Bachelorarbeit

Entfernen von Knoten in Graphen

Rico Feist

Alma Mater Lipsiensis
Institut für Informatik

25. März 2014

betreuender Hochschullehrer: Prof. Gerik Scheuermann

Ich möchte mich bei Allen bedanken, die mich beim Erstellen dieser Bachelorarbeit unterstützt haben.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Problemstellung . . . 1
- 1.2 Definitionen und Konventionen . . . 1

2 Verwandte Arbeiten **5**
- 2.1 Computernetzwerke . . . 5
- 2.2 Verkehrswesen . . . 6

3 Algorithmen **7**
- 3.1 Datenquellen und Import . . . 7
  - 3.1.1 Graph . . . 7
  - 3.1.2 Funktionswerte . . . 8
- 3.2 Entfernen von Knoten . . . 8
- 3.3 Erstellen von Kanten . . . 9
  - 3.3.1 Regeln zur Kantenerstellung . . . 9
  - 3.3.2 Kantenlänge . . . 9
- 3.4 Verknüpfung der Algorithmen . . . 14
  - 3.4.1 Lokal-Lokal . . . 14
  - 3.4.2 Global-Lokal . . . 15
  - 3.4.3 Global-Global . . . 17
- 3.5 Stabilität und Determinismus der Algorithmen . . . 17
- 3.6 Zeitkomplexität . . . 20

4 Implementierung **24**
- 4.1 Ergänzungen . . . 24
  - 4.1.1 Steuerung der Algorithmen . . . 24
  - 4.1.2 2D/3D-Ansicht . . . 26
  - 4.1.3 Importer . . . 29
  - 4.1.4 Ergänzungen an anderen Klassen . . . 29
- 4.2 Offene Punkte . . . 29
  - 4.2.1 UI-Verbesserung . . . 29
  - 4.2.2 Architektur . . . 31
  - 4.2.3 Laufzeit . . . 31
  - 4.2.4 Java 8 . . . 31

5 Quellcode **33**

Abbildungsverzeichnis **61**

Quellcodeverzeichnis **62**

Literaturverzeichnis **63**

1 Einleitung

1.1 Problemstellung

Werden in einem Graphen Knoten entfernt, so müssen auch alle Kanten entfernt werden, die diesen Knoten beinhalten. Dies kann dazu führen, dass Graphen nicht mehr zusammenhängend sind oder sich die Pfadlänge zwischen zwei Knoten verlängert. Um diesen Problemen entgegen zu wirken, müssen entsprechend der Graphenstruktur neue Kanten gezogen werden. Im Rahmen dieser Bachelorarbeit wurde ein Algorithmus entwickelt, der diese Kanten nach festen Regeln erstellt und somit die Struktur eines Graphen erhält, auch wenn Knoten aus diesem entfernt werden.

1.2 Definitionen und Konventionen

Die Beispiele in dieser Arbeit basieren auf einem Gitter mit den Dimensionen 3x4. Die Knoten sind dabei wie in der Abbildung 1.1 ungerichtet verbunden, bezeichnet und farbig markiert. In den von FDLView erzeugten Bildern sind im aktuellen Schritt erzeugte Kanten stets rot, alte Kanten stets blau. Das Gitter in Abbildung 1.2 ist noch nicht bearbeitet, daher sind alle Kanten blau. Bei den Beispielen werden entweder die Knoten 6, 7 und 8 oder zusätzlich Knoten 4 entfernt. Die Abbildungen 3.1 und 3.2 haben ebenfalls einen Bezug zum Beispiel.

Im Text werden die Verhältnisse im Graphen zur besseren Lesbarkeit durch umgangssprachliche Beschreibungen ausgedrückt. Die formalen Definitionen seien hier kurz aufgeführt, sie folgen (Bodendiek und Lang, 1995).

Nachbar Sei $G = (V, E) \neq \emptyset$ ein Graph, dann ist $n \in V$ ein Nachbar von $k \in V$, wenn gilt: $(n, k) \in E$.

Pfad Sei $P(u, v)$ der Pfad in einem Graph $G = (V, E) \neq \emptyset$, der die Knoten u und v miteinander verbindet. Dann ist $P(u, v)$ eine alternierende Folge $v_1, e_1, v_2, e_2, ..., v_n, e_n, v_{n+1}$ mit $v_i \in V, e_i \in E, n \geq 1$ für die gilt

1. $v_1 \neq v_{n+1}$
2. Für je zwei $i, j \in \{1, 2, ..., n\}, i \neq j$ gilt $v_i \neq v_j$
3. Für jedes $i \in \{1, 2, ..., n\}$ sind v_i und v_{i+1} die Knoten der zwischen ihnen stehenden Kante e_i.

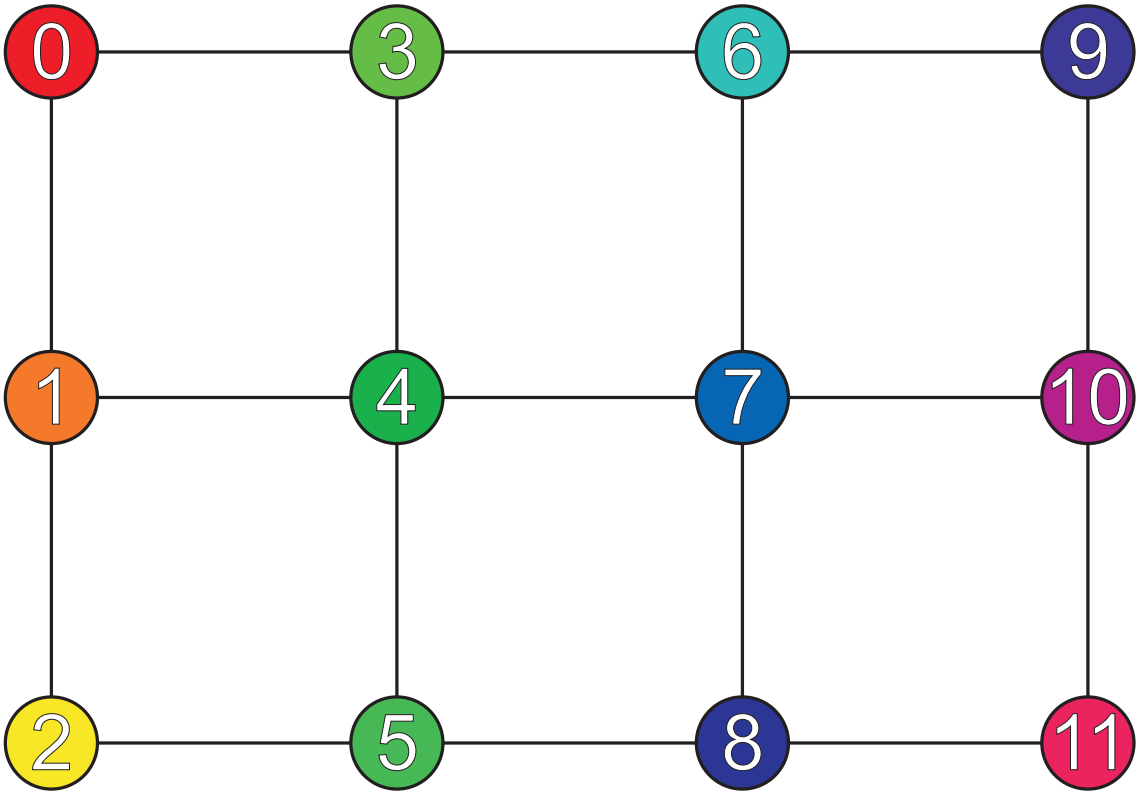


Abbildung 1.1: Knotenfarben und -positionen des in der Arbeit verwendeten, einfachen Beispiels.

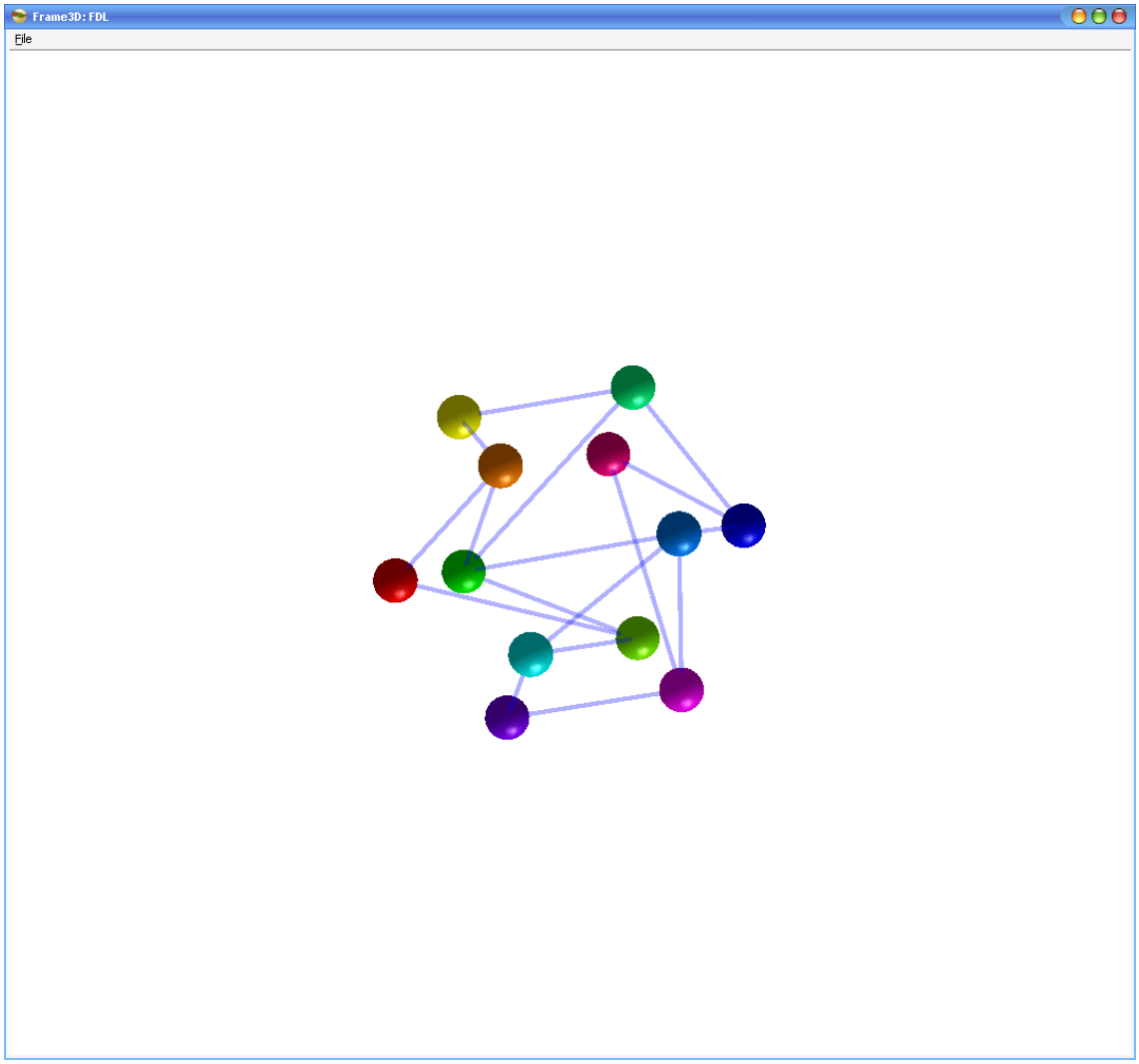


Abbildung 1.2: Das gleiche Beispiel in FDLView.

Entfernung zwischen zwei Knoten Sei $P(u, v)$ ein kürzester Pfad in einem Graph $G = (V, E) \neq \emptyset$, der die Knoten u und v miteinander verbindet. Dann ist die Entfernung zwischen den Knoten u und v die Länge dieses Pfades. Entfernung bezieht sich immer auf die Entfernung im graphentheoretischen Sinne.

2 Verwandte Arbeiten

Das Problem, in einem Graph neue Kanten zu ziehen, taucht in verschiedenen Disziplinen auf.

2.1 Computernetzwerke

Fällt bei einem Netzwerk ein Knoten aus, so ist die Aufgabe zunächst nicht, eine neue Verbindung zu erstellen, sondern zunächst die wegfallenden Route zu ersetzen. Während große Netzwerke jedoch tatsächlich ungerichtete Graphen sind, so sind mittlere und kleine Netzwerke nicht nur Graphen, sondern Bäume. Der Grund dafür sind die eingesetzten Routing-Algorithmen, die dafür sorgen, dass in einem unmanaged Netzwerk keine Zyklen entstehen dürfen. Da Netzwerkkabel grundsätzlich bi-direktional arbeiten, darf in diesen Fällen zwischen zwei beliebigen Knoten nur genau ein Pfad existieren. Grund dafür ist die Art, wie einfache Router die Verbindungen verwalten. Genaue Erläuterung sind in (Comers, 2003) nachzulesen. Zu jeder IP existiert dabei in der Routingtabelle genau ein Eintrag. Gibt es nun aber einen Zyklus im Netzwerk, so kann ein Paket zwischen zwei Knoten zwei Wege nehmen: Einmal mit und einmal ohne Durchlauf des Zyklus. Dadurch würde es jedoch zu einer Flutung des Netzwerkes kommen. Es ist daher ersichtlich, dass der Ausfall eines Knotens in einem solchen Netzwerk zu einem Teilausfall führt. Da es keine zweite Route gibt, über die das Netzwerk geführt werden könnte, ist das Problem auf diese Art Netzwerke nicht anwendbar.

In größeren Netzwerken gibt es hingegen aus Redundanzgründen genau den Fall, dass wichtige oder alle Leitungen mehrfach ausgeführt sind. Wie bereits beschrieben müssen dann jedoch die Routen manuell gesetzt werden, denn das automatische Routing funktioniert in diesem Fall nicht mehr. Während also in der in dieser Arbeit bearbeiteten Aufgabenstellung neue Kanten gezogen werden, wenn sie benötigt werden, müssen diese im Falle eines Netzwerkes bereits vorhanden sein, wenn ein Knoten ausfällt, ergo entfernt wird. Die Planung, welche Verbindungen erstellt werden, geschieht jedoch nicht automatisch. In einem Computernetzwerk müssen neben der reinen Erreichbarkeit (Gibt es einen Weg?) auch noch die Laufzeit (Wie lange benötigt ein Paket?) und der Durchsatz (Wie viele Pakete können pro Sekunde über eine Leitung versendet werden?) beachtet werden. Aufgrund dieser Daten sind die neu zu erstellenden Routen bereits im Voraus bekannt und entsprechend in den Netzwerkkomponenten eingestellt. Siehe dazu (Wright, 1999).

2.2 Verkehrswesen

Ein Straßennetz kann als einfacher, gerichteter Graph modelliert werden. Während beispielsweise Einbahnstraßen im Graph abgebildet werden, können andere Einschränkungen wie Abbiegeverbote nicht abgebildet werden. Wie bei Netzwerken auch werden jedoch bei geplanten Entfernungen die entsprechenden Umleitungen bereits im Vorfeld angelegt oder ausgeschildert. Dennoch finden Algorithmen zur Routenfindung auch in diesem Bereich Anwendung. Unvorhergesehene Störungen, wie beispielsweise eine aufgrund eines Unfalls gesperrte Kreuzung zwingen die anderen Verkehrsteilnehmer, eine neue Route zu wählen. Diese soll verständlicherweise nicht länger sein als die zuvor gewählte. In diesem Fall werden jedoch nur neue Routen auf dem bestehenden Graphen gesucht. Bei geplanten Sperrungen werden jedoch auch neue Kanten gezogen. Während diese im Graph anhand der Kürze der Route gewählt werden können, müssen bei für Umleitungsstrecken zu bauenden Straßen jedoch auch auf die örtlichen Gegebenheiten geachtet werden. Nichtsdestoweniger stehen Verkehrsplaner hier vor einer sehr ähnlichen Aufgabe. Auf die Modellierung von Straßennetzen geht (Foulds, 1992) genauer ein und modelliert diese mit gerichteten Graphen. Im Buch wird vorgestellt, dass sich das Problem der Erreichbarkeit in Straßennetzen zurückführen lässt auf die Problemstellung, ob ein Graph stark zusammenhängend ist. In dieser Arbeit wird auch auf Erreichbarkeit geprüft, jedoch muss dabei zusätzlich der Abstand zwischen den Knoten betrachtet werden. Des Weiteren geht der Autor auf das Braess-Paradoxon ein und zeigt, wie sich durch das Entfernen von Straßen die Kapazität in einem Straßennetz steigern lässt.

