Visualisierungsinstitut der Universität Stuttgart
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3330

Interaktive Visualisierung dynamischer gerichteter gewichteter Graphen mit partiellen Kanten

Xiaobo Gan

Studiengang:	Informatik
Prüfer:	Prof. Dr. Daniel Weiskopf
Betreuer:	Dr. rer. nat. Michael Burch
begonnen am:	07. Mai 2012
beendet am:	06. Dezember 2012
CR-Klassifikation:	D.1.5, H.5.2, I.3.3, I.3.4, I.3.6

Kurzfassung

Die Visualisierung dynamischer gerichteter und gewichteter Graphen ist eine herausfordernde Aufgabe. Animierte Diagramme führen zu hohen kognitiven Lasten und gute Layoutalgorithmen werden verlangt, um die sogenannte dynamische Stabilität und Mental Maps aufrechtzuerhalten. Statische Diagramme hingegen verwenden Small Multiples-Visualisierungen und einzelne Graphen haben deshalb weniger Platz zur Darstellung. In dieser Arbeit integrieren wir dynamische Katengewichte in ein Knoten-Kanten Diagramm, indem alle Kanten mit einer Zeitachse versehen werden und der Gewichtsverlauf somit immer vom Start- zum Zielknoten intuitiv über ein geeignetes Farbschema verstanden werden kann. Visual Clutter, der typischerweise wegen vielen Kantenkreuzungen auftritt, wird durch die Repräsentation mit partiell gezeichneten Kanten reduziert.

Abstract

Visualizing dynamic directed and weighted graph is a challenging task. Animated diagrams typically lead to high cognitive load, and good layout algorithms are required with the goal to have dynamic stability and to preserve a viewer's mental map. Static diagrams on the other side use small multiples representation, which requires more display space. In this thesis we integrate dynamic edge weights in a single node-link diagram by attaching a time line to all edges. By this the evolution of weights can be understand in an intuitive manner by following an edge from the start to the target vertex. The problem of visual clutter is reduced by using partially drawn links.

Inhaltsverzeichnis

1 Einleitung **11**
- 1.1 Motivation . . . 11
- 1.2 Aufgabenstellung . . . 11
- 1.3 Vorgehensweise und Realisierung . . . 12
- 1.4 Gliederung der Arbeit . . . 12

2 Grundlagen **15**
- 2.1 Einführung in die Informationsvisualisierung . . . 15
  - 2.1.1 Definition von Informationsvisualisierung . . . 15
  - 2.1.2 Aufgabenstellung der Visualisierung . . . 16
  - 2.1.3 Datentypen . . . 16
  - 2.1.4 Konkrete Visualisierungstechniken . . . 18
  - 2.1.5 Node-Link Techniken . . . 18
- 2.2 Partiell gezeichnete Kanten . . . 19
- 2.3 Darstellung von Kanten . . . 21
- 2.4 Dynamische Graphvisualisierung . . . 21

3 Entwurf **23**
- 3.1 Anforderungen . . . 23
  - 3.1.1 Einlesen von dynamischen Graphdaten . . . 23
  - 3.1.2 Graphische Darstellung von Knoten und Kanten . . . 26
  - 3.1.3 Interaktion . . . 28
  - 3.1.4 Graphische Darstellung des Kantengewichtes . . . 28
  - 3.1.5 Informationsanzeige . . . 29
- 3.2 Entwurfsmodell . . . 30
  - 3.2.1 Datenmodell . . . 30
  - 3.2.2 Referenzmodell . . . 30
- 3.3 Beschreibung der Funktionalitäten . . . 31
  - 3.3.1 Dateienoperation . . . 31
  - 3.3.2 Visualisierung . . . 32
  - 3.3.3 Steuerung . . . 32

4 Implementierung **33**
- 4.1 Auswahl der Entwicklungsumgebung . . . 33
- 4.2 Implementierung der Funktionsblöcke . . . 33
  - 4.2.1 Framework . . . 34
  - 4.2.2 Dateienoperationen . . . 34

4.2.3 Canvas 38
4.2.4 Bedienfeld 44
4.2.5 Knotenlayout 45

5 Fallstudie **51**

6 Zusammenfassung und Ausblick **59**
6.1 Zusammenfassung 59
6.2 Ausblick 59

Literaturverzeichnis **61**

Abbildungsverzeichnis

2.1 Datenstruktur . . . 17
2.2 Radiales Layout . . . 19
2.3 Klare Darstellung . . . 20
2.4 Unmengen von Kanten . . . 20
2.5 Kantendarstellung . . . 21
2.6 Kante mit 4 Gewichten . . . 21

3.1 Knotendatei . . . 24
3.2 Kantendatei . . . 25
3.3 Kantendatei mit drei Gewichten . . . 25
3.4 BGraph-Datei . . . 26
3.5 Knotendarstellung . . . 27
3.6 Kantendarstellung . . . 27
3.7 Visuelle Artefakte . . . 28
3.8 Kanten mit 4 Gewichten . . . 29
3.9 Datenmodell . . . 30
3.10 Referenzmodell . . . 31

4.1 Grapheditor . . . 34
4.2 Menü . . . 35
4.3 fileChooser . . . 36
4.4 Knoten und Kante . . . 40
4.5 Kante mit mehreren Gewichten . . . 41
4.6 Farbverlauf in den Kanten . . . 42
4.7 Farbverlauf in den Kanten: mit Logarithmus . . . 43
4.8 Bedienfeld des Grapheditors . . . 44
4.9 Zufällige Verteilung von 12 Knoten . . . 47
4.10 Ebene Polarkoordinaten und ihre Transformation in kartesische Koordinaten . . . 48
4.11 Radiale Verteilung von 12 Knoten . . . 49

