Threshold 5,7

4 Verbesserungsvorschläge zur Kreuzungsreduzierung

Wir haben nun die gewählte Visualisierung genauer kennengelernt und ihre Laufzeit und Speicherverbrauch analysiert. In diesem Kapitel werden weitere Ansätzen für das Problem der Kreuzungsreduzierung vorgestellt, auf die ich bei der Implementierung gestoßen bin. Diese stellen grundsätzlich Abwandlungen und Erweiterungen, der bereits kennengelernten Barycenter-Heuristik dar. Im Folgenden werde ich meine Ideen vorstellen und im letzten Schritt anhand von durchgeführten Tests zeigen, ob und wie sinnvoll die Veränderungen sind.

4.1 Gewichtete Kreuzungsreduzierung

In meiner Implementierung kommen Kantengewichte vor, mit denen Kanten unterschiedlich dick gezeichnet werden. Das hat zur Folge, dass Kreuzungen von stark gezeichneten Kanten bisher gleich behandelt werden wie Kreuzungen von kaum sichtbaren Kanten. Jedoch stechen dickere Kanten in der Visualisierung hervor und stellen meist besonders wichtige Informationen für den Betrachter dar. Deshalb wollen wir nun besonders Kreuzungen von schweren Kanten beachten, um so die Zeichnung übersichtlicher zu gestalten.

Dafür gibt es zwei Ansätze. Der erste davon ist das Beachten der Kantengewichte beim Ermitteln des Barycenters. Genauer gesagt, wird das Barycenter eines Knotens $v \in L_2$ nicht mehr als Durchschnitt der Positionen seiner Vorgänger errechnet. Sondern als Durchschnitt unter Beachtung der Kantengewichte.

$$b_w(v) = \sum_{u \in N(v)} p_u \cdot \frac{w(uv)}{\sum_{i \in N(v)} w(iv)}$$

Durch das *gewichtete Barycenter* wird ein Knoten mehr zu einem Vorgänger orientiert, zu dem er eine große Ähnlichkeit hat. Das hat den positiven Nebeneffekt, dass nun ähnliche Knoten insgesamt näher zueinander gezeichnet werden. Außerdem erfüllt es unser zuvor definiertes Ziel, indem schwere Kanten jetzt weniger schief, sondern horizontaler als vorher gezeichnet werden. Das sollte zu insgesamt weniger Kreuzungen zwischen stark gezeichneten Kanten führen.

Die zweite Idee ist es, die Kreuzungszahl anders zu berechnen. In klassischen Umsetzungen, wie auch bei Barth et al. [BJM02], zählt jede Kreuzung von Kanten, unabhängig vom Gewicht, gleich und es wird immer eine 1 pro Kreuzung aufaddiert. Die Frage, die sich stellt, ist welchen Wert summieren wir, der das Gewicht der Kanten berücksichtigt und einer Kreuzung von starken Kanten mehr Bedeutung zuweist. Für die folgende,

einfache Formel habe ich mich entschieden.

$$\text{crossValue} = w(e_1) \cdot w(e_2)$$

Die Summe aller *crossValues* bildet dann die *gewichtete Kreuzungszahl*. Offensichtlich haben dadurch Kanten mit niedrigeren Gewichten einen geringeren Einfluss auf die Gesamtsumme, als Kanten mit größerer Relevanz. Um die gewichtete Anzahl von Kreuzungen zu berechnen, gehen wir weiterhin nach dem Algorithmus von Barth et al. [BJM02] vor. Im Binärbaum wird nun aber keine 1 mehr durch den Baum nach oben addiert, sondern das entsprechende Kantengewicht. Der Wert im rechten Geschwisterknoten wird mit dem Gewicht der momentanen Kante multipliziert und auf die Kreuzungszahl aufsummiert.

Das Beispiel aus Abbildung 4.1 macht den Unterschied zum klassischen Ansatz erkenntlich. Für diese Zeichnung wurden die Kanten beidseitig kontrahiert. Das führt zu Kanten mit vergleichsweise hohen Gewichten.