3 Algorithmen

3.1 Datenquellen und Import

Die Daten für Graph und Funktionswerte stehen als CSV-Dateien zur Verfügung.

3.1.1 Graph

Ein Graph wird durch eine CSV-Datei beschrieben, welche aus so vielen Zeilen und Spalten besteht, wie der Graph Knoten besitzt. Jede der Zellen beschreibt eine Kante, wobei die Zeile den Startpunkt und die Spalte den Endpunkt einer Kante bestimmt. Der Wert in der Zelle gibt die Länge der Kante an. Da es theoretisch auch Kanten der Länge 0 geben kann, wird der Wert `Inf` verwendet, wenn zwischen zwei Knoten keine Kante in dieser Richtung existiert. Die Datei, aus der das in dieser Arbeit verwendete Beispiel erzeugt wurde, ist in Listing 3.1 abgedruckt. Die Algorithmen können auf beliebigen gerichteten und ungerichteten Graphen arbeiten, produktiv verarbeitet werden jedoch nur zwei- und dreidimensionale, regelmäßige, ungerichtete Gitter.

Knoten und Kanten erhalten beim Import per CSV keine bestimmte Bezeichnung. Die Knoten werden beginnend bei 0 durchnummeriert. Kanten werden entsprechend ihrer Start- und Endknoten, getrennt durch einen Bindestrich, bezeichnet. Dies ermöglicht eine schnelle Zuordnung der Kanten.

Die Farbe der Knoten wird durch das HSV-Modell bestimmt. Die Nummer des Knotens wird dabei auf den `Hue`-Anteil nach der Formel

$$\text{Hue} = 360 * (\frac{\text{Knotennummer}}{\text{Anzahl Knoten}}) \tag{3.1}$$

abgebildet. Auch die Knoten des Beispiels, siehe Graphik 1.1, sind auf diese Art gefärbt.

Beim Import werden Knoten und Kanten weitere Werte als Standard übergeben. Dies betrifft alle Werte, die für Layout und Darstellung benötigt werden sowie weitere Beschreibungen des Knotens. In den Abbildungen 3.1 und 3.2 sind die Knoten- respektive Kantenliste der Beispieldatei dargestellt.

Listing 3.1: CSV-Datei, aus der das in dieser Arbeit verwendete Beispiel erzeugt wird.

```
1  Inf;1;Inf;1;Inf;Inf;Inf;Inf;Inf;Inf;Inf;Inf
2  1;Inf;1;Inf;1;Inf;Inf;Inf;Inf;Inf;Inf;Inf
3  Inf;1;Inf;Inf;Inf;1;Inf;Inf;Inf;Inf;Inf;Inf
4  1;Inf;Inf;Inf;1;Inf;1;Inf;Inf;Inf;Inf;Inf
5  Inf;1;Inf;1;Inf;1;Inf;1;Inf;Inf;Inf;Inf
6  Inf;Inf;1;Inf;1;Inf;Inf;Inf;1;Inf;Inf;Inf
7  Inf;Inf;Inf;1;Inf;Inf;Inf;1;Inf;1;Inf;Inf
8  Inf;Inf;Inf;Inf;1;Inf;1;Inf;1;Inf;1;Inf
9  Inf;Inf;Inf;Inf;Inf;1;Inf;1;Inf;Inf;Inf;1
10 Inf;Inf;Inf;Inf;Inf;Inf;1;Inf;Inf;Inf;1;Inf
11 Inf;Inf;Inf;Inf;Inf;Inf;Inf;1;Inf;1;Inf;1
12 Inf;Inf;Inf;Inf;Inf;Inf;Inf;Inf;1;Inf;1;Inf
```

3.1.2 Funktionswerte

Die Funktionswerte in einem Graphen werden durch eine CSV-Datei beschrieben, welche aus einer Zeile mit so vielen Werten besteht, wie der Graph Knoten besitzt. Diese Werte werden prozentual auf die Saturation der Knoten abgebildet nach der Formel

$$\text{Saturation} = \frac{\text{Funktionswerte}_{\text{Knoten}}}{\text{Funktionswerte}_{\text{Maximum}}} \tag{3.2}$$

Abbildung 3.3 ist aus Listing 3.2 erstellt. In dieser Abbildung ist zu sehen, dass die Knoten 0 und 8 hoch saturiert sind, während die Knoten 4,6,7,8 komplett weiß sind.

Listing 3.2: Beispielhafte Datei mit Funktionswerten.

```
1  10;5;1;2;0;1;0;0;0;10;1;3
```

3.2 Entfernen von Knoten

Gemäß der Problemstellung sollen Knoten entfernt werden, wenn ihre Funktionswerte 0 sind. Wird ein Knoten entfernt, so müssen auch alle zugehörigen Kanten entfernt werden. In Abbildung 3.4 wurde Knoten 7 entfernt und folglich auch die Kanten 4-7, 6-7, 8-7 und 10-7.

ID	Charge	ClassName	OriginName
0	2.0	0	root
1	2.0	1	root
2	2.0	2	root
3	2.0	3	root
4	2.0	4	root
5	2.0	5	root
6	2.0	6	root
7	2.0	7	root
8	2.0	8	root
9	2.0	9	root
10	2.0	10	root
11	2.0	11	root

Abbildung 3.1: Knotenliste des Beispiels in FDLView

3.3 Erstellen von Kanten

3.3.1 Regeln zur Kantenerstellung

Eine neue Kante zwischen zwei Knoten soll nur dann erstellt werden, wenn folgende Bedingungen gelten:

1. Die Knoten sind Nachbarn eines entfernten Knoten.
2. Die Länge des Pfades zwischen den Knoten ist größer als vor dem Entfernen.

3.3.2 Kantenlänge

Die Länge der zu erzeugenden Kante muss aus den verbleibenden Kanten errechnet werden. Basis ist der kürzeste Pfad zwischen den Knoten, dessen Länge auch Basis der Algorithmen ist. Dieser wird über den Dijkstra-Algorithmus berechnet. Beschrieben wird die Erstellung für Pfade beliebiger Länge. Die Berechnung für Pfade der Länge 2 ist, obgleich sie am häufigsten auftritt, ein Spezialfall davon:

1. Nimm die letzte und die vorletzte Kante des Pfades.
2. Berechne die Länge der neuen Kanten aus den beiden gewählten Kanten.
3. Entferne die letzte und die vorletzte Kante des Pfades und füge die neue Kante am Ende des Pfades hinzu.
4. Wende den Algorithmus auf den neu entstandenen Pfad an, bis der Pfad nur noch aus einer Kante besteht.

Die Implementierung dieses Algorithmus ist im Listing 5.1 nachzulesen.

ID	Desired len...	Stiffness	PositiveInter...	NegativeInt...	Ratio 1
0-1	1.000	0.010			
0-3	1.000	0.010			
1-0	1.000	0.010			
1-2	1.000	0.010			
1-4	1.000	0.010			
2-1	1.000	0.010			
2-5	1.000	0.010			
3-0	1.000	0.010			
3-4	1.000	0.010			
3-6	1.000	0.010			
4-1	1.000	0.010			
4-3	1.000	0.010			
4-5	1.000	0.010			
4-7	1.000	0.010			
5-2	1.000	0.010			
5-4	1.000	0.010			
5-8	1.000	0.010			
6-3	1.000	0.010			
6-7	1.000	0.010			
6-9	1.000	0.010			
7-4	1.000	0.010			
7-6	1.000	0.010			
7-8	1.000	0.010			
7-10	1.000	0.010			
8-5	1.000	0.010			
8-7	1.000	0.010			
8-11	1.000	0.010			
9-6	1.000	0.010			
9-10	1.000	0.010			
10-7	1.000	0.010			
10-9	1.000	0.010			
10-11	1.000	0.010			
11-8	1.000	0.010			
11-10	1.000	0.010			

Abbildung 3.2: Kantenliste des Beispiels in FDLView

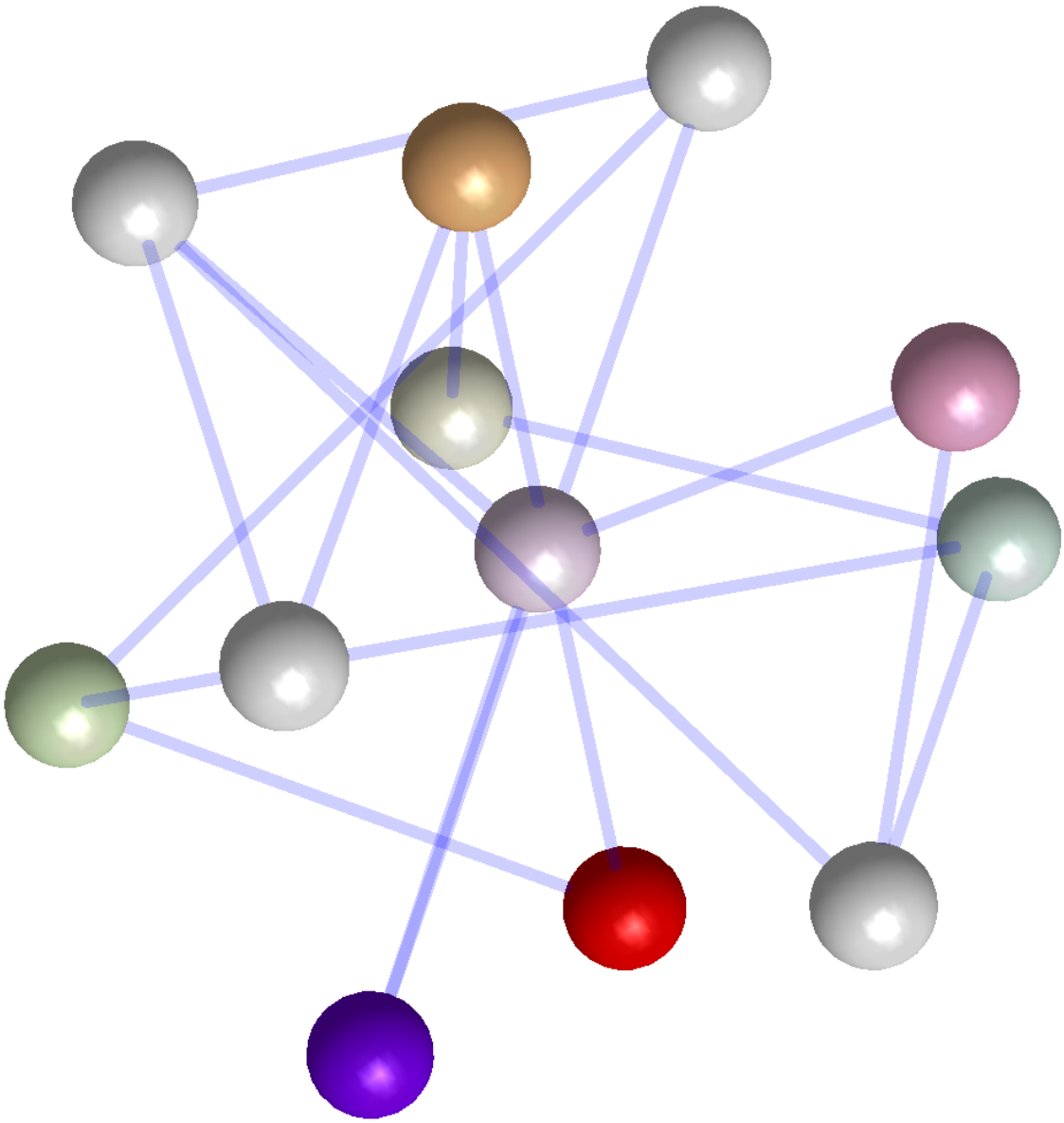


Abbildung 3.3: Aus Listing 3.2 erzeugter Graph mit importierten Funktionswerten.

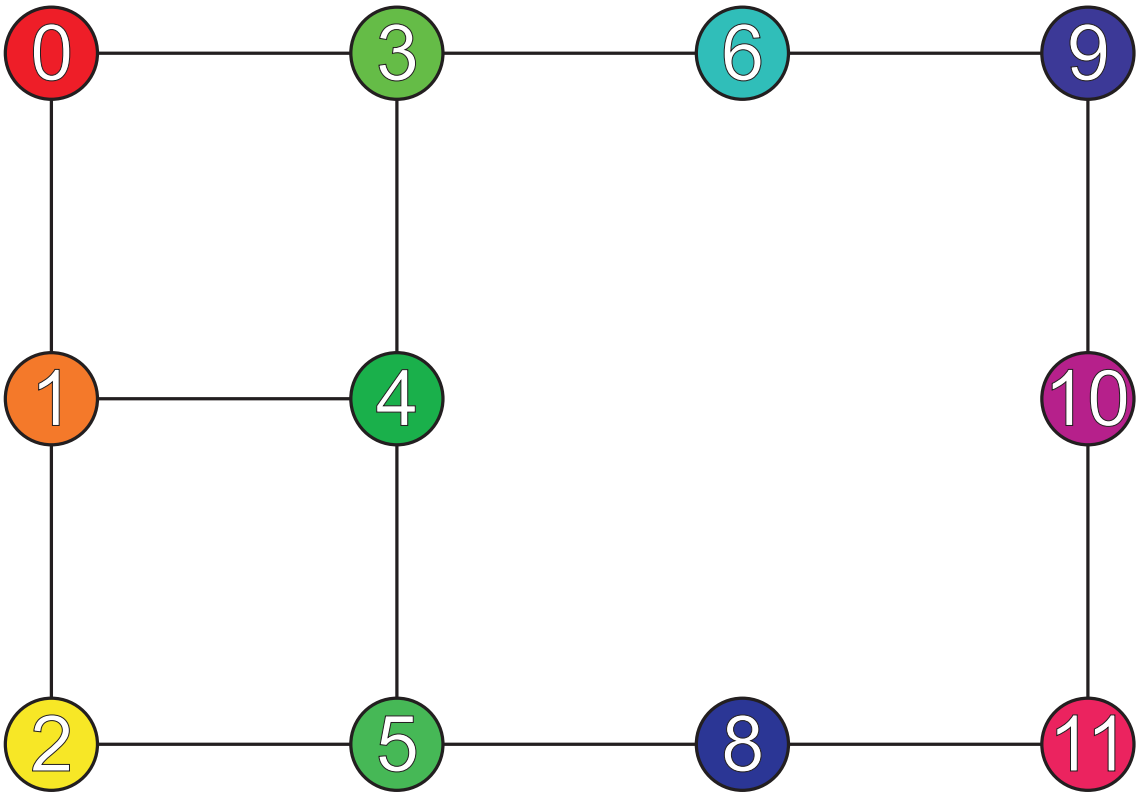


Abbildung 3.4: Im Beispielgitter wurde Knoten 7 entfernt.