5.1 Testdatei: Migration.bgraph . . . 51
5.2 Zufällige Verteilung mit Label . . . 52
5.3 Zufällige Verteilung ohne Label . . . 53
5.4 Radiale Verteilung: 50% der Kantenlänge . . . 54
5.5 Radiale Verteilung: 75% der Kantenlänge . . . 54
5.6 Ziehung der Knoten . . . 55
5.7 Ziehung der Knoten: mit Label . . . 55

5.8 Migration von sechs Ländern: zufällige Verteilung mit vollständiger Kantenlänge 56
5.9 Migration von sechs Ländern: radiale Verteilung mit vollständiger Kantenlänge 57
5.10 Migration von sechs Ländern: radiale Verteilung mit 37% Kantenlänge 57

Verzeichnis der Listings

4.1 Erzeugung eines Frameworks 34
4.2 Hinzufügen der Menüelemente 35
4.3 Behandlung des ActionListeners 36
4.4 Die Funktion openLabel 37
4.5 Einlesen der Knoten- und Kantendatei 37
4.6 Schleife zum Knoten einlesen 37
4.7 Schleife zum Kanten einlesen 38
4.8 Verknüpfung der Listen und dem dynamischen Graph 39
4.9 Funktion: Paint 39
4.10 Anti-Aliasing 39
4.11 Pseudo-Code im Stil von JAVA: Zeichenprozess 40
4.12 Funktion: addComponent 45
4.13 Neu definierte Funktionen 46
4.14 Einfügen des Bedienfeldes 46
4.15 Zufällige Verteilung 47
4.16 Radiale Verteilung 49

1 Einleitung

In diesem Kapitel werden zuerst die Motivation und die Aufgabenstellung vorgestellt, und dann die Vorgehensweise in der Arbeit und die Realisierung der gegebenen Aufgaben geklärt. Schließlich wird die Gliederung der Arbeit vorgestellt.

1.1 Motivation

Die heutigen Visualisierungstechniken von Graphen versuchen, eine möglichst ästhetische Darstellung zu generieren. Dabei können relationale Daten mathematisch als Graphen modelliert werden und visuell als Knoten-Kanten Diagramm dargestellt werden. Aber zahlreiche Kantenkreuzungen im Diagramm können oftmals zu Visual Clutter führen. Es existieren bereits spezielle Graphlayoutalgorithmen, um dieses Problem etwas zu entschärfen. Ein weiterer Ansatz wäre es, die Kanten nur partiell zu zeichnen, wodurch das Problem des Visual Clutters etwas reduziert wird. Als negative Folge fällt es aber auch schwerer, den Kanten zu folgen und den Endknoten einer Kante zu finden, was teilweise zu Mehrdeutigkeiten führt. Um die Lesbarkeit von partiell gezeichneten Kanten zu untersuchen, soll ein geeignetes interaktives Graphvisualisierungswerkzeug entwickelt werden. Dieses Graphvisualisierungswerkzeug kann auch dazu dienen, die Visualisierung dynamischer gerichteter und gewichteter Relationen zu realisieren, indem man die repräsentierten partiellen Kanten in entsprechend viele farbkodierte Teilsegmente aufspaltet.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist der Entwurf und die Implementierung eines Grafikeditors, der nicht nur die relationalen Dateien als Knoten-Kanten darstellt, sondern auch die Interaktion mit dem Benutzer unterstützt, um die Graphik nach Bedarf anzupassen. Er soll in der Lage sein, mit einem Bedienfeld mit dem Benutzer zu kommunizieren, die dynamischen Graphdaten aus Textdateien einzulesen und dann die Grafik im Hauptteil des Editors anzuzeigen.

Die Aufgabenstellung lässt sich daher in vier Hauptteile gliedern:

1. Das Einlesen von dynamischen Graphdaten: Es geht nicht nur um die Kantendatei mit einem einzelnen Kantengewicht, sondern auch um die Kantendatei mit mehreren Kantengewichten, also um einen dynamischen Graphen. Der Editor soll in der Lage sein, beide Formate zu behandeln. Das dient der Eingabe des Editors und arbeitet als Thread im Hintergrund.

2. Die graphische Darstellung von Knoten und gerichteten partiellen Kanten mit benutzerdefinierter Länge: Statt eines Punktes wird ein Kreis als visuelle Darstellung für einen Knoten gezeichnet und die gerichtete Kante wird durch ein nadelförmiges spitz zulaufendes Dreieck repräsentiert, der sogenannte Tapered Link Ansatz [HW09]. Das ist die Hauptaufgabe des Editors und dies wird dem Betrachter in der Mittelansicht des Editors angezeigt.
3. Die Interaktion zwischen Benutzer und Editor: Um die Analyse der Daten zu unterstützen, stellt der Editor eine Serie von Funktionalitäten zur Darstellung von Graphen zur Verfügung. Zum Beispiel soll die Länge der Kanten schneller angepasst werden. Je mehr Funktionalitäten es gibt, desto schneller und bequemer kann eine Datenanalyse von statten gehen, es müssen allerdings auch mehr Features gelernt werden. Die am meisten verwendeten Funktionalitäten werden im Bedienfeld auf der linken Seite des Editors angezeigt und die anderen befinden sich im entsprechenden Menü.
4. Die graphische Darstellung des Kantengewichtes: Die partiell gezeichneten Kanten sollen in farbkodierte Teilstücke untergliedert werden, um die dynamischen Kantengewichte darzustellen. Das heißt, die Farbe jedes Teilstückes ist abhängig vom Kantengewicht.