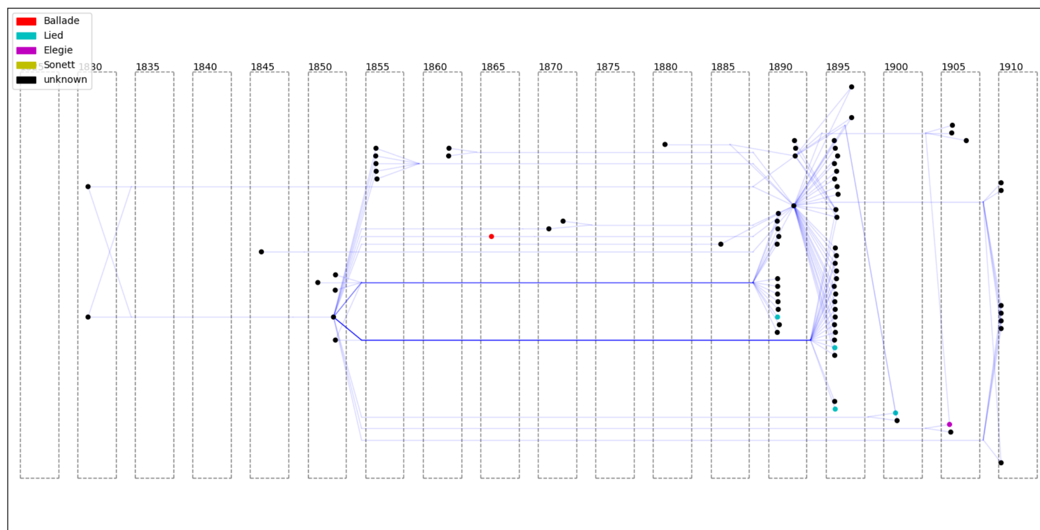

Abb. 4.1: Visualisierung mit gewichteter Kreuzungszahl

Die langen, dick gezeichneten Kanten in der Mitte der Zeichnung konnten hier kreuzungsfrei abgebildet werden. Überschneidungen mit diesen Kanten sind sehr teuer und treiben die gewichtete Kreuzungszahl schnell in die Höhe.

In Abbildung 4.2 hingegen, wurde die normale Kreuzungszahl errechnet und alle Kreuzungen somit gleich behandelt. Deshalb gibt es hier auch Kanten, die durch die stärker gezeichneten Segmente verlaufen.

4.2 Postprocessing

Häufig wird in Zuge der Kreuzungsminimierung ein Nachbearbeitungsschritt vollzogen. Nach der Umsortierung der Knoten mit einer beliebigen Heuristik, wird ein weiterer

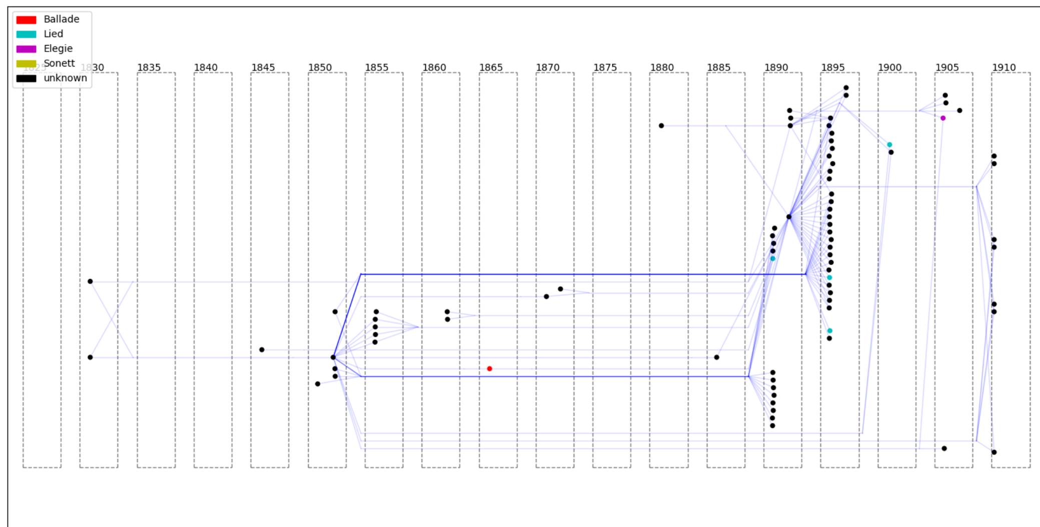

Abb. 4.2: Visualisierung ohne gewichteter Kreuzungszahl

Algorithmus ausgeführt, mit dem die Kreuzungszahl weiter verkleinert werden soll. Ein Beispiel dafür ist die Greedy-Switch-Methode von Eades und Kelly [EK86]. In meiner Umsetzung haben ich eine, nach meinem Wissen neue, Strategie implementiert, die für das Postprocessing genutzt wird.