Die Berechnung der Länge der Kante wird auf die Berechnung einer Kante im Dreieck mit dem Kosinussatz zurückgeführt. Hier ist der Fall `SWS` gegeben, das heißt, zwei Seiten und der von ihnen eingeschlossene Winkel sind gegeben. Die Umsetzung in Quellcode ist im Listing 5.2 abgedruckt, die Formel dazu lautet

$$a = \sqrt{b^2 + c^2 - 2bc\cos(\alpha)} \tag{3.3}$$

Die beiden Seiten b und c des Dreiecks sind bereits bekannt, der Winkel α zwischen diesen Kanten jedoch nicht und muss daher über die Position der Eckpunkte dieser Kanten bestimmt werden. Da es keine dedizierte Methode zur Berechnung des Schnittwinkels zweier Strecken gibt, muss diese Berechnung als Spezialfall der Berechnung für Geraden ausgeführt werden. Es ist zunächst bekannt, dass die beiden Strecken einen gemeinsamen Punkt haben, an dem der Winkel zu berechnen ist. Aus den beiden Punkten der Kante lässt sich zudem die Geradengleichung in Parameterform errechnen, die als Basis für die Winkelberechnung dient. Diese Berechnung erfolgt mit der Formel

$$\alpha = \arccos\left(\frac{\vec{b} \cdot \vec{c}}{|\vec{b}| \cdot |\vec{c}|}\right) \tag{3.4}$$

Die Winkelberechnung erfolgt in der Methode `calculateAngle`, siehe dazu Listing 5.3.

Problematisch ist, dass der Algorithmus deterministisch sein muss. Während es beim Schnitt zweier Geraden stets einen stumpfen und einen spitzen Winkel gibt, existieren für den Spezialfall hier ein überstumpfer Winkel und ein nicht überstumpfer Winkel, wobei der nicht überstumpfe Winkel für die weitere Berechnung relevant ist. Da jedoch die Schnittwinkelberechnung bei Geraden deterministisch ist, ist auch die Berechnung für Strecken deterministisch. Es wird dabei festgelegt, dass für die Geradengleichung der gemeinsame Punkt beider Geraden der Stützpunkt für die Parameterform der Gerade ist. Die weitere Berechnung erfolgt wie in (Rudolph, 2013) vorgestellt. Die Korrektheit des Algorithmus wurde durch Unittests sichergestellt, siehe Listing 5.4.

3.4 Verknüpfung der Algorithmen

Der Algorithmus zum Entfernen von Knoten und der Algorithmus zum Erstellen von Kanten lassen sich unterschiedlich miteinander verknüpfen. Unterscheidungsmerkmal ist dabei der Betrachtungsraum, also in welchem Umfang Nachbarn und entfernte Knoten betrachtet werden. Dabei wird unterschieden zwischen

Lokal In einem Schritt wird nur der in diesem Moment entfernte Knoten bzw. die erstellte Kante betrachtet.

Global Es werden alle entfernten Knoten und erstellten Kanten betrachtet.

Daraus lassen sich drei verschieden Arten der Verknüpfung ableiten, die im einzelnen vorgestellt werden sollen.

Lokal-Lokal Wird ein Knoten entfernt, werden alle zugehörigen Kanten erstellt. Danach wird der nächste Knoten entfernt.

Global-Lokal Die Knoten werden in einer festen Reihenfolge entfernt, aber nach jedem entfernten Knoten werden die zugehörigen Kanten erstellt.

Global-Global Es werden erst alle Knoten entfernt und anschließend die neuen Kanten erstellt.

Die Verknüpfung **Lokal-Global** ist nicht möglich, da bei dieser alle neuen Kanten erstellt werden müssten, ohne alle zu entfernenden Knoten zu kennen.

3.4.1 Lokal-Lokal

Bei der Lokal-Lokalen Verknüpfung wird ein Knoten entfernt, dann alle zugehörigen Kanten erstellt und danach der nächste Knoten entfernt. Der Code in Listing 5.5 wird aufgerufen, sobald ein Knoten entfernt wird.

Der Algorithmus liest zunächst die Nachbarn des markierten Knotens aus und berechnet für alle Knoten die jeweiligen Entfernungen untereinander. Anschließend wird der Knoten entfernt und die Entfernungen zwischen allen Knoten erneut berechnet. Im nächsten Schritt wird über alle Kanten iteriert, von denen mindestens ein Knoten Nachbar des entfernten Knoten war und entsprechend die Kanten erzeugt.

Dies ist der einfachste der implementierten Algorithmen, zudem wird dieser Algorithmus als Unterfunktion der Global-Lokalen Verknüpfung aufgerufen. Die Reihenfolge, in der die Knoten entfernt werden, wird bei diesem Algorithmus allein vom Nutzer des Programmes bestimmt.

3.4.2 Global-Lokal

Bei der Global-Lokalen Verknüpfung werden die Knoten in einer festgelegten Reihenfolge entfernt, aber die Kanten nach jedem Entfernen neu erstellt.

Hierbei gibt es zwei Möglichkeiten, wann die neuen Kanten berechnet und erstellt werden:

1. Nach jedem entfernten Knoten
2. Nachdem alle Knoten eines Grades entfernt wurden

Nach jedem entfernten Knoten

Diese Variante ist momentan im Programm implementiert. Die Reihenfolge der Knoten wird dabei durch den Grad des Knotens bestimmt, sowie bei Gleichheit des Grades durch die ID des Knotens. Letzteres führt zu einem Problem, denn der Algorithmus ist in dieser Art nicht stabil. Sollen beispielsweise die Knoten wie im Beispiel 3.2, also 4, 6, 7 und 8 entfernt werden, so ist die Reihenfolge in dieser Implementierung 6, 8, 4, 7. Da sowohl Knoten 6 als auch Knoten 8 den Grad 3 haben, wird die Reihenfolge dieser Knoten ausschließlich durch ihre ID bestimmt - diese wird aber nur durch die Reihenfolge beim Import festgelegt. Da der Algorithmus ansonsten stabil ist, muss lediglich dieses Problem gelöst werden. Die Lösung dazu ist, die Kantenberechnung erst durchzuführen, wenn alle Knoten eines Grades entfernt wurden.

Nachdem alle Knoten eines Grades entfernt wurden

Werden die Kanten nicht nach jedem Knoten neu erstellt, sondern erst dann, wenn alle Knoten eines Grades entfernt wurden, wird der Algorithmus dadurch nicht nur stabil, sondern es ist auch ein Schritt zur Global-Globalen Verknüpfung. Dies soll ein Beispiel verdeutlichen.

Werden die Knoten 3 und 6 entfernt, so haben beide Knoten den Grad 3, werden also in der Reihenfolge ihrer ID entfernt. Schaut man nun auf die erzeugten Kanten, so werden folgende Kanten erzeugt:

Erst 3, dann 6 entfernt 0-7 und 0-9, siehe Abbildung 3.5

Erst 6, dann 3 entfernt 0-9 und 4-9, siehe Abbildung 3.6

Vergleichend dazu wird nur die Kante 0-9 erzeugt, wenn die Kanten erst berechnet werden, wenn alle Knoten eines Grades entfernt wurden. Der dadurch erzeugte Graph ist in Abbildung 3.7 zu sehen. Diese Veränderung sorgt dafür, dass der Algorithmus stabil ist. Zudem werden in der Regel weniger Kanten erzeugt.

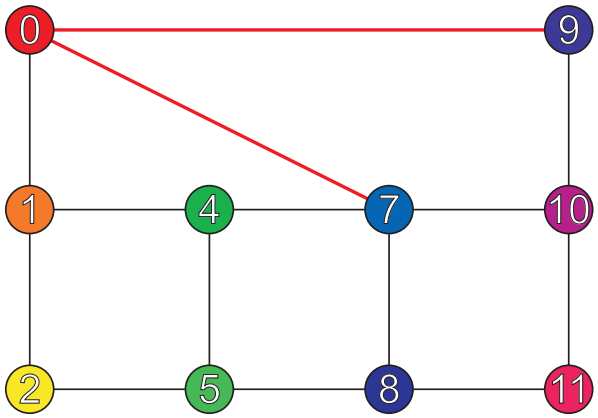


Abbildung 3.5: Graph, wenn bei Global-Lokal erst Knoten 3 und dann Knoten 6 entfernt werden.

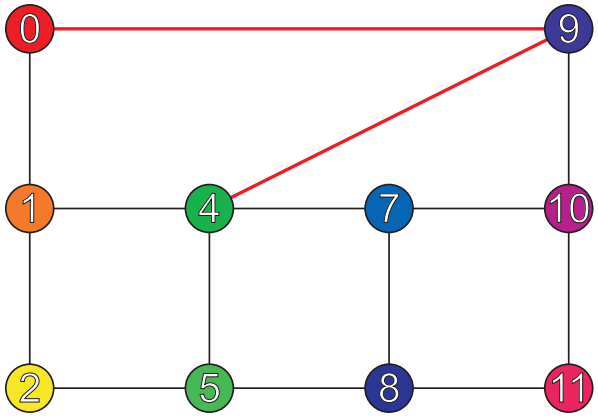


Abbildung 3.6: Graph, wenn bei Global-Lokal erst Knoten 6 und dann Knoten 3 entfernt werden.

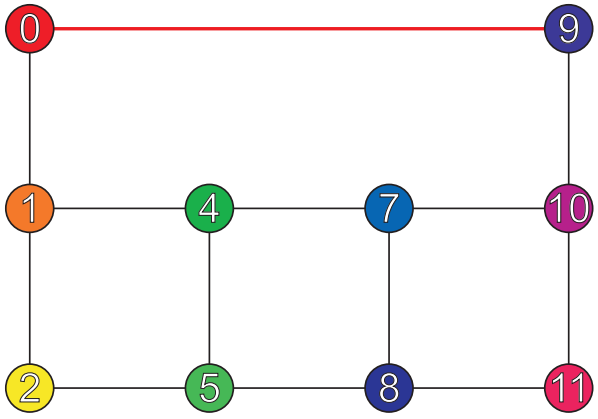


Abbildung 3.7: Graph, wenn bei Global-Lokal die Kanten erst nach jedem Grad entfernt werden.

3.4.3 Global-Global

Bei der Global-Globalen Verknüpfung werden die Knoten so wie bei der Global-Lokalen Verknüpfung vor dem Entfernen sortiert, die Kanten jedoch erst nach dem Entfernen aller Knoten in einer reproduzierbaren Reihenfolge erzeugt. Die Implementierung ist in Listing 5.6 nachzulesen.

Die Reihenfolge der Kanten wird über die Länge des kürzesten Pfades zwischen den zwei Knoten bestimmt, zwischen denen die neue Kante erzeugt werden soll. Die zu erzeugenden Kanten werden dabei nach der Pfadlänge gruppiert. So werden zunächst alle Kanten erzeugt, die einen Pfad der Länge 2 ersetzen. Anschließend wird überprüft, für welche Kanten die in 3.3.1 aufgestellten Bedingungen noch gelten. Nur diese Kanten werden weiterhin betrachtet, denn es ergibt sich zwangsläufig, dass mit dem Erstellen neuer Kanten die Bedingungen für alte Pfade größerer Länge nicht mehr gelten. Zusätzlich ist es auch möglich, dass mit dem Hinzufügen einer Kante bereits eine andere Kante der aktuell bearbeiteten Länge die Bedingungen nicht erfüllt. Es wäre jedoch nicht stabil, die Kante in diesen Fallen nicht hinzuzufügen.

3.5 Stabilität und Determinismus der Algorithmen

Allen Verknüpfungen ist gemein, dass sie deterministisch und stabil sein müssen. Es muss also eine Möglichkeit gefunden werden, bei gleicher Graphenstruktur und den gleichen zu entfernenden Knoten die gleichen neuen Kanten zu erzeugen.

Die Kantenerstellung muss stets deterministisch sein, der Determinismus der Knotenentfernung hängt von der Verknüpfung ab. Die Regeln, wann eine neue Kante erstellt werden soll, wurden bereits in Abschnitt 3.3.1 vorgestellt und sollen hier noch einmal aufgeführt werden:

1. Die Knoten sind Nachbarn eines entfernten Knoten.

2. Die Länge des Pfades zwischen den Knoten ist größer als vor dem Entfernen.

Die Kantenerstellung arbeitet auf den Knoten, folglich betrachten wir zunächst, ob es möglich ist, eine Reihenfolge der Knoten aufzustellen. Auf den ersten Blick vielleicht offensichtliche Eigenschaften wie die `ID` des Knotens oder die Position des Knotens in der Eingabedatei scheiden aus, da diese sich nicht aus der Struktur des Graphen ergeben. Bei genauer Betrachtung wird sogar deutlich, dass es gar keine Eigenschaft gibt, mit der eine feste Reihenfolge einzelner Knoten erzeugt werden kann. Es muss also davon ausgegangen werden, dass sich bei einer Iteration über beispielsweise die Nachbarn eines entfernten Knoten die Reihenfolge dieser Knoten ändert. Relevant wird dies im Zusammenhang mit der zweiten Regel zur Kantenerstellung. Werden neu erstellte Kanten bei dieser Überprüfung mit beachtet, tritt der Fall ein, dass sich die Entfernung zwischen zwei noch zu bearbeitenden Knoten

ändert. Je nachdem, welche Knoten zuvor bearbeitet wurden, trifft also Bedingung 2 zu - oder auch nicht. Das ist jedoch nicht stabil.

Die einfachste Lösung wäre, neue Kanten grundsätzlich nicht in die Berechnung weiterer neuer Kanten einfließen zu lassen. Dies würde jedoch dafür sorgen, dass viel mehr Kanten als notwendig gezogen werden würden. Welche Kanten der Algorithmus in diesem Fall erzeugen würde, ist in Abbildung 3.8 zu sehen. Im Beispiel wurden die Knoten 6, 7 und 8 Global entfernt und die neuen Kanten nur anhand der aufgestellten Bedingungen erzeugt. Die neuen Kanten sind rot markiert und es ist offensichtlich, dass mit Ausnahme der Verbindungen 3-5 und 9-11 alle Nachbarn der entfernten Knoten miteinander verbunden werden würden.

Folglich muss eine Grenze gefunden werden, ab der neue Kanten in der Berechnung beachtet werden. Es bietet sich dabei die ersetzte Entfernung an. Wie bei den Knoten ergibt sich jedoch das Problem der nicht stabilen Reihenfolge. Jedoch lassen sich die zu ersetzenden Entfernungen als Merkmal zur Gruppierung nutzen. Zunächst werden die Kanten mit der kleinsten zu ersetzenden Entfernung erzeugt. Beim Erzeugen dieser werden die neuen Kanten wie ausgeführt jedoch noch nicht beachtet. Erst wenn alle Kanten einer Entfernung erzeugt sind, werden diese in weiteren Berechnungen benutzt. Dadurch ist die Wahrscheinlichkeit hoch, dass es nun Kanten gibt, auf die Bedingung 2 nicht mehr zutrifft. Da die Entfernung zwischen zwei Knoten jedoch deterministisch ist, sind auch die erzeugten Kanten deterministisch und können folglich für die weitere Berechnung verwendet werden. In Abbildung 3.9 ist dies verdeutlicht. Grün markierte Kanten ersetzen einen Pfad der Länge 2, blau markiert sind die Kanten, die aufgrund der gerade dargelegten Eigenschaften nicht erzeugt werden.