1.3 Vorgehensweise und Realisierung

Die Entwicklung des Grapheditors wird nach einem inkrementellen Modell durchgeführt. Zuerst werden die grundlegenden Funktionalitäten implementiert und danach werden die weiteren Funktionalitäten hinzugefügt, um die Anforderungen zu erfüllen. Das heißt, am Anfang wird nur ein einfacher Grapheditor aufgebaut, der nur die Kantendatei mit einzelnem Kantengewicht einlesen kann und simple kleine Graphen mit wenigen Knoten und Kanten visualisieren kann. Danach werden die Steuerungsfunktionen implementiert, z.B. Reaktionen auf Mausaktionen vom Benutzer, um Interaktivität zu erlauben. Dadurch können die Darstellungen und Layouts von Knoten-Kanten Diagrammen vom Benutzer beeinflusst und gesteuert werden. Danach wird der Grapheditor so erweitert, dass die Kantendatei mit mehreren Kantengewichten eingelesen werden kann und im Grapheditor dargestellt wird. Am Anfang wird nur die zufällige Verteilung von Knoten realisiert. Schließlich wird die radiale oder zirkuläre Verteilung von Knoten ebenfalls unterstützt. Solche radiale Darstellungen eignen sich besser, um mit partiell gezeichneten Kanten zu arbeiten, da Mehrdeutigkeiten bei Zielknoten reduziert werden.

1.4 Gliederung der Arbeit

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: In diesem Kapitel werden alle grundlegenden Technologien sowie die theoretischen Grundlagen erörtert, soweit sie für das Verständnis dieser Arbeit notwendig sind. Dazu gehören Begriffe wie Informationsvisualisierung, Node-Link-Techniken, partielle Links und dynamische Graphvisualisierung. Des Weiteren werden weiterführende Begriffe und Definitionen aus dem Problemumfeld aufgeführt.

Kapitel 3 – Entwurf: In diesem Kapitel werden alle Details über den Entwurf dargestellt. Am Anfang werden die Anforderungen analysiert und danach ein passendes Entwicklungsmodell ausgewählt. Die Hauptfunktionalitäten werden hier vorgegeben und die Ideen, um jedes Ziel zu erreichen, dargelegt.

Kapitel 4 – Implementierung: In diesem Kapitel werden alle Details über das Programm umfasst. Es wird erläutert, wie der Editor aussieht, d.h. wie er in die drei elementaren Bauteile getrennt programmiert wird und diese Teile dann untereinander interagieren.

Kapitel 5 – Fallstudie: In diesem Kapitel wird der Grapheditor anhand eines praktischen Beispiels vorgestellt.

Kapitel 6 – Zusammenfassung und Ausblick: Die gesamte Arbeit wird hier zusammengefasst und es wird ein Ausblick auf die Zukunft vorgestellt. Hier wird beschrieben, in welche Richtung der Editor weiter verbessert und erweitert werden kann.

2 Grundlagen

Im diesem Kapitel werden wichtige, zum Verständnis der Arbeit notwendige Grundlagen vermittelt. Zuerst werden ein paar Theorien über Informationsvisualisierung erläutert. Anschließend wird die Darstellung von Kanten im Allgemeinen und auch speziell zu partiell gekennzeichneten Kanten vorgestellt. Das Konzept der Dynamischen Graphvisualisierung wird am Ende des Kapitels vorgestellt.

2.1 Einführung in die Informationsvisualisierung

„Mehr als 60% der Informationen, die das menschliche Gehirn weiter verarbeitet, werden über die Augen aufgenommen“ [Zep04]. Deswegen spielt die Visualisierung eine wichtige Rolle bei der Informationsvermittlung. Dieses Gebiet hat sich seit Mitte der 90er-Jahre als eigenständiges Forschungsgebiet entwickelt. In den letzten Jahren wurden dafür verschiedene spezielle Visualisierungstechniken entwickelt, die den heutigen gestiegenen Anforderungen der Anwendungsgebiete entsprechen.

2.1.1 Definition von Informationsvisualisierung

Eine Definition der Visualisierung von Stephan Diehl ist wie folgt [PD10, Seite 440]:

> Visualisierung ist der Prozess, der die Transformation von Daten in eine visuelle Darstellung gestattet, um somit verborgene Aspekte in den Daten zu entdecken, die für die Exploration und die Analyse wesentlich sind.

Normalerweise versteht man die klassische Visualisierung als die wissenschaftliche Visualisierung. Hingegen ist die Informationsvisualisierung ein relativ jüngerer Zweig von Visualisierung. Der grundsätzliche Unterschied zwischen Informationsvisualisierung und wissenschaftlicher Visualisierung liegt an ihren Datengrundlagen. Bei der wissenschaftlichen Visualisierung sind die Daten numerische Messwerte, die in einem Gitter positioniert werden können, und zwar meist physikalischer Natur. Im Gegensatz dazu können die Daten für Informationsvisualisierung neben numerischen Werten alle möglichen abstrakten Datentypen sein, die nicht unmittelbar mit physikalischen Zuständen oder Vorgängen verknüpft sind. Eine genauere Definition ist wie folgt gegeben [PD10, Seite 440]:

> Informationsvisualisierung ist die Nutzung computergenerierter, interaktiver, visueller Repräsentationen von abstrakten, nicht-physikalischen Daten zur Verstärkung des Erkenntnisgewinns.