Ein Problem, das bei Durchführung eines Sweeps im Sugiyama-Framework auftritt, ist das lange Kanten meist mehr Kreuzungen verursachen als kurze. Einfach aus dem Grund, dass sie sich über mehrere Lagen erstrecken. Somit kann der folgende Fall häufig auftreten. Bei der Durchführung eines Sweeps werden Knoten so umsortiert, dass Kreuzungen zwischen kurzen Kanten eliminiert wurden. Jedoch wurden gleichzeitig auch Dummy-Knoten von langen Kanten umgeordnet und diese veränderte Sortierung führt in den darauf folgenden Lagen zu mehr Überschneidungen als zuvor. Dadurch wird die Anzahl von Kreuzungen insgesamt so groß, dass keine der Veränderungen aus diesem Sweep übernommen wird.

Deshalb schlage ich die folgende Strategie vor. Zunächst merkt man sich die Richtung des Sweeps, bei dem, nach Ende des klassischen Ansatzes, das beste Ergebnis gefunden wurde. Man führt nun einen Durchlauf in die entgegengesetzte Richtung aus. Es wird weiterhin das Barycenter eines Knotens kalkuliert und für die Sortierung genutzt. Die Werte von p- oder q-Knoten müssen aber jetzt so manipuliert werden, dass sich die Anordnung von Segmenten untereinander nicht verändert. Im Gegensatz zu normalen Knoten, die frei einsortiert werden können.

Dabei besteht die Schwierigkeit darin, die errechneten Barycenter-Werte für p-Knoten zu verändern, dass das verbundene Segment die gleiche Sortierung, in Bezug auf andere Segmente, hat wie zuvor. Gleichzeitig sollen die veränderten Werte möglichst nah am tatsächlichen Barycenter sein, denn diese werden weiterhin genutzt, um normale Knoten einzusortieren. In einem Sweep haben wir bereits eine Liste mit den, durch die momentane Lage verlaufenden, Segmenten. Die in einer Lage L_i zu platzierenden p-Knoten

müssen sich sowohl an diesen Segmenten, aber auch an sich gegenseitig orientieren, damit die Ordnung beibehalten wird.

Für diesen Schritt werden bisherige Segmente und neue p-Knoten in die Liste *fixedOrder* eingefügt und ihrer alten Ordnung entsprechend sortiert. Es werden immer zwei, bereits platzierte Segmente als Grenzen herangezogen. Diese haben bereits feste Barycenter-Werte und zwar genau ihre Position in der vorherigen Lage. Zwischen diesen Schranken befindliche Knoten werden einmal von oben nach unten und daraufhin in die andere Richtung betrachtet. Dabei wird immer geprüft, ob $b(v)$ eines Knotens v innerhalb der Grenzen liegt und speziell in Hinblick auf seinen direkten Vorgänger passt. Sollte er eine Grenze verletzen, wird $b(v)$ um ein δ kurz vor oder nach diesen Wert verschoben. So ein Durchlauf kann abhängig, ob wir von der oberen Grenze zur unteren starten oder andersherum, zu sehr unterschiedlichen Ergebnissen führen.

Wenn beispielsweise ganz oben, in der Sortierung zwischen zwei Grenzen, ein Knoten v_i ist, der die untere Grenze nicht einhält, wird er kurz vor diese gesetzt. Die anderen Knoten, müssen sich nun ebenfalls an dieser Grenze orientieren, um die Sortierung einzuhalten. Wenn wir andersherum vorgehen, könnte es möglich sein, bei allen Knoten die $b(v)$-Werte unverändert zu lassen und nur v_i an seinem direkten Vorgänger v_{i-1} anzupassen. Deshalb wird in beide Richtungen vorgegangen und daraufhin der Mittelwert der beiden Ergebnisse gebildet. Wir speichern einmal den angepassten Barycenter-Wert in v.up und einmal in v.down ab. Die beiden Werte führen wir dann zusammen, indem wir den Durchschnitt in v.value ablegen. Das Vorgehen ist in Algorithmus 2 zu sehen.