Nachdem nun für die Kantenerzeugung eine Möglichkeit der stabilen Erzeugung gefunden wurde, muss dies noch für die Knoten geschehen. Aufgrund der Algorithmenverknüpfung ist dies nur für die Verknüpfung `Global-Lokal` nötig. Bei der Verknüpfung `Lokal-Lokal` wird die Reihenfolge der Knotenentfernung nicht verändert und muss deshalb als zufällig angesehen werden. Bei der Verknüpfung `Global-Global` ist die Reihenfolge, in der die Knoten entfernt werden, irrelevant weil die Kanten erst erzeugt werden, nachdem alle Knoten entfernt wurden.

Analog zur Unterteilung der Kanten in Gruppen können auch die Knoten in deterministische Gruppen unterteilt werden. Unterscheidungsmerkmal ist hier der Grad der Knoten mit der Anweisung, dass Knoten mit niedrigem Grad zuerst entfernt werden. Diese Sortierung wurde in dieser Implementierung gewählt. Es ist auch möglich, erst die Knoten mit dem höchsten Grad zu entfernen. An der Stabilität des Algorithmus ändert dies nichts. Dazu soll kurz gezeigt werden, wie sich das Ergebnis unterscheide. Es werden die Knoten 4, 6, 7 und 8 entfernt und anschließend einmal aufsteigend (Reihenfolge: 6, 8, 4, 7) und einmal absteigend (7, 4, 8, 6) geordnet verarbeitet. Die unterschiedlichen Graphen sind in den Abbildungen 3.10 und 3.11 zu sehen. Es ist offensichtlich, dass sich die erzeugten Kanten unterscheiden, dennoch gibt es Kanten, die in beiden Graphen gleich sind. In beiden Fällen werden die

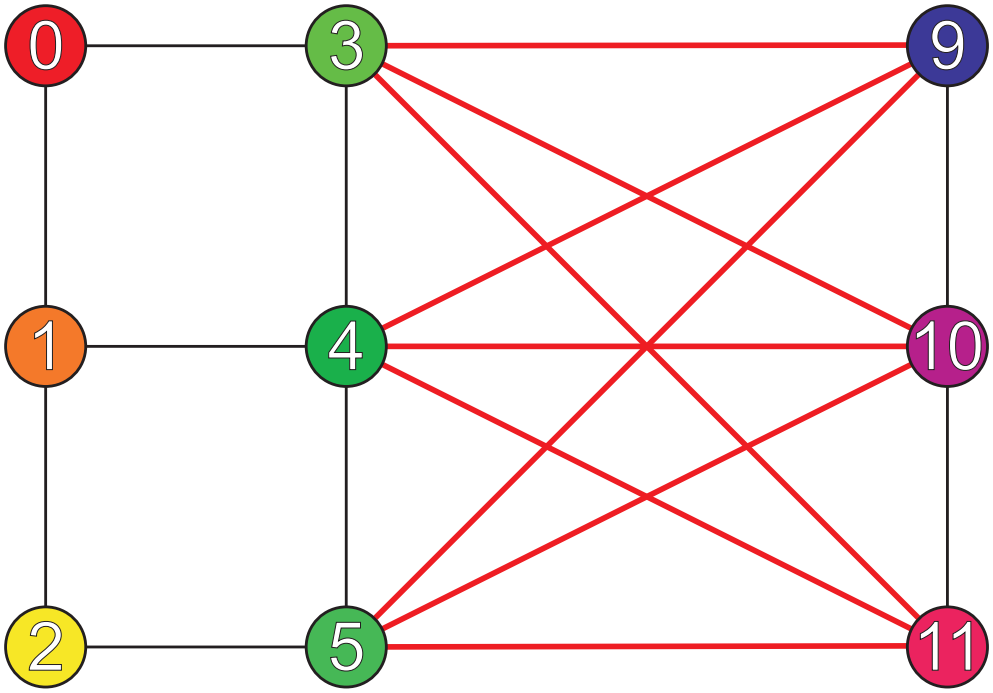


Abbildung 3.8: Erzeugte Kanten, wenn neue Kanten niemals in die Berechnung einfließen.

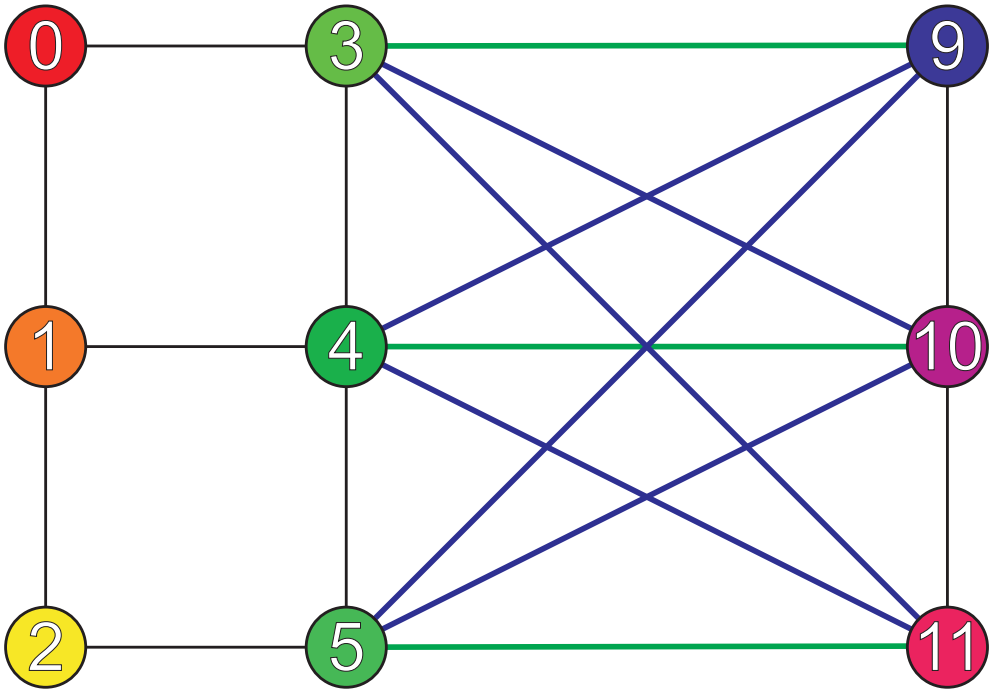


Abbildung 3.9: Erzeugte Kanten, wenn neue Kanten nach Entfernung gruppiert in die Berechnung einfließen.

Kanten 1-10 und 3-5 gezogen, unterschiedlich sind die Kanten 3-9 und 5-11 respektive 3-10 und 5-10. Der Grund für die unterschiedlichen Kanten ist in Knoten 4 zu finden. Bei aufsteigender Sortierung wird Knoten 7 zuletzt entfernt und es wurden beim Entfernen der Knoten 6 und 8 die Kanten 3-9 und 5-11 gezogen. Bei absteigender Sortierung wird nach Knoten 7 noch Knoten 4 entfernt. Dadurch werden die Kanten 3-10 und 5-10 erzeugt. Aufgrund dieser Kanten werden beim Entfernen der Knoten 6 und 8 keine weiteren Kanten erzeugt. Es wird also deutlich, dass zwei unterschiedliche Graphen entstehen, die Erzeugung selbst ist jedoch deterministisch.

3.6 Zeitkomplexität

Die Zeitkomplexität hängt bei allen Verknüpfungen von mehreren Faktoren ab. Allen drei Verknüpfungen ist gemein, dass die Entfernungen zwischen Knoten davor, danach und während des Entfernens bestimmt werden müssen.

Grundlage der Entfernungsbestimmung im Graph ist der Dijkstra-Algorithmus. Die Laufzeit dieses Algorithmus ist abhängig von der Anzahl der Kanten e und Knoten n, siehe dazu (Wikipedia, 2014). Des Weiteren sei d = Anzahl der Dimensionen des Gitters und r = Anzahl der Knoten, die nicht Nachbar eines Nachbarn eines entfernten Knoten sind. Für die hier durchgeführte theoretische Betrachtung werden alle Operationen auf der Datenstruktur mit konstanter Komplexität $O(1)$ angenommen. Anmerkungen zur tatsächlichen Laufzeit folgen in Kapitel 4. Listing 3.3 zeigt den Dijkstra-Algorithmus in Pseudocode nach (Quasthoff, 2009) für einen Knoten s auf einem Graphen $G = (V, E)$.

Listing 3.3: Dijkstra-Algorithmus in Pseudocode

```
1  for each Knoten v ∈ V − s do { D[v] = ∞; };
2  D[s] = 0; PriorityQueue Q = V;
3  while not isEmpty(Q) do {
4    v = extractMinimum(Q);
5    for each u ∈ succ(v) ∩ Q do {
6      if D[v] + g ((v, u)) < D[u] then {
7        D[u] = D[v] + g ((v,u));
8        adjustiere Q an neuen Wert D[u];
9      };
10   };
11 }
```

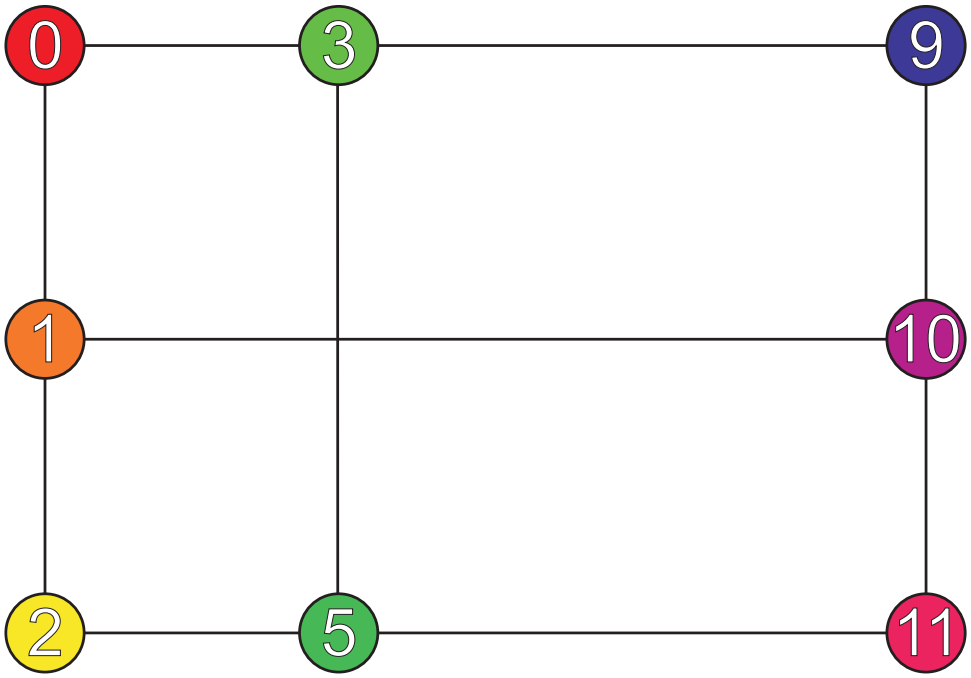


Abbildung 3.10: Graph, wenn bei Global-Lokal die Knoten 4, 6, 7 und 8 entfernt und dann die Knoten aufsteigend geordnet verarbeitet werden.

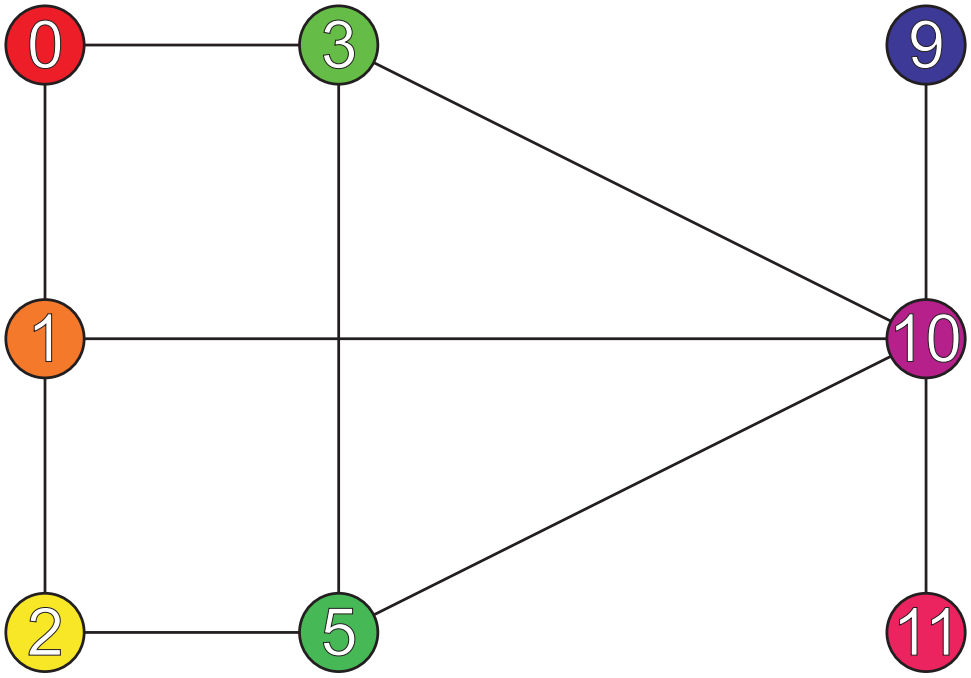


Abbildung 3.11: Graph, wenn bei Global-Lokal die Knoten 4, 6, 7 und 8 entfernt und dann die Knoten absteigend geordnet verarbeitet werden.

Es ist zunächst offensichtlich, dass in jedem Fall alle Knoten abgearbeitet werden, die Laufzeit ist somit mindestens

$$O(n) \tag{3.5}$$

Da auch alle Kanten durchlaufen werden, ergibt sich dafür

$$O(e) \tag{3.6}$$

Die Funktion `extractMinimum(Q)` hat die Komplexität

$$O(\log n) \tag{3.7}$$

Dies ist leicht ersichtlich, denn diese Funktion ist vom Prinzip her eine Suchfunktion, die das oberste Element der (in dem Fall aufsteigend sortierten) Liste zurückgibt. Da Sortieralgorithmen nicht besser als $O(n \log n)$ sein können, ist auch die Komplexität von `extractMinimum(Q)` $O(n \log n)$. Bringt man nun die einzelnen Komplexitäten aus den Formeln 3.5, 3.6 und 3.7 in eine Formel, so ergibt sich für den Dijkstra-Algorithmus die Zeitkomplexität

$$O((n \log n) + e) \tag{3.8}$$

Wie in Abschnitt 3.1.1 dargelegt, handelt es sich bei den hier verarbeiteten Graphen um regelmäßige Gitter. Für ein 2D-Gitter hat ein Knoten somit maximal 4 Nachbarn mit jeweils maximal 4 Nachbarn, für ein 3D-Gitter sind es maximal 6 Nachbarn mit jeweils maximal 6 Nachbarn. Dadurch ist die Anzahl der Kanten abhängig von der Anzahl der Knoten. Geht man von einem unendlichen Gitter aus, so ist

$$e = dn \tag{3.9}$$

Da d konstant ist, entfällt es bei der Komplexitätsbetrachtung und es ergibt sich als Zeitkomplexität für den Dijkstra-Algorithmus in diesem Fall

$$O(n \log n) \tag{3.10}$$

Zur Vor- und Nachbereitung der Knotenentfernung müssen jedoch noch weitere Algorithmen ausgeführt werden. Für die Überprüfung der Entfernung zwischen den verbleibenden Knoten müssen alle Entfernungen zwischen Nachbarn von Nachbarn (= verarbeitete Knoten) der entfernten Knoten durchlaufen werden. Sei k = Anzahl der verarbeiteten Knoten, die sich aus der Anzahl der entfernten Knoten s und der Anzahl Dimensionen d ergibt:

$$k = s((2d)^2)^2 \tag{3.11}$$

Die Zeitkomplexität dieses Durchlaufes ist dann

$$O(k) \tag{3.12}$$

Die Anzahl der Dimensionen ist konstant ist, somit hängt die Zeitkomplexität eines Durchlaufs nicht von der Größe des Graphen, sondern nur von der Anzahl der entfernten Knoten s ab.