2.1.2 Aufgabenstellung der Visualisierung

Die Aufgabenstellung der Informationsvisualisierung ist es, abstrakte Daten graphisch so zu repräsentieren, dass strukturelle Zusammenhänge und relevante Eigenschaften der Daten intuitiv erfasst werden können. Damit können die in den Daten enthaltenen Informationen möglichst schnell und vollständig erkannt werden. Dieses prinzipielle Vorgehen schließt zwei Aspekte ein:

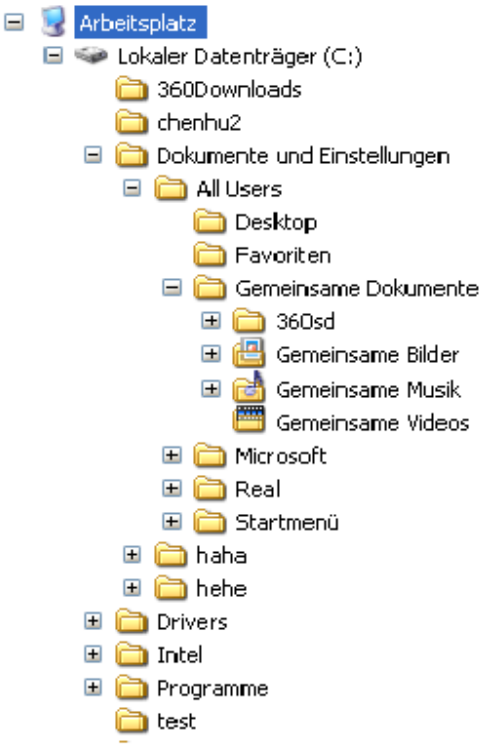
1. Expressive Darstellung
   Alle Daten und die in den Daten enthaltenen Informationen sollen mit Visualisierungstechniken graphisch dargestellt werden.
2. Effektive Darstellung
   Die Visualisierungsdaten sollen vom Betrachter möglichst schnell und deutlich wahrgenommen werden.

2.1.3 Datentypen

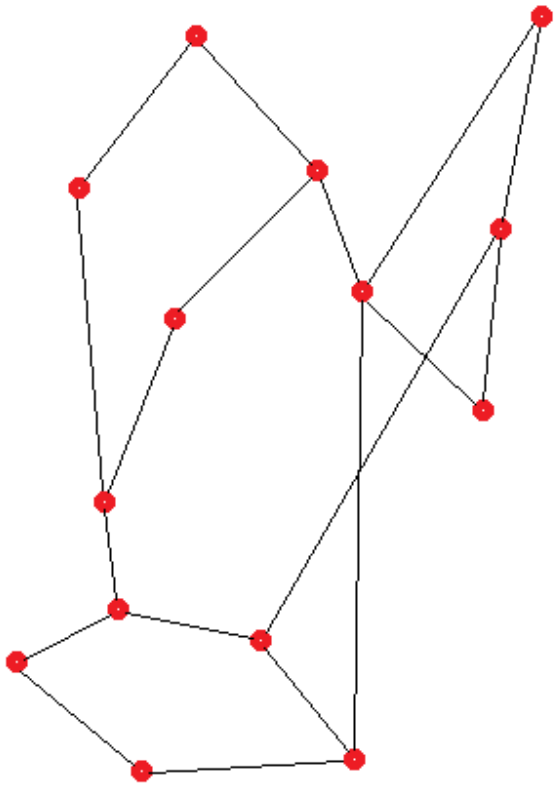
Wie im vorausgegangen Abschnitt erwähnt, handelt es sich bei der Informationsvisualisierung hauptsächlich um die Darstellung abstrakter Datentypen. In diesem Abschnitt soll deshalb näher auf diese Datentypen eingegangen werden, die in den meisten Datensätzen in der Informationsvisualisierung vorkommen. Neben den primitiven Datentypen wie etwa quantitative, ordinale oder kategorische werden hier Beispiele für bereits komplexere Datentypen gegeben:

1. Eindimensionale Daten (z.B. zeitabhängige Daten)
2. Zweidimensionale Daten (zwei ausgewählte Attribute eines Produktes)
3. Multidimensionale Daten (mehr als zwei Attribute)
4. Relationen: Hierarchien (Bäume)
5. Relationen: Netzwerke (Graphen)
6. Text und Hypertext
7. Programmcode und Softwaresysteme

Die oben genannten Datentypen, nämlich Graphen und Bäume zählen zu den relationalen Daten. Sie beschreiben beispielsweise Dateisysteme (Baumstruktur), Netzwerke (Graph) und Vererbungshierarchien (hierarchischer Graph) (siehe Abbildung 2.1).