Algorithmus 2: Anpassung der Barycenter-Werte

```
1  i = 0                                                    // Laufvariable
2  up = None                                   // obere Schranke Segment
3  upBound = ∞                                    // obere Schranke Wert
4  down = None
5  downBound = 0
6  while i < len(fixedOrder) do
                                              // finde neue obere Schranke
7      while up is None and i < len(fixedOrder) do
8          if fixedOrder[i] is Segment then
9              up = i
10             upBound = b(fixedOrder[i])
11         i++
12     predecessor = downBound
           // prüfe von unten nach oben ob p-Knoten innerhalb Schranken
       liegen
13     for j From down To up do
14         if b(fixedOrder[j]) < upBound then
15             if b(fixedOrder[j]) > predecessor then
16                 fixedOrder[j].down = b(fixedOrder[j])
17                 predecessor = b(fixedOrder[j])
18             else
19                 fixedOrder[j].down = predecessor + δ
20                 predecessor = predecessor + δ
21         else
22             fixedOrder[j].down = upBound −(δ · (up − j))
23             predecessor = upBound −(δ · (up − j))
           // hier fehlt symmetrischer Code zu Zeilen 13-23 für die
       Berechnung von v.up der Knoten, Durchlauf von oben nach unten
24     down = up
25     downBound = upBound
26     up = None
27     upBound = ∞
28 foreach p-vertex in fixedOrder do
29     p.value = (p.down + p.up) / 2
```

4.3 Test der Algorithmen

Um die zuvor vorgeschlagenen Ideen für die Kreuzungsreduzierung zu prüfen, habe ich einige Tests durchgeführt. Insgesamt wurden 15 Graphen getestet, 5 kleine Graphen mit ca. 30 Kanten, 5 mittlere mit ca. 300 Kanten und 5 große Graphen mit um die 1000 Kanten. Für jeden der Graphen wurden die Tests einmal mit einseitig kontrahierten Kanten umgesetzt und einmal ohne kontrahierte Kanten. Ein Graph wurde 100 mal getestet, mit jeweils zufälligen Startpermutationen der Knoten auf einer Lage. Geprüft wurde die Anwendung der normalen Barycenter-Heuristik, im Vergleich zur vorgeschlagenen gewichteten Barycenter-Heuristik, sowie die Verwendung der standardmäßigen Kreuzungszahl, gegenüber der gewichteten Kreuzungszahl. Pro Durchlauf wurde zusätzlich der zuvor beschriebene Postprocessing-Schritt anschließend durchgeführt, der versucht die Anzahl der Überschneidungen nochmals zu verkleinern.

So ergeben sich beispielsweise für den folgenden Graphen, der aus der gegebenen Emotions-Matrix konstruiert wurde und aus 161 Knoten und 243 Kanten besteht, die Folgenden durchschnittlichen Ergebnisse.

Mittlerer Beispielgraph Emotion								
	Ohne Kontrahieren				Mit Kontrahieren			
	Barycenter		Gew. Barycenter		Barycenter		Gew. Barycenter	
Zählweise:	Norm.	Gew.	Norm.	Gew.	Norm.	Gew.	Norm.	Gew.
Pre:								
Kreuzungen:	2593	2564	2553	2568	1165	1392	1222	1516
Gew. Kreuz.:	50375	49535	49148	49245	100326	59725	97279	64342
Post:								
Kreuzungen:	1959	1998	1895	1962	1165	1332	1209	1414
Gew. Kreuz.:	37937	38426	36227	37390	100326	55639	96195	59005

Um die Signifikanz der Tests besser einzuordnen habe ich einige Kennzahlen nach 100 Durchläufen notiert. Die durchschnittliche Anzahl an Kreuzungen, die sich mit der klassischen Herangehensweise erzielen lässt, ist bereits in der Tabelle, oben links, festgehalten und liegt bei 2593. Die unkorrigierte Stichprobenvarianz nach 100 Durchläufen wurde mit ca. 14719 bestimmt, somit ergibt sich eine Standardabweichung von ungefähr 119. Im besten Durchlauf konnte die Kreuzungszahl auf 2222 an dieser Stelle reduziert werden. Das Maximum lag bei 2962 Überschneidungen.