Da es bei dieser Nachbearbeitung Knoten gibt, deren Entfernungen für die Kantenerstellung nicht gebraucht werden, können diese nun vor dem Durchlauf des Dijkstra-Algorithmus aus dem Graph entfernt werden. Dessen Laufzeit würde sich mit $r =$ Anzahl der Knoten, die nicht Nachbar eines Nachbarn eines entfernten Knoten sind auf

$$O((n-r)\log(n-r)) \tag{3.13}$$

reduzieren. Dabei ist $r < n$.

4 Implementierung

Während in Kapitel 3 auch auf die Implementierung der Algorithmen eingegangen wurde, sollen hier die weiteren Anpassungen vorgestellt werden, die zur Nutzung der Algorithmen notwendig waren. Dazu wurde das im Rahmen von (Zeckzer u. a., 2008) entwickelte Programm FDLView erweitert.

4.1 Ergänzungen

Zur Nutzung der Algorithmen wurde ein neuer Tab im Bedienfeld eingeführt, sowie einige Klassen erweitert.

4.1.1 Steuerung der Algorithmen

Die Algorithmen werden über das neu eingefügte Panel `Control Nodes` gesteuert, dieses ist zusammen mit dem geladenen Beispielgraphen in Abbildung 4.1 abgedruckt. In der zugehörigen Klasse `NodesControlPanel.java` sind die Algorithmen implementiert, die in dieser Arbeit vorgestellt wurden. Das Panel bietet Funktionen zur Analyse des Graphen sowie die Möglichkeit, Knoten manuell oder automatisch zu entfernen. Die Buttons `Show Node Information`, `Show Distances` und `Show Neighbors` geben Informationen über `System.Err` aus. Diese Ausgaben sind daher nicht in der GUI zu lesen. Knoten mit den Verknüpfungen Global-Lokal und Global-Global werden in einem zweistufigen Verfahren entfernt. Zuerst werden die zu entfernenden Knoten in einer Liste gespeichert. Dies geschieht entweder manuell über den Button `Delete Node`, automatisch über die Funktionswerte mit dem Button `MarkNodesWithOHitsForDeletion` oder zu Testzwecken hälftig mit dem Button `MarkHalfOfAllNodesForDeletion`. Anschließend werden die Knoten entfernt, je nach gewünschter Verknüpfung mit dem Button `DeleteNodesGlobalLocal` oder `DeleteNodesGlobalGlobal`. Die Funktion der einzelnen Buttons ist in der folgenden Übersicht zusammengefasst.

Show Node Information Gibt auf der System.Err-Ausgabe einige grundlegende Informationen zum selektierten Knoten aus.

Delete Node Entfernt den selektierten Knoten, entspricht der Verknüpfung Lokal-Lokal.

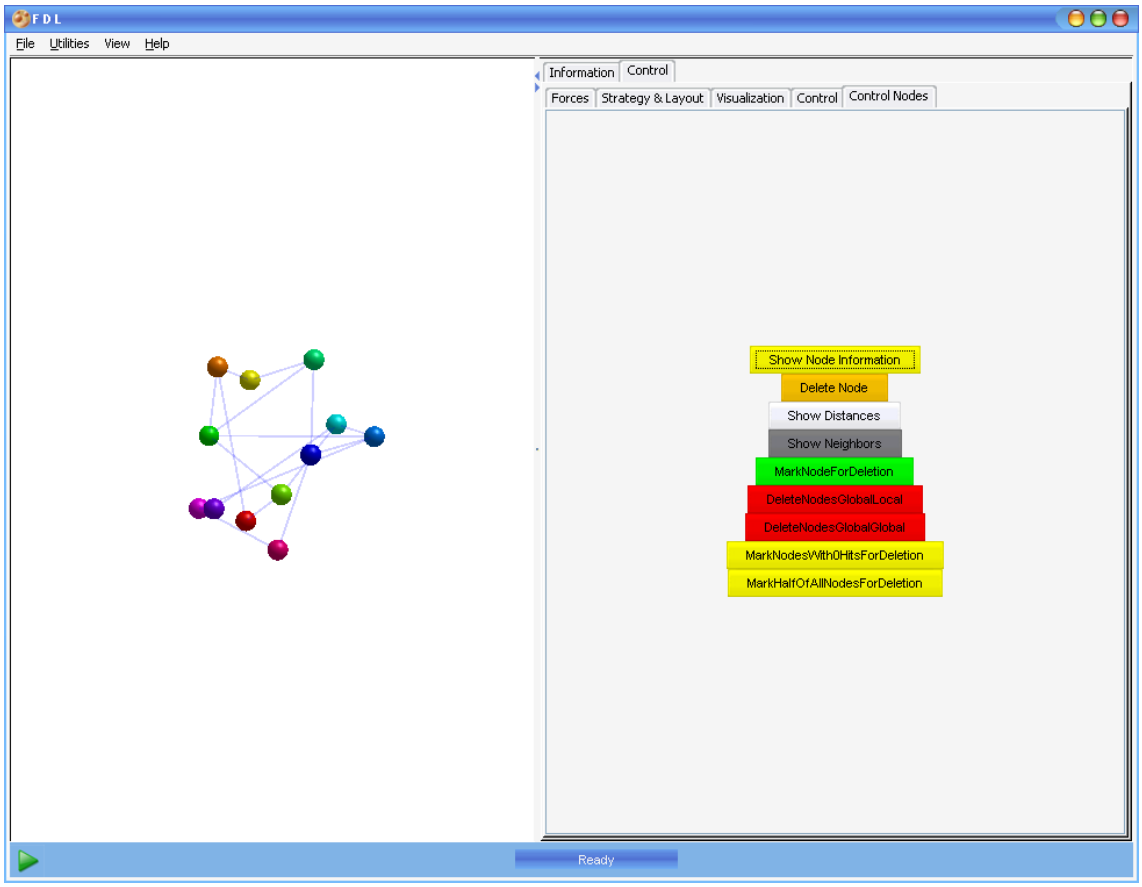


Abbildung 4.1: Screenshot der UI zum Entfernen von Knoten

Show Distances Gibt auf der System.Err-Ausgabe die Entfernungsmatrix des geladenen Graphen aus.

Show Neighbors Gibt auf der System.Err-Ausgabe die Nachbarn des selektierten Knotens aus.

MarkNodeForDeletion Markiert den selektierten Knoten zum Entfernen.

DeleteNodesGlobalLocal Entfernt alle zum Entfernen markierten Knoten mit der Algorithmenverknüpfung Global-Lokal.

DeleteNodesGlobalGlobal Entfernt alle zum Entfernen markierten Knoten mit der Algorithmenverknüpfung Global-Global.

MarkNodesWith0HitsForDeletion Markiert alle Knoten zum Entfernen, deren Funktionswert (siehe Abschnitt 3.1.2) gleich 0 ist.

MarkHalfOfAllNodesForDeletion Markiert zu Testzwecken die Hälfte aller Knoten zum Entfernen. Im Beispielgraph führt dies dazu, dass lediglich Knoten entfernt werden, aber keine neuen Kanten erzeugt werden.

4.1.2 2D/3D-Ansicht

Wie in Abschnitt 3.1 dargelegt, sind die zu verarbeitenden Graphen zwei- oder dreidimensionale Gitter. Die Darstellung und Verarbeitung dieser Graphen soll somit auch, entsprechend ihrer Struktur, in 2D oder 3D erfolgen. Da sowohl die Darstellung, als auch die Verarbeitung auf Basis der Knotenkoordinaten erfolgen, genügt es, diese entsprechend zu modifizieren. Im Programm ist jeder Knoten auf 3 Achsen abgetragen. Für die 2D-Ansicht wird nun die Z-Achse für alle Knoten 0 gesetzt, sodass sich alle Knoten auf der X-Y-Ebene befinden. Der Force-Directed Layout Algorithmus (siehe (Zeckzer u. a., 2008)) modifiziert die Knoten anhand ihrer Positionen, sodass die Z-Achse, wenn sie auf 0 gesetzt ist, vom Algorithmus nicht weiter modifiziert wird. Beim Umschalten auf 3D werden jedoch nicht wie beim Laden des Graphen die Positionen der Knoten zufällig gesetzt, sodass die Werte für die Z-Achse weiter 0 bleiben. Dies führt dazu, dass ein Graph, der in 2D verarbeitet wurde, mit der derzeitigen Implementierung nicht wieder in einen 3D-Graphen überführt werden kann. Das Setzen der Z-Achse geschieht in der `iterate`-Methode der `Algorithms.java`, siehe Listing 5.7. Diese Methode wird u.a. aufgerufen, wenn der Graph neu gezeichnet wird. Um sicherzustellen, dass nach einem Wechsel der Anzahl der Dimensionen die Daten aktuell sind, wird der Graph neu gezeichnet, sobald sich die Dimensionen ändern. Abbildung 4.2 zeigt den Beispielgraph in 3D, Abbildung 4.3 den Graph nach dem Umschalten auf 2D-Verarbeitung. In den Abbildungen 4.4 und 4.5 wurde zusätzlich der Layout-Algorithmus ausgeführt.

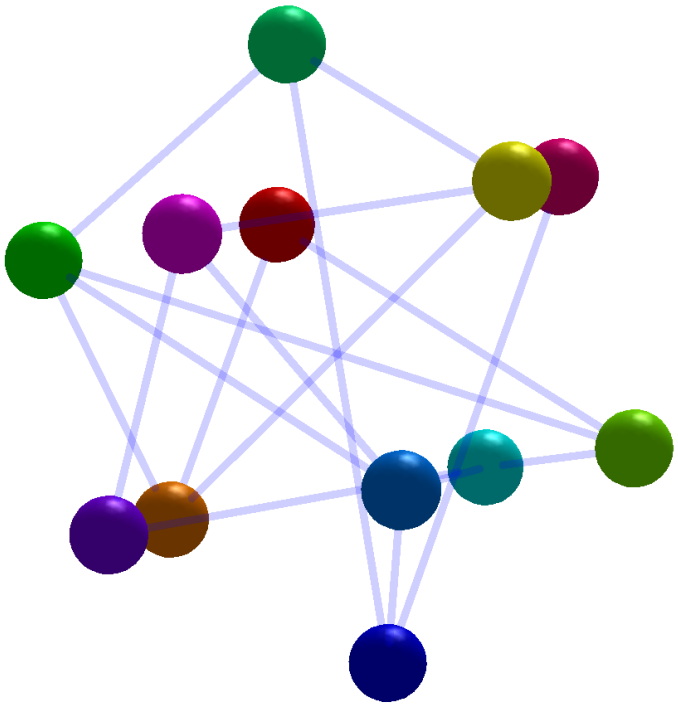


Abbildung 4.2: Beispielgraph nach Layout in 3D.

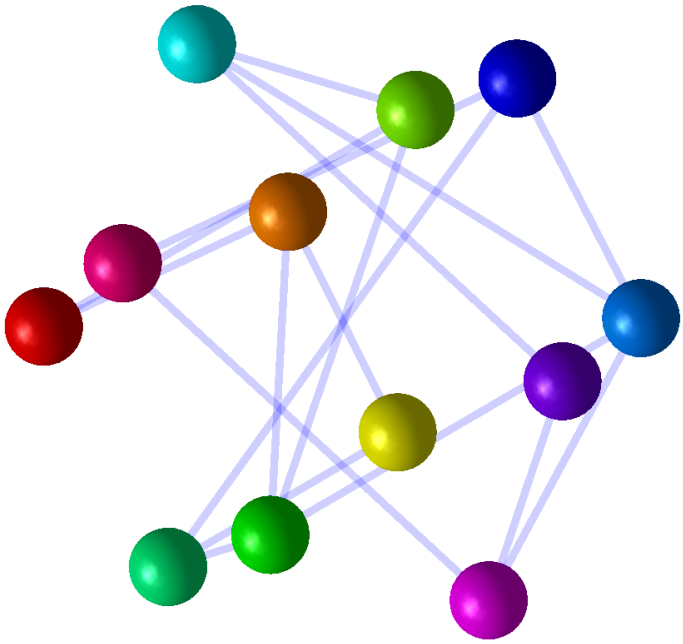


Abbildung 4.3: Beispielgraph nach Layout in 3D und Umschalten der Verarbeitung auf 2D.

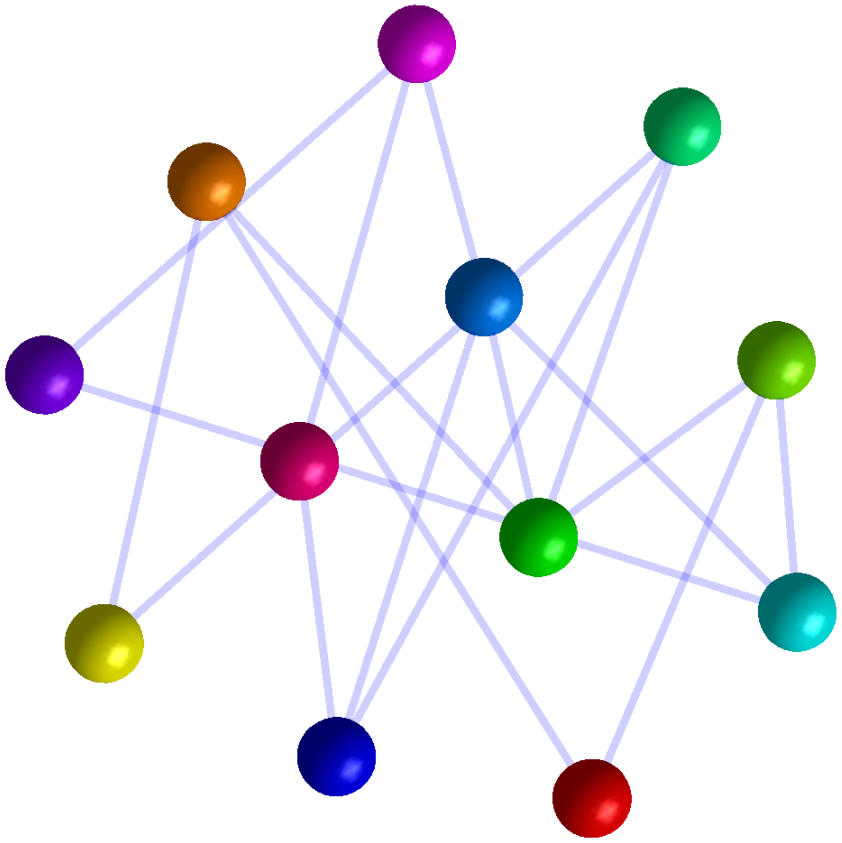


Abbildung 4.4: Beispielgraph nach Layout in 2D (Aufsicht).

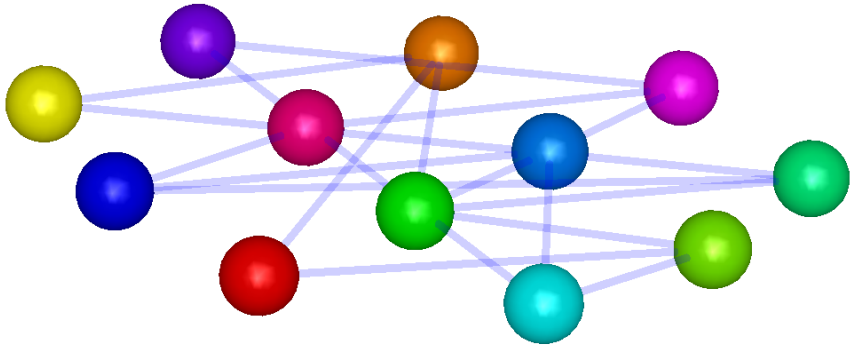


Abbildung 4.5: Beispielgraph nach Layout in 2D (Ansicht von schräg).