(a) Baum



(b) Graph

Abbildung 2.1: Datenstruktur: (a) Baum vs. (b) Graph

2.1.4 Konkrete Visualisierungstechniken

Zur expressiven und effektiven Visualisierung werden verschiedene konkrete Techniken eingesetzt. Sie werden hauptsächlich in drei Kategorien eingeteilt, von denen jede verschiedene Techniken enthält:

1. *Visualisierung mehrdimensionaler Daten*
    - Geometrische Techniken
    - Ikonische Techniken
    - Pixelbasierte Techniken
2. *Hierarchievisualisierungen*
    - Einfache Einrückungen
    - Node-Link-Diagramme
    - Flächenfüllende Verschachtelung
    - Stapelungs-basierte Ansätze
3. *Netzwerkvisualisierungen*
    - Node-Link-Techniken
    - Matrixvisualisierung

Diese unterschiedlichen Techniken spielen eine wichtige Rolle bei der Informationsvisualisierung. Für konkrete Visualisierungsaufgaben sollen die entsprechenden Techniken gewählt und eingesetzt werden. Dadurch können die strukturellen Zusammenhänge und relevanten Eigenschaften der Daten effektiv und expressiv dargestellt werden.

2.1.5 Node-Link Techniken

Node-Link Techniken sind die populärsten Visualisierungsmethoden. Mit diesen Methoden lassen sich Graphen visualisieren, also eine Menge von Knoten und Kanten, wobei jede Kante eine Verbindung zwischen zwei Knoten in dieser Menge darstellt. Die Kanten in Graphen sind häufig gerichtet und gewichtet, etwa im Falle von Netzwerken. Der Vorteil von Node-Link Techniken liegt daran, dass die Relationen explizit visualisiert werden können und somit vom Betrachter auch gut wahrgenommen werden können. Aber irrelevante Knoten könnten etwa durch ein schlechtes Layoutverfahren auf einer Kante platziert werden, was einen schlechten Einfluss auf den Betrachter hat und sogar zu einer falschen Wahrnehmung und Misinterpretationen führt.

„Radiale Techniken bieten dafür eine elegante Lösung, indem sie die Knoten eines Netzwerkes entlang eines größeren Kreises platzieren und im Inneren des Kreises Liniensegmente

zwischen einzelnen Knoten gezeichnet werden“ [PD10]. Abbildung 2.2 zeigt ein solches Beispiel für ein radiales Layout.

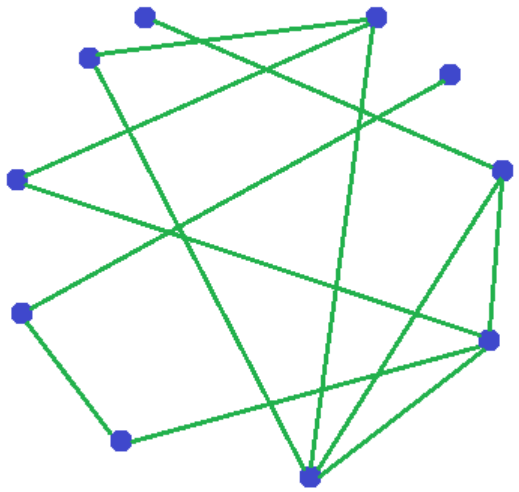


Abbildung 2.2: Radiales Layout: Alle Knoten werden auf dem Kreis platziert.

Außerdem entstehen viele Kantenkreuzungen, wenn die Anzahl der Knoten bzw. der Kanten groß ist. Je mehr Kanten und Knoten auf einem Graph dargestellt werden, desto mehr Kreuzungen werden dadurch erzeugt, was der Wahrnehmung der Informationen schadet. Um eine verbesserte Darstellung zu bekommen, werden dazu entsprechende Lösungen im nachfolgenden Abschnitt vorgestellt.

2.2 Partiell gezeichnete Kanten

Becker et al. hat eine Methode in [BBE+95] vorgestellt, um Visual Clutter zu reduzieren. Die Kanten werden nicht komplett, sondern partiell gezeichnet. Dann werden Kantenkreuzungen im Diagramm deutlich reduziert, aber gleichzeitig die Mehrdeutigkeit erhöht. Aber die Optimierung der Kantenlänge kann die Lesbarkeit verbessern. Zum Beispiel gibt Abbildung 2.3 einen deutlichen Eindruck für dieses Phänomen, wenn man sie mit Abbildung 2.4 vergleicht. Eine weitere Untersuchung von Burch et al. [BVKW11] zeigt, dass die partiell gezeichneten Kanten zu einer effektiveren Visualisierung führen und die optimale Länge der Kante abhängig von der gegebenen Aufgabe ist.

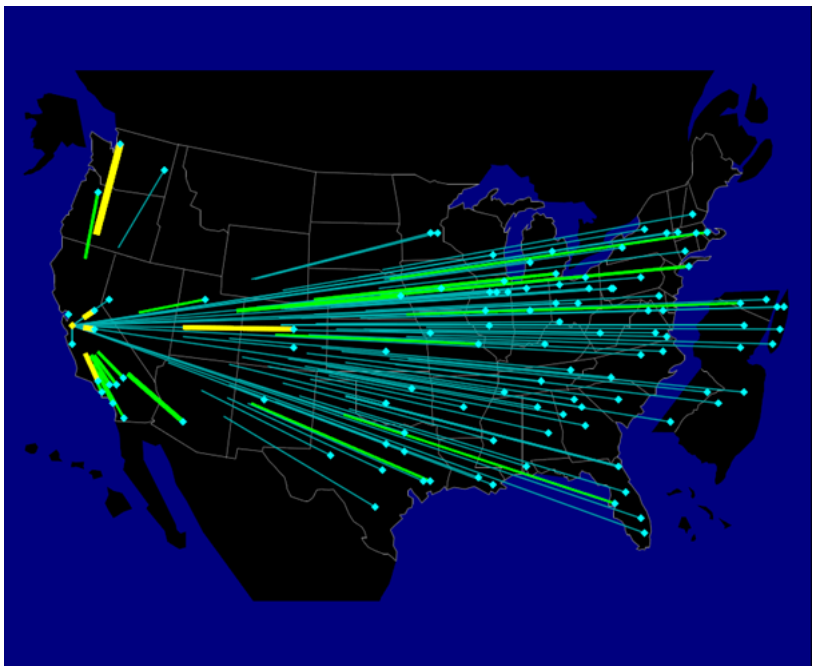


Abbildung 2.3: Klare Darstellung [BBE+95]: Die Kanten sind zwar teilweise gezeichnet aber bieten eine klare Darstellung des gesamten Graphs.