Schon bei diesem Graphen können einige interessante Ergebnisse abgelesen werden. Zum einen scheint der Postprocessing-Schritt hier sehr effektiv zu sein, vor allem ohne kontahierte Kanten ist die normale Kreuzungszahl deutlich geringer als die zunächst berechnete Anzahl von Überschneidungen. Des Weiteren fallen hier bei Durchläufen mit der gewichteten Barycenter-Heuristik weniger Kreuzungen an als bei Verwendung der klassischen Methode.

Wenn man sich die Resultate für kontrahierte Kanten ansieht, kann man erkennen, dass das gewichtete Zählen zu einer eindeutig niedrigeren gewichteten Kreuzungszahl

führt, auch wenn die tatsächliche Anzahl von Kreuzungen etwas größer wird. Es scheint also weniger Überschneidungen von stark gezeichneten Kanten zu geben, was die Lesbarkeit dieses Graphen fördern sollte.

Im Folgenden habe ich die Ergebnisse aus den 15 Graphen verwendet und nach Größe der Graphen gruppiert. So habe ich sie in drei Tabellen zusammengefasst, indem ich pro Graph die prozentuale Veränderung der Kreuzungszahlen, im Vergleich zur klassischen Herangehensweise errechnet habe. Daraus habe ich das arithmetische Mittel, der jeweiligen 5 Graphen gebildet und in den folgenden Tabellen festgehalten.

Ebenso wollte ich die Verbesserung, die durch das Postprocessing erzielt wird, prozentual angeben. Dafür habe ich für jeden Graph die durchschnittliche Veränderung, bei Verwendung von normalen Barycentern und Zählweise in Prozent bestimmt. Auch darüber habe ich dann pro Graphklasse den Mittelwert gebildet. Die Resultate davon sind in einer weiteren Tabelle je Graphklasse anzufinden.

Gesamte Veränderung bei kleinen Graphen

	Ohne Kontrahieren			Mit Kontrahieren		
	Barycenter	Gew. Barycenter		Barycenter	Gew. Barycenter	
Zählweise:	Gew.	Norm.	Gew.	Gew.	Norm.	Gew.
Pre:						
Kreuzungen:	6%	3%	5%	9%	-8%	5%
Gew. Kreuz.:	6%	2%	5%	-4%	-7%	-10%
Post:						
Kreuzungen:	7%	2%	6%	11%	-12%	7%
Gew. Kreuz.:	9%	3%	8%	-1%	-9%	-8%

Veränderung durch Postprocessing bei kleinen Graphen

	Ohne Kontrahieren	Mit Kontrahieren
Kreuzungen:	-7%	-7%
Gew. Kreuz.:	-8%	-8%

Gesamte Veränderung bei mittleren Graphen

	Ohne Kontrahieren			Mit Kontrahieren		
	Barycenter	Gew. Barycenter		Barycenter	Gew. Barycenter	
Zählweise:	Gew.	Norm.	Gew.	Gew.	Norm.	Gew.
Pre:						
Kreuzungen:	0%	3%	2%	22%	-1%	17%
Gew. Kreuz.:	0%	2%	1%	-19%	-8%	-21%
Post:						
Kreuzungen:	1%	2%	2%	23%	-1%	18%
Gew. Kreuz.:	1%	2%	1%	-22%	-9%	-23%

Veränderung durch Postprocessing bei mittleren Graphen		
	Ohne Kontrahieren	Mit Kontrahieren
Kreuzungen:	-8%	-3%
Gew. Kreuz.:	-8%	-3%