4.1.3 Importer

Neu geschrieben wurden die Klassen `CSVImporter` und `HitsImporter`. Die Funktionsweise der Importer ist bereits in Abschnitt 3.1 beschrieben worden. Bei der Implementierung musste beachtet werden, dass die vorhandene Klassen `Cluster`, `Node` und `Edge` sehr viele Eigenschaften besitzen, die gesetzt werden müssen, jedoch aus einer einfachen CSV nicht gelesen werden können. Der überwiegende Teil der Werte wird mit experimentell ermittelten Werten gesetzt, andere Werte wie z.B. die Beschreibung aus anderen Eigenschaften (wie z.B. der Knoten-ID) abgeleitet.

4.1.4 Ergänzungen an anderen Klassen

Um die Funktionswerte verarbeiten zu können, wurden einige Klassen ergänzt. Die Klasse `Node`, siehe Listing 5.8, erhielt die zusätzliche Property `Hits`, welche im Zuge des Imports der Funktionswerte gesetzt wird. Da die Knoten nach ihrem Grad sortiert werden müssen, wurde zusätzlich die Property `Grade` hinzugefügt, die den Grad eines Knotens angibt. Um mit dieser besser arbeiten zu können, implementiert die Klasse das Interface `Comparable`, das den Grad vergleicht. Da ein Knoten seine Nachbarn nicht kennt, wird diese Eigenschaft bei der Verarbeitung in den Algorithmen gesetzt.

In der Klasse `Model` wurden Funktionen zur Entfernungsbestimmung implementiert, die Methode zum Entfernen von Knoten im Model sowie eine Methode zur Bestimmung von Knotennachbarn des höchsten Funktionswertes. Diese sind in Listing 5.9 abgedruckt.

4.2 Offene Punkte

Es gibt in der Implementierung einige Punkte, die nicht vollständig umgesetzt wurden.

4.2.1 UI-Verbesserung

Die derzeitige UI zur Steuerung der Algorithmen, abgebildet in Abbildung 4.1, ist funktional, aber nicht ergonomisch. Vergleicht man diese beispielsweise mit der UI zur Visualisierung in Abbildung 4.6, so wird deutlich, dass diese wesentlich geordneter und aufgeräumter ist.

Ein weiterer Punkt ist das umständliche manuelle Löschen von Knoten. Kann der zu löschende Knoten nicht in der gerenderten Darstellung markiert werden, so muss er über die Knotenliste markiert und anschließend die jeweilige Aktion im Algorithmen-Panel ausgelöst werden. Dies erfordert für jeden Knoten mehrere Mausklicks, sodass das Entfernen mehrerer Knoten sehr viele Benutzeraktionen

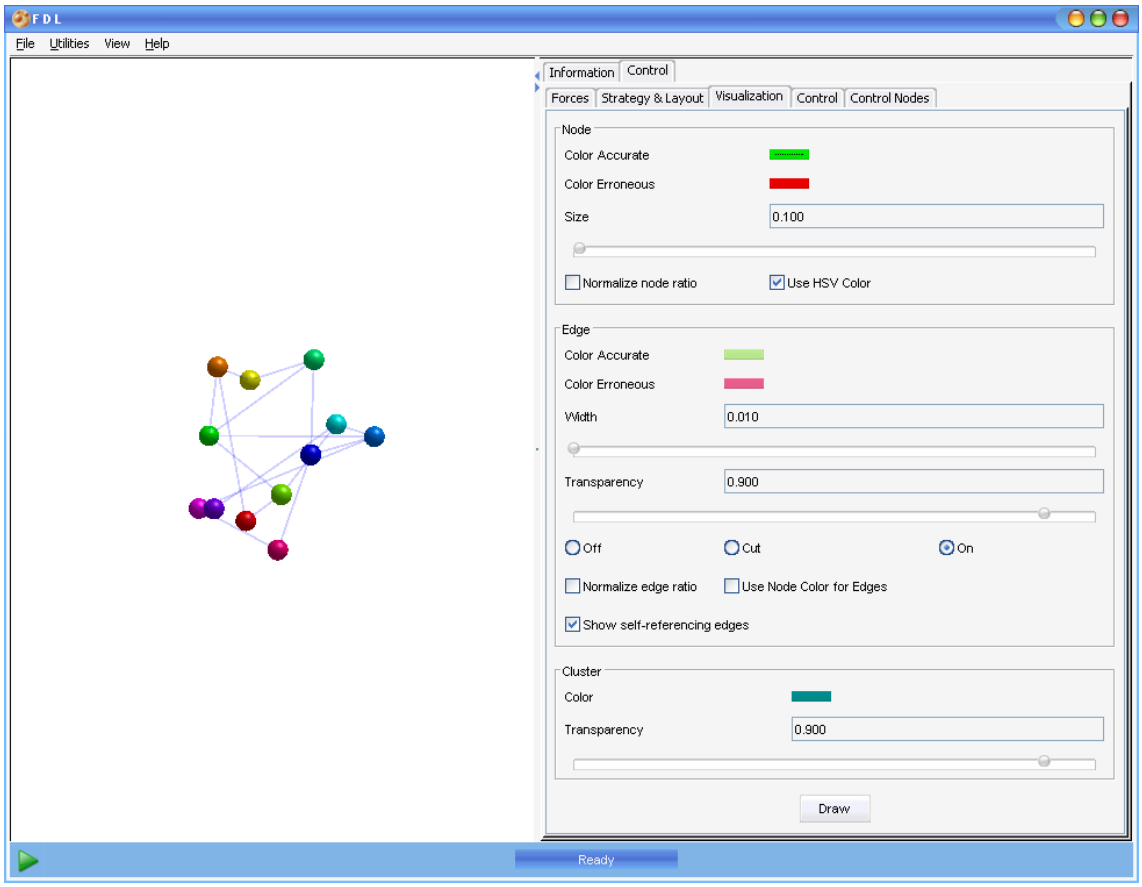


Abbildung 4.6: Screenshot der UI zum Einstellen der Visualisierungsoptionen

erfordert. Es ist zu überlegen, hier eine benutzerfreundlichere Lösung zu implementieren. Auch wenn Knoten standardmäßig über die Funktionswerte (siehe Abschnitt 3.1.2) entfernt werden, so sollte doch das manuelle Entfernen benutzerfreundlich sein.

4.2.2 Architektur

Sämtliche Algorithmen zur Knotenentfernung wurden als private Funktionen in der Datei `NodesControlPanel.java` implementiert. Als Parameter werden entweder nur der zu entfernende Knoten oder gar nichts übergeben, die Funktion arbeitet stets auf dem Model des übergeordneten FDLViews. Dies genügt für eine prototypische Implementierung wie sie hier vorliegt, nicht jedoch für einen produktiven Einsatz, bei dem die entsprechenden Funktionen möglicherweise erweitert und durch Unit-Tests auf Funktion geprüft werden müssen.

4.2.3 Laufzeit

Die in Abschnitt 3.6 angegebene Zeitkomplexität wird vom Programm nicht erreicht. Ursache dafür ist, dass die Vor- und Nachbereitung der Algorithmen suboptimal programmiert ist. Das meiste Optimierungspotential hat dabei die Entfernungsberechnung. Bei dieser wird für jedes Knotenpaar der komplette Dijkstra-Algorithmus für das jeweilige Paar als Start und Ziel ausgeführt. Besser wäre es, einmal komplett die Entfernungsmatrix zu berechnen und dann lediglich daraus zu lesen. Des Weiteren ist noch zu evaluieren, ob die gewählten Datenstrukturen die schnellstmöglichen sind und ob beispielsweise die Entfernungen nicht besser in einem Array fester Größe statt in einer HashMap gespeichert werden sollten.

4.2.4 Java 8

Der Quellcode wurde unter Java 7 entwickelt. Während diese Arbeit geschrieben wurde, veröffentlichte Oracle Java 8. Eine wesentlich Neuerung dieses Releases sind Lambda-Expressions, siehe (Oracle, 2014). Diese standen zu der Zeit, als die für diese Arbeit entwickelten Quelltexte geschrieben wurden, noch nicht zur Verfügung. Daher sind an einigen Stellen Schleifenkonstrukte entstanden, um die Funktion entsprechender Lambda-Expressions abzubilden. Dies führt jedoch zu sehr unschönem Quelltext. Beispielhaft sollen an dieser Stelle die Funktionen
`MarkNodesWith0HitsForDeletion()` (siehe Listing 5.10) und
`GetNeighbors(Node n)` (siehe Listing 5.9)
erwähnt werden.In der Funktion `MarkNodesWith0HitsForDeletion()` wird über eine For-Each-Schleife über alle Knoten iteriert. Diese Iteration könnte durch die Lambda-Expression `node -> node.getHits() == 0` ersetzt werden. In der Funktion `GetNeighbors(Node n)` werden über zwei Schleifen alle Kanten darauf überprüft,

ob ein Knoten Punkt der Kante ist und die Kante einer ArrayList hinzugefügt. Diese Arraylist könnte nun über die Lambda-Expression

```
edge -> edge.getElement1Id().equals(n.getId()) ||
edge.getElement2Id().equals(n.getId())
```

erstellt werden. Der Quelltext würde damit wesentlich kürzer und da die Auswertung von der VM durchgeführt wird, vermutlich auch schneller. Letzteres hängt von der Implementierung in der Java Runtime ab, dazu gibt es jedoch noch keine Praxiserfahrung.

5 Quellcode

Listing 5.1: Implementierung der Kantenlaenge

```
1  private Edge CreateEdgeFromPath(List<Edge> path, Nodes nodes)
2  {
3    Edge e1 = path.get(path.size()-2);
4    Edge e2 = path.get(path.size()-1);
5    Node n1 = (Node)e1.getElement1();
6    Node n2 = (Node)e2.getElement2();
7    Edge e = new Edge(n1.getId(), n2.getId());
8
9    double angle = Tools.calculateAngle(e1, e2);
10   double length = Tools.calculateLength(angle,
       e1.getDesiredLength(), e2.getDesiredLength());
11   e.setDesiredLength(length);
12   e.SetElement1(e1.getElement1());
13   e.SetElement2(e2.getElement2());
14
15   if (path.size()>2)
16   {
17     path.remove(path.size()-1);
18     path.remove(path.size()-1);
19     path.add(e);
20     return CreateEdgeFromPath(path, nodes);
21   }
22     else
23   {
24     return e;
25   }
26 }
```

Listing 5.2: Implementierung der Laengenberechnung mit SWS

```
1 public static double calculateLength(double angle, double
     length1, double length2)
2 {
3   angle = Math.toRadians(angle);
4   return Math.sqrt((length1 * length1) + (length2 * length2) -
       (2 * length1 * length2 * Math.cos(angle)));
5 }
```

Listing 5.3: Implementierung der Winkelberechnung

```
1  public static double calculateAngle(Edge e1, Edge e2) throws
       Exception
2  {
3    Node edge1;
4    Node edge2;
5    Node angleNode;
6
7    if (e1.getElement1().equals(e2.getElement1()))
8    {
9      angleNode = (Node) e1.getElement1();
10     edge1 = (Node) e1.getElement2();
11     edge2 = (Node) e2.getElement2();
12   } else if (e1.getElement2().equals(e2.getElement1()))
13   {
14     angleNode = (Node) e1.getElement2();
15     edge1 = (Node) e1.getElement1();
16     edge2 = (Node) e2.getElement2();
17   } else if (e1.getElement1().equals(e2.getElement2()))
18   {
19     angleNode = (Node) e1.getElement1();
20     edge1 = (Node) e1.getElement2();
21     edge2 = (Node) e2.getElement1();
22   } else if (e1.getElement2().equals(e2.getElement2()))
23   {
24     angleNode = (Node) e1.getElement2();
25     edge1 = (Node) e1.getElement1();
26     edge2 = (Node) e2.getElement1();
27   } else
28   {
29     throw new Exception();
30   }
31
32   //according to
33   //http://www,frustfrei-lernen.de/mathematik/
         schnittwinkel-zweier-geraden.html
34
35   Point3d edge1Point = edge1.getPosition();
36   Point3d edge2Point = edge2.getPosition();
37   Point3d angleNodePoint = angleNode.getPosition();
38
39   double vx1 = angleNodePoint.x - edge1Point.x;
40   double vy1 = angleNodePoint.y - edge1Point.y;
41   double vz1 = angleNodePoint.z - edge1Point.z;
```

```
42    double vx2 = angleNodePoint.x - edge2Point.x;
43    double vy2 = angleNodePoint.y - edge2Point.y;
44    double vz2 = angleNodePoint.z - edge2Point.z;
45
46    double a1Timesa2 = (vx1 * vx2) + (vy1 * vy2) + (vz1 * vz2);
47    double Betraga1 = Math.sqrt((vx1 * vx1) + (vy1 * vy1) + (vz1
          * vz1));
48    double Betraga2 = Math.sqrt((vx2 * vx2) + (vy2 * vy2) + (vz2
          * vz2));
49
50    double arccos = Math.acos(a1Timesa2 / (Betraga1 * Betraga2));
51
52    double angle = Math.toDegrees(arccos);
53
54    if (angle < 0)
55    {
56      angle = angle * (-1);
57    }
58    return angle;
59  }
```

Listing 5.4: UnitTests zur Winkelberechnung

```
1  @Test
2  public void TestAngle()
3  {
4    Point3d anglepoint = new Point3d(3, 6, 11);
5    Point3d edge1point = new Point3d(4, 2, 8);
6    Point3d edge2point = new Point3d(5, 8, 21);
7
8    Node anglenode = new Node();
9    anglenode.setPosition(anglepoint);
10   anglenode.setId("anglenode");
11   Node edge1node = new Node();
12   edge1node.setPosition(edge1point);
13   edge1node.setId("edge1node");
14   Node edge2node = new Node();
15   edge2node.setPosition(edge2point);
16   edge2node.setId("edge2node");
17
18   Nodes n = new Nodes();
19   n.add(anglenode);
20   n.add(edge1node);
21   n.add(edge2node);
22
23   Edge e1 = new Edge("anglenode", "edge1node");
24   e1.linkEdge(n);
25   Edge e2 = new Edge("anglenode", "edge2node");
26   e2.linkEdge(n);
27
28   try
29   {
30     double angle = Tools.calculateAngle(e1, e2);
31     assertEquals(132.77, angle, 0.1);
32   } catch (Exception ex)
33   {
34   }
35 }
36
37 @Test
38 public void TestCalculateLength()
39 {
40   assertEquals(8.45, Tools.calculateLength(50, 10, 10), 0.1);
41   assertEquals(7.79, Tools.calculateLength(50, 10, 5), 0.1);
42
43   Point3d anglepoint = new Point3d(0, 0, 0);
```

```
44    Point3d edge1point = new Point3d(0, 1, 0);
45    Point3d edge2point = new Point3d(1, 1, 1);
46
47    Node anglenode = new Node();
48    anglenode.setPosition(anglepoint);
49    anglenode.setId("anglenode");
50    Node edge1node = new Node();
51    edge1node.setPosition(edge1point);
52    edge1node.setId("edge1node");
53    Node edge2node = new Node();
54    edge2node.setPosition(edge2point);
55    edge2node.setId("edge2node");
56
57    Nodes n = new Nodes();
58    n.add(anglenode);
59    n.add(edge1node);
60    n.add(edge2node);
61
62    Edge e1 = new Edge("anglenode", "edge1node");
63    e1.linkEdge(n);
64    Edge e2 = new Edge("anglenode", "edge2node");
65    e2.linkEdge(n);
66
67    try
68    {
69      double angle = Tools.calculateAngle(e1, e2);
70      assertEquals(1, Tools.calculateLength(angle,
          e1.getDesiredLength(), e2.getDesiredLength()), 0.1);
71    } catch (Exception ex)
72    {
73    }
74  }
75
76  @Test
77  public void TestAngle2()
78  {
79    Point3d anglepoint = new Point3d(0, 0, 0);
80    Point3d edge1point = new Point3d(0, 1, 0);
81    Point3d edge2point = new Point3d(1, 1, 0);
82
83    Node anglenode = new Node();
84    anglenode.setPosition(anglepoint);
85    anglenode.setId("anglenode");
86    Node edge1node = new Node();
```

```
87    edge1node.setPosition(edge1point);
88    edge1node.setId("edge1node");
89    Node edge2node = new Node();
90    edge2node.setPosition(edge2point);
91    edge2node.setId("edge2node");
92
93    Nodes n = new Nodes();
94    n.add(anglenode);
95    n.add(edge1node);
96    n.add(edge2node);
97
98    Edge e1 = new Edge("anglenode", "edge1node");
99    e1.linkEdge(n);
100   Edge e2 = new Edge("anglenode", "edge2node");
101   e2.linkEdge(n);
102
103   try
104   {
105     double angle = Tools.calculateAngle(e1, e2);
106     assertEquals(45, angle, 0.1);
107   } catch (Exception ex)
108   {
109   }
110 }
```