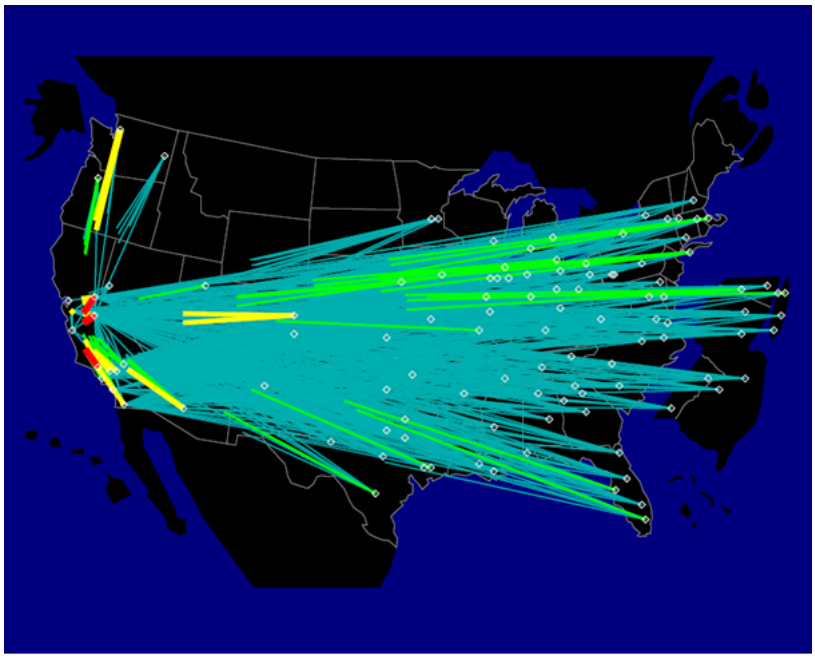
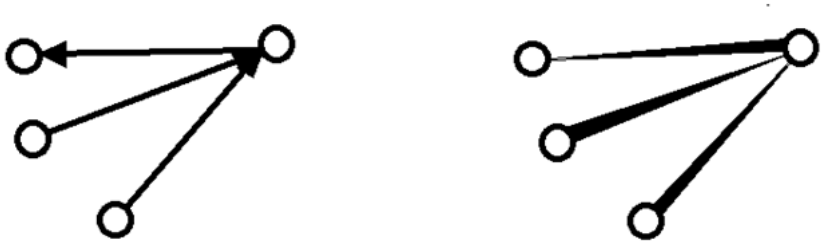


Abbildung 2.4: Unmengen von Kanten [BBE+95]: Vollständige Kanten führen oft zum Stau, wenn die Anzahl der Kanten groß ist.

2.3 Darstellung von Kanten

Graphen werden oft als Knoten-Kanten Diagramme visualisiert. Dabei werden die ungerichteten Kanten oft mit einfachen geraden Linien dargestellt. Für die Darstellung der gerichteten Kanten werden einfach gerade Linien mit Pfeilspitzen verwendet. Diese Darstellung ist zwar intuitiv für die meisten Menschen, es kann aber zu Problemen bei Knoten mit hohen Graden kommen, also bei Knoten, die viele eingehende und ausgehende Kanten haben.

In dieser Arbeit werden nur die gerichteten Kanten im Knoten-Kanten Diagramm dargestellt. In [HW09] hat eine Untersuchung gezeigt, dass sich bei Tapered und Animated Edges Verbindungen sehr schnell erkennen lassen (siehe Abbildung 2.5).



(a) Straight-Link-Edges (b) Tapered Edges

Abbildung 2.5: Kantendarstellung: (a) Straight-Link-Edges vs. (b) Tapered Edges

2.4 Dynamische Graphvisualisierung

Jede Kante besitzt ein Gewicht zu einem bestimmten Zeitpunkt. Wenn die Kantengewichte mehrerer Zeitpunkte auf einem Graph dargestellt werden, sollen die Änderungen der Kantengewichte auch visuell gezeigt werden. Dabei spricht man von Dynamischer Graphvisualisierung. Abbildung 2.6 zeigt beispielsweise eine Kante mit vier Kantengewichten.

Abbildung 2.6: Kante mit 4 Gewichten: Die Kante wird gleichmäßig in vier Abschnitten unterteilt. Jeder Teilkante wird mit der Farbe gezeichnet, die vom entsprechenden Kantengewicht abhängig ist.

3 Entwurf

In diesem Kapitel wird das Konzept des Grapheditors beschrieben. Es werden erst die Anforderungen an den Editor erläutert und dann werden Ideen und Lösungen für diese Anforderungen ausgegeben. Am Ende werden die resultierenden Funktionalitäten vorgestellt.