Gesamte Veränderung bei großen Graphen						
	Ohne Kontrahieren			Mit Kontrahieren		
	Barycenter	Gew. Barycenter		Barycenter	Gew. Barycenter	
Zählweise:	Gew.	Norm.	Gew.	Gew.	Norm.	Gew.
Pre:						
Kreuzungen:	-2%	1%	0%	13%	9%	10%
Gew. Kreuz.:	-2%	0%	0%	-16%	7%	-14%
Post:						
Kreuzungen:	-2%	0%	-1%	12%	7%	9%
Gew. Kreuz.:	-2%	0%	-1%	-15%	4%	-18%

Veränderung durch Postprocessing bei großen Graphen		
	Ohne Kontrahieren	Mit Kontrahieren
Kreuzungen:	-1%	-1%
Gew. Kreuz.:	-1%	-11%

Zu diesen zusammengefassten Ergebnissen kann die Beobachtung gemacht werden, dass ohne Verwendung von kontrahierten Kanten, die klassische Methode, mit normalen Barycenter und Zählen, im Schnitt zu den besten Resultaten führt. Die einzige Ausnahme stellt die Veränderung bei großen Graphen dar, bei welchen das gewichtete Zählen mit normaler Barycenter-Heuristik, zu insgesamt leicht besseren Werten geführt haben.

Aus den Tests mit kontrahierten Kanten kann gefolgert werden, dass die Nutzung der gewichteten Kreuzungszahl, zu deutlich weniger Kreuzungen von dicken Kanten führt, auch wenn die Anzahl an Überschneidungen im Schnitt etwas höher ist, als normalerweise. Interessant ist auch, dass sich bei Verwendung der gewichteten Kreuzungszahl, die Ergebnisse zwischen gewichteten und normalen Barycentern nicht sonderlich unterscheiden, sondern auf einem sehr ähnlichen Niveau sind. Bei kleinen und mittleren Graphen mit kontrahierten Kanten schneidet die gewichtete Barycenter-Variante gut ab, während sie bei großen Graphen, mit normaler Berechnung der Kreuzungen, schlechtere Werte erzielt.

Ähnliches kann man auch über den Postprocessing-Schritt sagen. Bei kleinen und mittleren Graphen wird die Kreuzungszahl um etwa 8% verbessert. Bei großen Graphen kommt es im Schnitt nur noch zu einer Veränderung von ca. 1%. Nur bei der gewichteten Kreuzungszahl, mit kontrahierten Kanten konnte eine Minderung von 11% erzielt werden. Insgesamt geht somit aus den Daten hervor, dass das Postprocessing besonders für kleine und mittlere Graphen effektiv ist.

5 Diskussion

Abschließend möchte ich auf die gefundenen Resultate dieser Arbeit eingehen und diese bewerten. Zunächst stellt sich die Frage nach der Effizienz des entwickelten Programms. Für die Kreuzungsreduzierung haben wir eine Laufzeit von $O((l \cdot |E| + |V|) \log |V|)$. Des Weiteren wird für den Algorithmus von Brandes und Köpf, der dem Graphen ein übersichtliches und ausgewogenes Layout verschafft, $O(|V| + |E|)$-Zeit benötigt. Das Zeichnen des Graphens ist ebenfalls in linearer Zeit möglich. Insgesamt hat die Anwendung eine Laufzeit von $O(n^2)$, denn für das Einlesen einer $n \times n$ Matrix, müssen $\frac{1}{2}n^2$ Zellen ausgelesen werden, um die verwendeten Kanten und ihre Gewichte zu bestimmen. Der Speicherverbrauch liegt, dank der Verwendung von Segmenten für lange Kanten, anstelle von einzelnen Dummy-Knoten, jedoch nur bei $O(|V| + |E|)$. Die Idee hierfür kam von Eiglsperger et al. [ESK04]. Man kann also sagen, dass der veränderte Ansatz die Effizienz des Sugiyama-Algorithmus steigert, für dieses spezielle Problem bringt er aber nur bedingt eine Verbesserung.

Außerdem stellt sich die Frage, wie gut konnten die zu Beginn der Arbeit definierten Anforderungen an unsere Visualisierung umgesetzt werden. Aufgrund von vielen verstellbaren Parametern in der Anwendung, können die Ziele besonders im einzelnen sehr gut erreicht werden. Jedoch sind sie häufig schwierig gleichzeitig in einer Zeichnung erkennbar. Um beispielsweise die Abhängigkeiten von mehreren Zeitperioden zueinander nachzuvollziehen, ist es empfehlenswert Kanten im Graphen beidseitig zu kontrahieren. Dann kann der Leser aber nicht mehr alle Kanten eines einzelnen Werks genau nachvollziehen. Die beiden Ziele stehen im Konflikt.