Listing 5.5: Knotenentfernung Lokal-Lokal

```
1  private void DeleteNode(Node node) {
2    System.err.println("Removing: " + node.getId());
3    Model m = _parent.GetModel();
4    this.ResetNewEdges();
5    HashMap<String, HashMap<String, Integer>> DistancesOld =
       m.GetDistances();
6    ArrayList<Node> Neighbors = m.GetNeighbors((Node) node);
7    m.DeleteNode(node);
8    HashMap<String, HashMap<String, Integer>> DistancesNew =
       m.GetDistances();
9    for (Node neighbor1 : Neighbors)
10   {
11     for (Node neighbor2 : Neighbors)
12     {
13       Integer DistanceOld = null;
14       Integer DistanceNew = null;
15
16       HashMap<String, Integer> getOld =
           DistancesOld.get(neighbor1.getId());
17       if (getOld != null)
18       {
19         DistanceOld = getOld.get(neighbor2.getId());
20       } else
21       {
22         DistanceOld = 0;
23       }
24
25       HashMap<String, Integer> getNew =
           DistancesNew.get(neighbor1.getId());
26       if (getNew != null)
27       {
28         DistanceNew = getNew.get(neighbor2.getId());
29       } else
30       {
31         DistanceNew = Integer.MAX_VALUE;
32       }
33
34       if (DistanceOld == null || DistanceNew == null ||
           DistanceOld < DistanceNew)
35       {
36         System.err.println("Distance longer: " +
             neighbor1.getId() + "-" + neighbor2.getId());
```

```
37              Edge newEdge = new Edge(neighbor1.getId(),
                    neighbor2.getId());
38              //EdgeStuff
39              newEdge.SetStandardValues();
40
41              newEdge.setColor1(neighbor1.getColor());
42              newEdge.setColor2(neighbor2.getColor());
43
44              m.putEdge(newEdge);
45            }
46        }
47    }
48 }
```

Listing 5.6: Knotenentfernung Global-Global

```
1
2  private void DeleteNodesGlobalGlobal()
3  {
4  Model m = _parent.GetModel();
5  if (m == null)
6  {
7    System.err.println("Model was NULL!");
8    return;
9  }
10 this.ResetNewEdges();
11 ArrayList<Node> Neighbors = new ArrayList<Node>();
12 for (Node x : _nodesToBeDeleted)
13 {
14   x.setGrade(m.GetNeighbors(x).size());
15   for (Node neighbor : m.GetNeighbors((Node) x))
16   {
17     if (!Neighbors.contains(neighbor))
18     {
19       Neighbors.add(neighbor);
20     }
21   }
22 }
23 for (Node x : _nodesToBeDeleted)
24 {
25   if (Neighbors.contains(x))
26   {
27     Neighbors.remove(x);
28   }
29 }
30 Date start = new Date();
31
32 //Create Distances in Original Model.
33 HashMap<String, HashMap<String, Integer>> DistancesOld =
     m.GetDistances();
34 HashMap<String, HashMap<String, List<Edge>>> PathsOld =
     m.GetDistancesLenghtsPaths();
35 Nodes NodesOld = m.getAllNodes();
36
37 //Create special Distances List;
38 ArrayList<Distance> DistancesOldPrepared = new ArrayList<>();
39 HashMap<Integer, ArrayList<Distance>> DividedDistances = new
     HashMap<>();
40
```

```
41 for (Node node1 : Neighbors)
42 {
43   for (Node node2 : Neighbors)
44   {
45     Distance d = new Distance(node1, node2,
         DistancesOld.get(node1.getId()).get(node2.getId()));
46     DistancesOldPrepared.add(d);
47     if (DividedDistances.containsKey(d.distance))
48     {
49       DividedDistances.get(d.distance).add(d);
50     } else
51     {
52       DividedDistances.put(d.distance, new
           ArrayList<Distance>());
53       DividedDistances.get(d.distance).add(d);
54     }
55   }
56 }
57
58
59 //Sort this List
60 Collections.sort(DistancesOldPrepared);
61
62 //remove All Nodes that shall be removed
63 for (int i = 0; i < _nodesToBeDeleted.size(); i++)
64 {
65   System.err.println(i + 1 + "/" + _nodesToBeDeleted.size());
66   System.err.println("Removing: " +
       _nodesToBeDeleted.get(i).getId());
67   m.DeleteNode(_nodesToBeDeleted.get(i));
68 }
69
70 // get New Distances
71 HashMap<String, HashMap<String, Integer>> DistancesNew;
72
73 Boolean finished = false;
74 //Explaination:
75 //We start at Distance 1.
76 //There are no "holes", e.g. Nodes with distance 4 and 6, but
     not with distance 5
77 //Therefore we stop when a key does not exist in
     DividedDistances
78 for (int i = 1; finished == false; i++)
79 {
```

```
80    System.err.println("Working on Distance " + i);
81    if (!DividedDistances.containsKey(i))
82    {
83      finished = true;
84    } else
85    {
86      DistancesNew = m.GetDistances();
87      ArrayList<Distance> Distances = DividedDistances.get(i);
88      for (Distance d : Distances)
89      {
90        if (DistancesNew.containsKey(d.node1.getId()))
91        {
92          if (DistancesNew.get(d.node1.getId())
              .containsKey(d.node2.getId()))
93          {
94            if (DistancesNew.get(d.node1.getId())
                .get(d.node2.getId()) > d.distance)
95            {
96              List<Edge> path = PathsOld
                  .get(d.node1.getId()).get(d.node2.getId());
97
98              //This is a very, very, very bad Bugfix.
99              if (path.size() == 0)
100             {
101               List<Edge> pathReversed = PathsOld
                    .get(d.node2.getId()).get(d.node1.getId());
102               for (int ii = pathReversed.size() - 1; ii >= 0;
                      ii--)
103               {
104                 Edge e = pathReversed.get(ii);
105                 Element e1 = e.getElement1();
106                 Element e2 = e.getElement2();
107                 e.SetElement1(e2);
108                 e.SetElement2(e1);
109                 path.add(e);
110               }
111             }
112             Edge newEdge = CreateEdgeFromPath(path, NodesOld);
113
114             newEdge.setColor1(new Color3f((float) 0.5, (float)
                    0.0, (float) 0.0));
115             newEdge.setColor2(new Color3f((float) 0.5, (float)
                    0.0, (float) 0.0));
116
```

```
117                 newEdge.linkEdge(m.getAllNodes());
118
119                 m.putEdge(newEdge);
120                 System.err.println("New Edge: " + d.node1.getId() +
                        "-" + d.node2.getId());
121               }
122             }
123           }
124         }
125       }
126   }
127
128   Date end = new Date();
129   long diffInMilliSeconds = (end.getTime() - start.getTime());
130   System.err.println("Removing took " + diffInMilliSeconds +
        "ms.");
131
132   _nodesToBeDeleted.clear();
133
134   ModelPreprocessor.buildModel(m);
135   ModelPreprocessor.prepareModel(m);
136
137   _parent.UpdatePanels();
138
139   _parent.SetModel(m);
140   _parent.synchViews(m);
141   _parent.redraw();
142   }
```

Listing 5.7: Iterate-Methode in Algorithmen.java

```
1  public static boolean iterate(Model model, int ipf, int
       dimensions) {
2    boolean converged = false;
3
4    double maxMovement = 2.0 * Constants.minMovement;
5    double averageMovement = 0.0;
6    double movement = 0.0;
7
8    for (int i = 0; (i < ipf) && (!converged); i++) {
9      // Calculate new forces
10     resetForces(model.getAllClusters(), model.getAllNodes());
11     switch (Strategy.strategy) {
12       case Strategy.BASIC:
13         calculateRepulsionForces(model.getRepCons());
14         break;
15       case Strategy.CLUSTER_REPULSION:
16         calculateRepulsionForces(model.getRoot());
17         break;
18     }
19     calculateSpringForces(model.getAllEdges());
20     calculateOriginForces(model.getAllClusters(),
         model.getAllNodes());
21
22     // Propagate cluster forces
23     model.getRoot().propagateForce(null);
24
25     // Compute resulting force and move nodes to new position
26     maxMovement = 0.0;
27     averageMovement = 0.0;
28     for (Node node : model.getAllNodes()) {
29       //node.debugForces();
30
31       // Saves the old position. Necessary for calculating the
           movement later.
32       node.savePosition();
33
34       // clips too large forces
35       node.clipForce();
36
37       // Cools down or heats up the element, depending on it's
           change in the direction.
38       node.cooling();
39
```

```
40          // Finally changes the position of the element + it's
                children
41          node.changePositionByForce();
42
43          // Saves the current force. Necessary for calculating the
                temperature.
44          node.saveForce();
45
46          if (dimensions == 2)
47          {
48            Point3d p = node.getPosition();
49            p.setZ(0);
50            node.setPosition(p);
51          }
52
53          // Compute largest movement of a node.
54          movement = node.getMovement();
55          if (maxMovement < movement) {
56            maxMovement = movement;
57          }
58          averageMovement += movement;
59          //node.debugForces();
60            }
61
62        // Compute cluster position and radius
63        model.getRoot().calculateGeometry(true);
64
65        averageMovement /= model.getAllNodes().size();
66        converged = (maxMovement < Constants.minMovement);
67        /*
68        System.out.println("Movement: " +
69        averageMovement + "|" +
70        maxMovement);
71        */
72
73        ++currentStep;
74      }
75
76      if (converged) {
77        System.out.println("Converged!");
78      }
79      // Completed iteration
80      return converged;
81  }
```

Listing 5.8: Relevante Teile der Klasse Node.java

```
1  /**
2   *
3   * @author Timo Klein
4   * @author Leon Schroeder
5   * @author Dirk Zeckzer
6   * @author Rico Feist
7   */
8  // Package
9  package de.kl.fdl.model.structures;
10
11 //Import
12 import java.util.ArrayDeque;
13 import java.util.Queue;
14 import javax.vecmath.Color3f;
15
16 import javax.vecmath.Point3d;
17
18 /**
19  * Stores the information about one Element.<p> This model does
       not distinguish
20  * between nodes and clusters directly. Clusters are just
       elements with
21  * children.
22  *
23  */
24 public class Node extends Element implements Comparable<Node>
25 {
26
27   public final static double DEFAULT_RADIUS = 0.1;
28   private static final double COOLING_STEP = 0.1;
29   private static final double HEATING_STEP = 0.04;
30 // Old position and movements, for termination condition
31   private Point3d oldPosition = new Point3d(0.0, 0.0, 0.0);
32   protected Queue<Double> movementQueue = new
       ArrayDeque<Double>();
33   protected double sumMovement = 0.0;
34   protected int hits = 0;
35
36   /**
37   * Update color of this element.
38   *
39   * @param fdlColors color and transparency settings
40   */
```

```
41    public void updateColor(FDLColors fdlColors)
42    {
43      if (nodeDescription != null)
44      {
45        float ratio = 0.0f;
46
47        if (fdlColors.isNormalizeRatioNode())
48        {
49          ratio = (float) nodeDescription.getNormalizedRatio();
50        } else
51        {
52          ratio = (float) nodeDescription.getRatio();
53        }
54
55        if (this.getHits() == 0)
56        {
57          color = new Color3f((float) 0, (float) 0, (float) 0);
58        } else
59        {
60          color = fdlColors.computeNodeColor(ratio);
61        }
62        graRadius = radius = fdlColors.getNodeRadius();
63      }
64    }
65
66    @Override
67    public int compareTo(Node b)
68    {
69      return (Integer.compare(this.getGrade(), b.getGrade()));
70    }
71  }
```