3.1 Anforderungen

Wie in der Einleitung bereits angedeutet wurde, soll der Grapheditor für die primären Funktionalitäten, die als fundamentale Anforderungen des Editors gelten, realisiert werden. Diese sind das Einlesen von dynamischen Graphdaten, die graphische Darstellung von Knoten und gerichteten partiellen Kanten mit benutzerdefinierter Länge und die graphische Darstellung des Kantengewichtes.

3.1.1 Einlesen von dynamischen Graphdaten

Die erste Anforderung ist das Einlesen von dynamischen Graphdaten, die in Form von Textdateien vorliegen. Alle Informationen über die Relationen von Konten und Kanten werden in zwei verschiedene Textdateien getrennt gespeichert: die eine zeigt, wie die Bezeichnung für jeden Knoten aussieht, die andere beschreibt die Verbindungen zwischen Knoten mit entsprechendem Gewicht. Vereinfacht gesagt dient die eine Datei für Knotenbeschreibungen und die andere für Kantenbeschreibungen. Das Format von Knotendateien ist festgelegt und sieht wie in Abbildung 3.1 aus.

Jede Zeile zeigt einen Konten: die erste Spalte (1) entspricht der ID eines Knotens und die zweite (2) zeigt die entsprechende Beschreibung. Alle Knoten werden mit einer eindeutigen ID in der Datei gekennzeichnet. Eine Duplikation von Knoten darf hier nicht vorkommen und sollte durch die gegebenen Daten ausgeschlossen werden. Das Format der Kantendatei ist ähnlich wie das der Knotendatei und zeigt jede Kante in einer einzelnen Zeile. Eine Kantendatei, die nur ein Kantengewicht enthält, ist in Abbildung 3.2 dargestellt.

Die erste Spalte (1) zeigt den Quellknoten mit der entsprechenden ID und die zweite Spalte (2) zeigt den Zielknoten mit der ID. Die dritte Spalte (3) zeigt das entsprechende Kantengewicht. Jede Kombination aus Startknoten und Endknoten ist einzigartig und eindeutig. Weil alle Kanten gerichtet sind, ist die Kante von A nach B nicht mit der Kante von B nach A identisch.

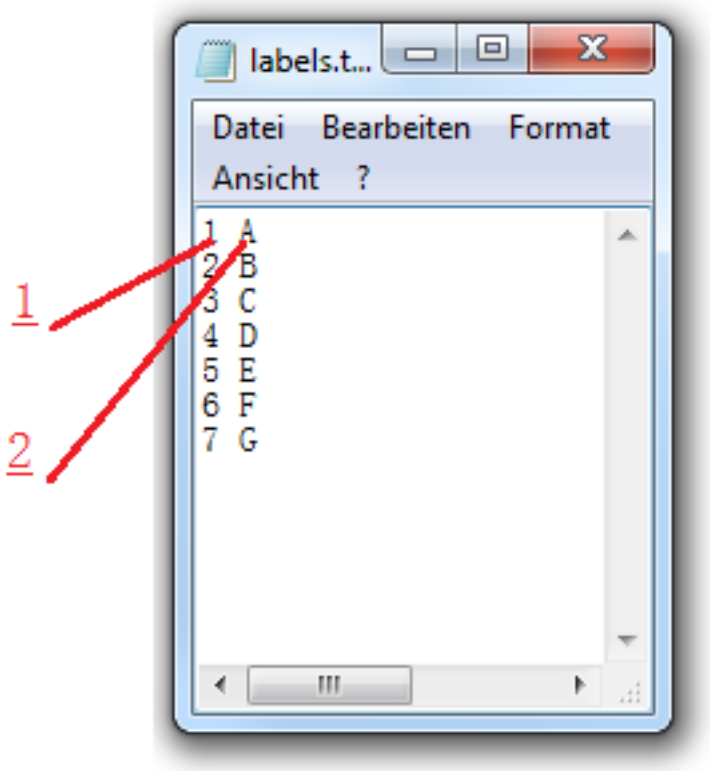


Abbildung 3.1: Knotendatei: Knoten A bis G mit 1 bis 7 indiziert in einem zeilenbasierten Textformat

Eine erweiterte Kantendatei enthält nicht nur ein Gewicht, sondern eine Liste mit Gewichten. Dann enthalten alle Kanten in der Kantendatei die gleiche Anzahl von Kantengewichten.

Abbildung 3.3 zeigt eine Kantendatei mit drei Gewichten für drei Zeitpunkte. Es gibt in dieser Kantendatei insgesamt 5 Spalten.

Wie in Abbildung 3.3 dargestellt ist, zeigt die erste Spalte (1) die ID der Startknoten und die zweite (2) die ID der Endknoten. Die restlichen Spalten $(3-5)$ repräsentieren die drei Gewichte. In der Kantendatei, in der jede Kante nur ein Kantengewicht enthält, darf das Gewicht nicht auf 0 gesetzt werden: Wenn es keine Kante dazwischen gibt, erscheint die entsprechende Kante in der Kantendatei nicht. In der Kantendatei, in der jede Kante mehrere Kantengewichte enthält, ist es möglich, dass einige der Gewichte vom Wert 0 sind. Das bedeutet, dass zum entsprechenden Zeitpunkt keine Verbindung dazwischen existiert. Deswegen muss jede Kante mindestens ein Gewicht größer als 0 enthalten. Neben den oben vorgestellten Dateiformaten gibt es noch eine weitere Variante (im Format BGraph), in der sowohl die Knoten als auch die Kanten mit entsprechenden Kantengewichten gespeichert sind. Abbildung 3.4 zeigt eine Datei eines solchen Formates.