Leider können besonders große Graphen weiterhin sehr unübersichtlich werden. So fällt es aufgrund von vielen Kanten schwer, auf den ersten Blick Ergebnisse abzulesen. Dazu kommt, dass die horizontale Verschiebung innerhalb einer Lage, einen Nachteil mit sich bringt. Denn durch die nachträgliche Umsiedelung von Knoten, können Kanten manche Knoten überdecken, zu welchen sie eigentlich keine Verbindung haben. Um das zu verhindern, müsste das Programm erweitert werden und bei einer horizontalen Verschiebung eines Knotens geprüft werden, ob der Knoten durch diese Änderung in den Weg einer Kante gelegt wird und wie sich inzidente Kanten dann zu anderen Knoten verhalten.

Die vorgeschlagenen Änderungen bei der Kreuzungsreduzierung lieferten in den durchgeführten Tests interessante Ergebnisse. Das Postprocessing, bei dem die Sortierung von Segmenten fix gehalten und nur echte Knoten umgeordnet wurden, schnitt dort bei kleinen und mittleren Graphen sehr gut ab, jedoch wurde bei den großen Graphen durchschnittlich nur einen kleine Verbesserung erzielt.

Außerdem konnte bei Tests mit kontrahierten Kanten, durch die Verwendung von gewichtetem Zählen der Kreuzungen, die endgültige gewichtete Kreuzungszahl sehr stark

reduziert werden. Das deutet daraufhin, dass so Überschneidungen von besonders wichtigen, stark gezeichneten Kanten verringert wurden und die Übersichtlichkeit so gestiegen ist. Jedoch ist es schwer messbar, wie sinnvoll die gewählte Methode zur Berechnung dieser Zahl ist und ob sie wirklich stets zu besser verständlicheren Zeichnungen führt, als die herkömmliche Variante.

Trotz meiner Bemühungen möglichst umfangreiche und zufällige Tests durchzuführen, ist es nicht auszuschließen, dass sich die gelieferten Ergebnisse bei weiteren Prüfungen möglicherweise anders darstellen, da diese wohl besonders vom gewählten Graphen und von den zufällig generierten Startpermutationen abhängen. Dennoch bin ich von der Tendenz, der zuvor erwähnten Ergebnissen, überzeugt.

6 Fazit

In dieser Arbeit habe ich eine Anwendung implementiert, mit der eine $n \times n$-Matrix, in der ein Eintrag die Stärke der Ähnlichkeit zwischen zwei Entitäten darstellt, übersichtlich in einen hierarchischen Graphen umgewandelt werden kann. Die Laufzeit meiner Implementierung liegt in $O(n^2)$. Jedoch konnten die relevanten Teilschritte des Sugiyama-Frameworks effizienter umgesetzt werden, sodass diese nur noch $O((l \cdot |E| + |V|) \log |V|)$ Rechenschritte benötigen. Der Speicherverbrauch ist in $O(|V| + |E|)$.

Die durch die Anwendung erstellten Graphen verfügen über einige besondere Merkmale, die auf den vorliegenden Datensatz zugeschnitten sind. Somit kann durch Färbung der Knoten bildlich das Genre zugeordnet werden. Unterschiede im Gewicht einer Kante, bzw. der Ähnlichkeit zwischen zwei Dateneinheiten werden in der Zeichnung widergespiegelt. Außerdem wird für den Leser durch eine horizontale Verschiebung eines Knotens innerhalb seiner Lage deutlich, ob dieser durchschnittlich stärker verbunden zu Knoten links oder rechts von sich ist. Letztlich kann durch kontrahierte Kanten besonders gut auf die Abhängigkeit zwischen einzelnen Lagen geschlossen werden.