Listing 5.9: Relevante Teile der Klasse Model.java

```
1
2  /**
3   *
4   * @author Timo Klein
5   * @author Leon Schroeder
6   * @author Dirk Zeckzer
7   */
8  // Package
9  package de.kl.fdl.model;
10
11 // Import
12 import java.util.LinkedList;
13 import java.util.List;
14
15 import de.kl.fdl.model.structures.Cluster;
16 import de.kl.fdl.model.structures.Clusters;
17 import de.kl.fdl.model.structures.Edge;
18 import de.kl.fdl.model.structures.Edges;
19 import de.kl.fdl.model.structures.Node;
20 import de.kl.fdl.model.structures.Nodes;
21 import de.kl.fdl.model.structures.RepulsionConnection;
22 import java.util.ArrayList;
23 import java.util.HashMap;
24 import java.util.Map;
25 import java.util.Map.Entry;
26 import java.util.Queue;
27
28 /**
29  * The graph model containing all elements and all edges and
       the iteration
30  * method.<p> Clusters information is stored too ( but in this
       model, clusters
31  * are just elements with children ).<p>
32  *
33  *
34  */
35 public class Model
36 {
37
38   public final static String ROOT_ELEMENT = "Root";
39   // Representation of the model
40   private Cluster root = new Cluster();
41   private Nodes allNodes = new Nodes();
```

```
42    private Edges allEdges = new Edges();
43    private Clusters allClusters = new Clusters();
44    private LinkedList<RepulsionConnection> repCons = new
          LinkedList<RepulsionConnection>();
45    //Distances
46    private HashMap<String, HashMap<String, Integer>>
          DistancesHops = new HashMap<String, HashMap<String,
          Integer>>();
47    private HashMap<String, HashMap<String, Double>>
          DistancesLengths = new HashMap<String, HashMap<String,
          Double>>();
48    private HashMap<String, HashMap<String, List<Edge>>>
          DistancesLengthsPaths = new HashMap<String,
          HashMap<String, List<Edge>>>();
49
50    public void DeleteNode(Node node)
51    {
52      this.allNodes.remove(node);
53
54      String nodeId = node.getId();
55      Edges toBeRemoved = new Edges();
56      for (Edge e : this.allEdges)
57      {
58        String Id1 = e.getElement1Id();
59        String Id2 = e.getElement2Id();
60        if (Id1.equals(nodeId) || Id2.equals(nodeId))
61        {
62          toBeRemoved.add(e);
63        }
64      }
65
66      for (Edge e : toBeRemoved)
67      {
68        this.allEdges.remove(e);
69      }
70
71      for (Cluster c : this.allClusters)
72      {
73        this.DeleteNode(node, c);
74      }
75      StructureChanged = true;
76    }
77
78    public void DeleteNode(Node node, Cluster c)
```

```
79    {
80      c.removeElement(node);
81
82      String nodeId = node.getId();
83      Edges toBeRemoved = new Edges();
84      for (Edge e : c.GetEdges())
85      {
86        String Id1 = e.getElement1Id();
87        String Id2 = e.getElement2Id();
88        if (Id1.equals(nodeId) || Id2.equals(nodeId))
89        {
90          toBeRemoved.add(e);
91        }
92      }
93      for (Edge e : toBeRemoved)
94      {
95        c.removeEdge(e);
96      }
97      StructureChanged = true;
98    }
99
100   private void MakeDistancesLengths()
101   {
102     //Distances.clear();
103     DistancesLengths = new HashMap<String, HashMap<String,
          Double>>();
104     //DistancesLengthsPaths = new HashMap<String,
          HashMap<String, List<Edge>>>();
105
106     int NumberOfNodes = this.allNodes.size();
107     //Create Structure
108     for (Integer j = 0; j < NumberOfNodes; j++)
109     {
110       DistancesLengths.put(this.allNodes.get(j).getId(), new
            HashMap<String, Double>());
111       //DistancesLengthsPaths.put(this.allNodes.get(j).getId(),
            new HashMap<String, List<Edge>>());
112       for (Integer i = 0; i < NumberOfNodes; i++)
113       {
114         if (i == j)
115         {
116           DistancesLengths.get(this.allNodes.get(j).getId())
                .put(this.allNodes.get(i).getId(), 0.0);
117         } else
```

```
118           {
119             DistancesLengths.get(this.allNodes.get(j).getId())
                    .put(this.allNodes.get(i).getId(),
                    Double.MAX_VALUE);
120           }
121         }
122       }
123
124       //Fill Table with current edges
125       for (Edge e : this.allEdges)
126       {
127         String n1 = e.getElement1Id();
128         String n2 = e.getElement2Id();
129         DistancesLengths.get(n1).put(n2, e.getDesiredLength());
130         //DistancesLengthsPaths.get(n1).get(n2).add(e);
131         if (n1.equals(n2))
132         {
133           DistancesLengths.get(n1).put(n2, 0.0);
134         }
135       }
136
137       //Calculate Distances
138       for (Integer j = 0; j < NumberOfNodes; j++)
139       {
140         for (Integer i = 0; i < NumberOfNodes; i++)
141         {
142           for (Integer k = 0; k < NumberOfNodes; k++)
143           {
144             //if(A[i][j] + A[j][k] < A[i][k]) then A[i][k] =
                    A[i][j] + A[j][k]
145             Double ij = DistancesLengths.get(this.allNodes.get(i)
                    .getId()).get(this.allNodes.get(j).getId());
146             Double jk = DistancesLengths.get(this.allNodes.get(j)
                    .getId()).get(this.allNodes.get(k).getId());
147             Double ik = DistancesLengths.get(this.allNodes.get(i)
                    .getId()).get(this.allNodes.get(k).getId());
148             Double ijk = ij + jk;
149             if (ijk > 0 && ijk < ik)
150             {
151               DistancesLengths.get(this.allNodes.get(i)
                      .getId()).put(this.allNodes.get(k).getId(), ij +
                      jk);
152             }
153           }
```

```
154         }
155       }
156       StructureChanged = false;
157     }
158
159     private void MakeDistancesHops()
160     {
161       //Distances.clear();
162       DistancesHops = new HashMap<String, HashMap<String,
              Integer>>();
163
164       int NumberOfNodes = this.allNodes.size();
165       //Create Structure
166       for (Integer j = 0; j < NumberOfNodes; j++)
167       {
168         DistancesHops.put(this.allNodes.get(j).getId(), new
              HashMap<String, Integer>());
169         for (Integer i = 0; i < NumberOfNodes; i++)
170         {
171           if (i == j)
172           {
173             DistancesHops.get(this.allNodes.get(j).getId())
                  .put(this.allNodes.get(i).getId(), 0);
174           } else
175           {
176             DistancesHops.get(this.allNodes.get(j).getId())
                  .put(this.allNodes.get(i).getId(),
                  Integer.MAX_VALUE);
177           }
178         }
179       }
180
181       //Fill Table with current edges
182       for (Edge e : this.allEdges)
183       {
184         String n1 = e.getElement1Id();
185         String n2 = e.getElement2Id();
186         DistancesHops.get(n1).put(n2, 1);
187         if (n1.equals(n2))
188         {
189           DistancesHops.get(n1).put(n2, 0);
190         }
191       }
192
```

```
193       //Calculate Distances
194       for (Integer j = 0; j < NumberOfNodes; j++)
195       {
196         for (Integer i = 0; i < NumberOfNodes; i++)
197         {
198           for (Integer k = 0; k < NumberOfNodes; k++)
199           {
200             Integer ij = DistancesHops.get(this.allNodes.get(i)
                  .getId()).get(this.allNodes.get(j).getId());
201             Integer jk = DistancesHops.get(this.allNodes.get(j)
                  .getId()).get(this.allNodes.get(k).getId());
202             Integer ik = DistancesHops.get(this.allNodes.get(i)
                  .getId()).get(this.allNodes.get(k).getId());
203             Integer ijk = ij + jk;
204             if (ijk > 0 && ijk < ik)
205             {
206               DistancesHops.get(this.allNodes.get(i).getId())
                    .put(this.allNodes.get(k).getId(), ij + jk);
207             }
208           }
209         }
210       }
211       StructureChanged = false;
212     }
213
214     public HashMap<String, HashMap<String, Integer>>
          GetDistances()
215     {
216       if (StructureChanged)
217       {
218         MakeDistancesLengths();
219         MakeDistancesHops();
220       }
221       return DistancesHops;
222     }
223
224     public HashMap<String, HashMap<String, Double>>
          GetDistancesLengths()
225     {
226       if (StructureChanged)
227       {
228         MakeDistancesLengths();
229         MakeDistancesHops();
230       }
```

```
231       return DistancesLengths;
232     }
233
234     public HashMap<String, HashMap<String, List<Edge>>>
            GetDistancesLenghtsPaths()
235     {
236       if (StructureChanged)
237       {
238         MakeDistancesLengths();
239         MakeDistancesHops();
240       }
241       MakeDistancesLengthsPaths();
242       return DistancesLengthsPaths;
243     }
244
245     public void MakeDistancesLengthsPaths()
246     {
247       DistancesLengthsPaths = new HashMap<String, HashMap<String,
              List<Edge>>>();
248
249       int NumberOfNodes = this.allNodes.size();
250       //Create Structure
251       for (Integer j = 0; j < NumberOfNodes; j++)
252       {
253         DistancesLengthsPaths.put(this.allNodes.get(j).getId(),
                new HashMap<String, List<Edge>>());
254         for (Integer i = 0; i < NumberOfNodes; i++)
255         {
256           DistancesLengthsPaths.get(this.allNodes.get(j).getId())
                .put(this.allNodes.get(i).getId(),
                MakeDijkstra(this.allNodes.get(j),
                this.allNodes.get(i)));
257         }
258       }
259     }
260
261     private List<Edge> MakeDijkstra(Node Start, Node Target)
262     {
263       List<Edge> path =  new LinkedList<Edge>();
264       HashMap<String, Boolean> visited = new HashMap<>();
265       HashMap<String, Node> preceders = new HashMap<>();
266       HashMap<String, Double> distances = new HashMap<>();
267       for (Node n : this.allNodes)
268       {
```

```
269         visited.put(n.getId(), false);
270         preceders.put(n.getId(), null);
271         distances.put(n.getId(), Double.MAX_VALUE);
272       }
273       distances.put(Start.getId(), 0.0);
274       List<Node> q = new LinkedList<>();
275       q.add(Start);
276
277       while (!q.isEmpty())
278       {
279         Node u = this.allNodes.get(getMinimumVisited(distances,
              visited));
280         if (u.equals(Target))
281         {
282           break;
283         }
284         q.remove(u);
285         visited.put(u.getId(),true);
286
287         ArrayList<Node> Neighbors = this.GetNeighbors(u);
288         for (Node v : Neighbors)
289         {
290           double alt =  distances.get(u.getId()) +
                allEdges.get(u, v).getDesiredLength();
291           if (alt < distances.get(v.getId()))
292           {
293             distances.put(v.getId(), alt);
294             preceders.put(v.getId(), u);
295             if (!visited.get(v.getId()))
296             {
297               q.add(v);
298             }
299           }
300         }
301       }
302
303       List<Node> pathNodes = new LinkedList<>();
304       Node u = Target;
305       while (preceders.get(u.getId()) != null)
306       {
307         pathNodes.add(u);
308         u = preceders.get(u.getId());
309       }
310       pathNodes.add(Start);
```

```
311
312       for (int i = pathNodes.size(); i > 1; i--)
313       {
314         Node n1 = pathNodes.get(i-1);
315         Node n2 = pathNodes.get(i-2);
316         path.add(this.allEdges.get(n1,n2));
317       }
318
319       return path;
320     }
321
322     private String getMinimumVisited(HashMap<String, Double> map,
          HashMap<String, Boolean> visited )
323     {
324       Double distance = Double.MAX_VALUE;
325       String returnKey = "";
326       for (String key : map.keySet())
327       {
328         Double distanceTemp = map.get(key);
329         if (distanceTemp < distance && visited.get(key) == false)
330         {
331           distance = distanceTemp;
332           returnKey = key;
333         }
334       }
335       return returnKey;
336     }
337
338
339     public ArrayList<Node> GetNeighbors(Node n)
340     {
341       ArrayList<Node> ret = new ArrayList<Node>();
342
343       for (Edge e : this.allEdges)
344       {
345         if (e.getElement1Id().equals(n.getId()))
346         {
347           if (!ret.contains((Node) e.getElement2()))
348           {
349             ret.add((Node) e.getElement2());
350           }
351         }
352         if (e.getElement2Id().equals(n.getId()))
353         {
```

```
354             if (!ret.contains((Node) e.getElement1()))
355             {
356               ret.add((Node) e.getElement1());
357             }
358           }
359         }
360
361         return ret;
362       }
363
364       public int GetMaxHits()
365       {
366         int max = 0;
367
368         for (Node n:this.allNodes)
369         {
370           Integer numberakt = n.getHits();
371           if (max < numberakt)
372           {
373               max = numberakt;
374           }
375         }
376         return max;
377       }
378   }
```

Listing 5.10: Methode zum Markieren aller Knoten, bei denen der Funktionswert 0 ist.

```
1  private void MarkNodesWith0HitsForDeletion() {
2    Model m = _parent.GetModel();
3    if (m == null) {
4      System.err.println("Model was NULL!");
5      return;
6    }
7
8    for (Node node : m.getAllNodes()) {
9      if (!_nodesToBeDeleted.contains(node)) {
10       if (node.getHits() == 0) {
11         System.err.println("Marking for Deletion: " +
               node.getId());
12         _nodesToBeDeleted.add(node);
13       }
14     } else {
15       System.err.println("Already marked: " + node.getId());
16     }
17   }
18 }
```

Abbildungsverzeichnis

1.1 Knotenfarben und -positionen des in der Arbeit verwendeten, einfachen Beispiels. . . . 2
1.2 Das gleiche Beispiel in FDLView. . . . 3

3.1 Knotenliste des Beispiels in FDLView . . . 9
3.2 Kantenliste des Beispiels in FDLView . . . 10
3.3 Aus Listing 3.2 erzeugter Graph mit importierten Funktionswerten. . 11
3.4 Im Beispielgitter wurde Knoten 7 entfernt. . . . 12
3.5 Graph, wenn bei Global-Lokal erst Knoten 3 und dann Knoten 6 entfernt werden. . . . 16
3.6 Graph, wenn bei Global-Lokal erst Knoten 6 und dann Knoten 3 entfernt werden. . . . 16
3.7 Graph, wenn bei Global-Lokal die Kanten erst nach jedem Grad entfernt werden. . . . 16
3.8 Erzeugte Kanten, wenn neue Kanten niemals in die Berechnung einfließen. . . . 19
3.9 Erzeugte Kanten, wenn neue Kanten nach Entfernung gruppiert in die Berechnung einfließen. . . . 19
3.10 Graph, wenn bei Global-Lokal die Knoten 4, 6, 7 und 8 entfernt und dann die Knoten aufsteigend geordnet verarbeitet werden. . . . 21
3.11 Graph, wenn bei Global-Lokal die Knoten 4, 6, 7 und 8 entfernt und dann die Knoten absteigend geordnet verarbeitet werden. . . . 21

4.1 Screenshot der UI zum Entfernen von Knoten . . . 25
4.2 Beispielgraph nach Layout in 3D. . . . 27
4.3 Beispielgraph nach Layout in 3D und Umschalten der Verarbeitung auf 2D. . . . 27
4.4 Beispielgraph nach Layout in 2D (Aufsicht). . . . 28
4.5 Beispielgraph nach Layout in 2D (Ansicht von schräg). . . . 28
4.6 Screenshot der UI zum Einstellen der Visualisierungsoptionen . . . 30

Quellcodeverzeichnis

3.1 CSV-Datei, aus der das in dieser Arbeit verwendete Beispiel erzeugt wird. 8
3.2 Beispielhafte Datei mit Funktionswerten. 8
3.3 Dijkstra-Algorithmus in Pseudocode 20

5.1 Implementierung der Kantenlaenge 33
5.2 Implementierung der Laengenberechnung mit SWS 34
5.3 Implementierung der Winkelberechnung 35
5.4 UnitTests zur Winkelberechnung 37
5.5 Knotenentfernung Lokal-Lokal 40
5.6 Knotenentfernung Global-Global 42
5.7 Iterate-Methode in Algorithmen.java 46
5.8 Relevante Teile der Klasse Node.java 48
5.9 Relevante Teile der Klasse Model.java 50
5.10 Methode zum Markieren aller Knoten, bei denen der Funktionswert 0 ist. 60

Literaturverzeichnis

[Bodendiek und Lang 1995] Bodendiek, Rainer ; Lang, Rainer: *Lehrbuch der Graphentheorie Band 1.* Spektrum, 1995

[Comers 2003] Comers, Douglas: *Computernetzwerke und Internets.* Pearson Studium, 2003

[Foulds 1992] Foulds, L.R.: *Graph Theory Applications.* Springer-Verlag, 1992

[Oracle 2014] Oracle: *What's New in JDK 8.* März 2014. – URL `http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`

[Quasthoff 2009] Quasthoff, Uwe: *Vorlesung Algorithmen und Datenstrukturen 2.* Mai 2009

[Rudolph 2013] Rudolph, Dennis: *Schnittwinkel zweier Geraden.* Dezember 2013. – URL `http://www.frustfrei-lernen.de/mathematik/schnittwinkel-zweier-geraden.html`

[Wikipedia 2014] Wikipedia: *Dijkstra-Algorithmus.* März 2014. – URL `https://de.wikipedia.org/w/index.php?title=Dijkstra-Algorithmus&oldid=127308746`

[Wright 1999] Wright, Robert: *IP-Routing-Grundlagen.* Markt und Technik, Buch- und Software-Verlag, 1999

[Zeckzer u. a. 2008] Zeckzer, Dirk ; Schröder, Leon ; Kalcklösch, Robert ; Hagen, Hans ; Klein, Timo: *Analyzing the reliability of communication between software entities using a 3D visualization of clustered graphs.* In Proceedings of the 4th ACM Symposium on Software Visualization (Ammersee, Germany, September 16 - 17, 2008). SoftVis '08. pp. 37-46, ACM, New York, 2008

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann

Leipzig, den 25. März 2014 Rico Feist