Abschließend habe ich veränderte Ansätze zur Reduzierung von Kantenkreuzungen vorgestellt und diese getestet. Darunter auch eine alternative Metrik für die Anzahl von Kreuzungen, bei der für eine Kreuzung das Produkt der Gewichte der beteiligten Kanten aufaddiert wird. Bei dieser gewichteten Kreuzungszahl fallen Überschneidungen von schweren Kanten stärker ins Gewicht. Tests für Graphen mit kontrahierten Kanten haben gezeigt, wenn diese Metrik direkt im Algorithmus verwendet wird, fällt die am Ende gefundene Anzahl deutlich geringer aus als es bei der standardmäßigen Zählweise der Fall wäre.

Außerdem habe ich einen zusätzlichen Schritt nach der eigentlichen Kreuzungsreduzierung eingeführt, mit der das bisherig beste Ergebnis nochmals verbessert werden kann. In diesem Postprocessing-Schritt wird die Sortierung von langen, über mehrere Lagen erstreckende Kanten, zueinander nicht mehr verändert. Stattdessen können nur echte Knoten umgeordnet werden. Aus den durchgeführten Tests geht hervor, dass dieser Schritt bei kleinen, bis mittelgroßen Graphen, die Kreuzungszahl durchschnittlich um ca. 8 % verbessert.

Literaturverzeichnis

[BJM02] W. Barth, M. Jünger und P. Mutzel: Simple and efficient bilayer cross counting. In: M.T. Goodrich und S.G. Kobourov (Herausgeber): *International Symposium on Graph Drawing*, Band 2528 der Reihe *Lecture Notes in Computer Science*, Seiten 130–141. Springer, 2002, 10.1007/3-540-36151-0$_1$3.

[BK01] U. Brandes und B. Köpf: Fast and simple horizontal coordinate assignment. In: P. Mutzel, M. Jünger und S. Leipert (Herausgeber): *International Symposium on Graph Drawing*, Band 2265 der Reihe *Lecture Notes in Computer Science*, Seiten 31–44. Springer, 2001, 10.1007/3-540-45848-4$_3$.

[BWZ20] U. Brandes, J. Walter und J. Zink: Erratum: Fast and Simple Horizontal Coordinate Assignment. *arXiv preprint arXiv:2008.01252*, 2020, 10.48550/arXiv.2008.01252.

[EK86] P. Eades und D. Kelly: Heuristic for reducing Crossing in Two-Layer Networks. *Ars. Combinatoria*, 21(A):89–98, 1986.

[ESK04] M. Eiglsperger, M. Siebenhaller und M. Kaufmann: An efficient implementation of Sugiyama's algorithm for layered graph drawing. In: J. Pach (Herausgeber): *International Symposium on Graph Drawing*, Band 3383 der Reihe *Lecture Notes in Computer Science*, Seiten 155–166. Springer, 2004, 10.1007/978-3-540-31843-9$_1$7.

[EW94] P. Eades und N.C. Wormald: Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994, doi.org/10.1007/BF01187020.

[GKNV93] E.R. Gansner, E. Koutsofios, S.C. North und K P Vo: A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993, 10.1109/32.221135.

[JM02] M. Jünger und P. Mutzel: 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. In: *Graph Algorithms And Applications I*, Seiten 3–27. World Scientific, 2002, 10.1142/9789812777638$_0$001.

[MS94] E. Mäkinen und M. Sieranta: Genetic algorithms for drawing bipartite graphs. *International Journal of Computer Mathematics*, 53(3-4):157–166, 1994, 10.1080/00207169408804322.

[NY04] H. Nagamochi und N. Yamada: Counting edge crossings in a 2-layered drawing. *Information Processing Letters*, 91(5):221–225, 2004, 10.1016/j.ipl.2004.05.001.

[San94] G. Sander: Graph layout through the VCG Tool. In: R. Tamassia und I.G. Tollis (Herausgeber): *International Symposium on Graph Drawing*, Band 894 der Reihe *Lecture Notes in Computer Science*, Seiten 194–205. Springer, 1994, 10.1007/3-540-58950-$3_3$71.

[STT81] K. Sugiyama, S. Tagawa und M. Toda: Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981, 10.1109/TSMC.1981.4308636.

[Tam13] R. Tamassia: *Handbook of graph drawing and visualization.* CRC press, 2013, 10.1201/b15385.