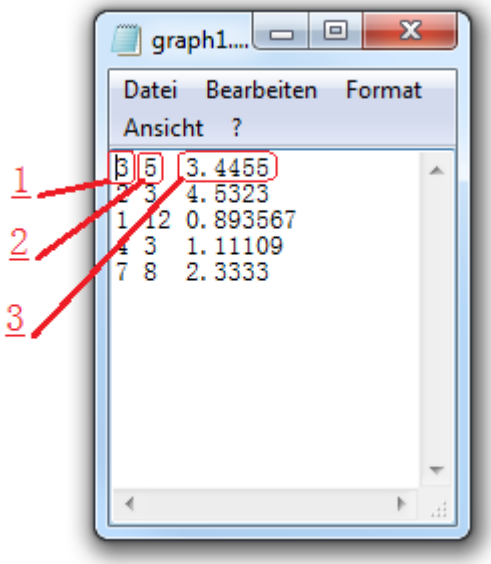


Abbildung 3.2: Kantendatei: Beispiel für 5 gewichtete gerichtete Kanten in einem zeilenbasierten Textformat

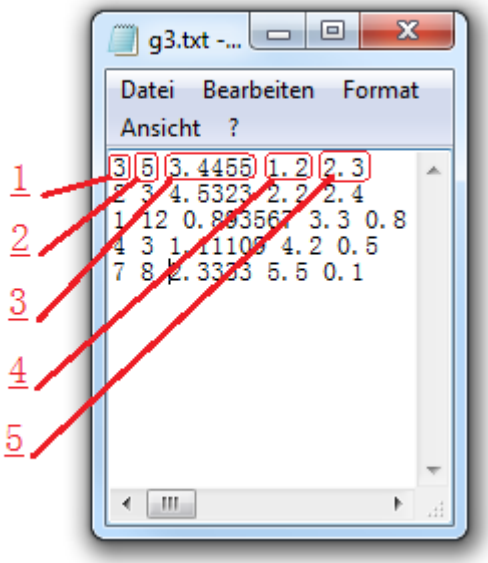


Abbildung 3.3: Kantendatei mit drei Gewichten: Jede Spalte (ab der dritten) in der Textdatei drückt einen Zeitabschnitt aus.

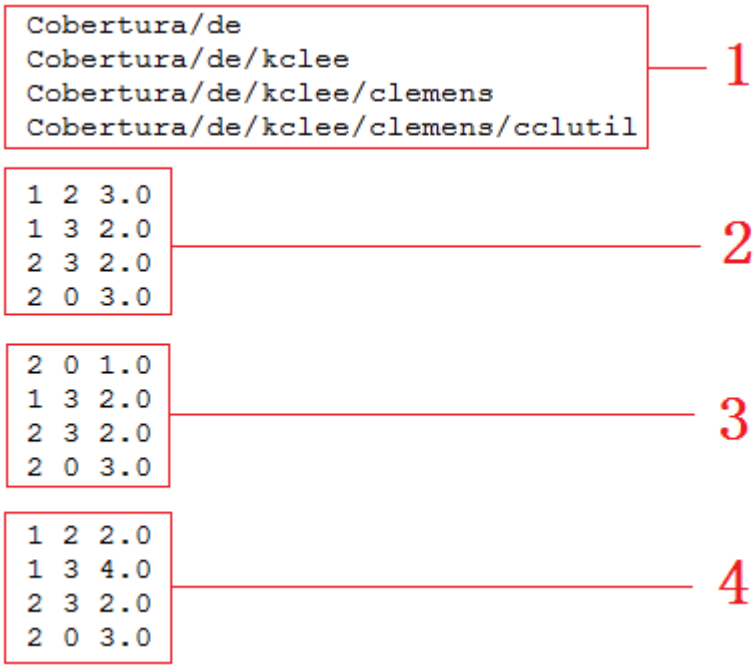


Abbildung 3.4: BGraph-Datei: Textdatei unterteilt in Prolog und dazu leerzeilen getrennte Graphen

Der Prolog vorne (1) listet die Labels in der gleichen Reihenfolge wie die Nummern der Knoten auf. Die IDs der Knoten werden von 0 aufwärts durchnummeriert. Die Datenformate der Teile $2-4$ sind genauso wie das in Abbildung 3.2 gezeigte Format der Kantendatei, die nur ein Kantengewicht enthält. Teil 2 zeigt die Kanten mit Kantengewichten zum Zeitpunkt 1 und Teil 3 zeigt die Kanten mit Kantengewichten zum Zeitpunkt 2 und so weiter. Die Leerzeile gilt als Abgrenzung dieser Teile (also Graphen).

Alle Spalten, nicht nur in der Knotendatei, sondern auch in der Kantendatei, sind durch ein Leerzeichen getrennt. Die Leerzeile darin ist aber verboten.

Als Haupteingabeparameter des Editors sollen die drei verschiedenen Textdateien richtig eingelesen werden, um die enthaltenden Informationen zu verwalten und bereit zur weiteren Bearbeitung zu sein. Ein Dateifilter wird dann nützlich, wenn es gilt, eine Auswahl in einer betroffenen Datei schnell zu treffen.

3.1.2 Graphische Darstellung von Knoten und Kanten

Die zweite Anforderung ist die graphische Darstellung von Knoten und gerichteten partiellen Kanten mit benutzerdefinierter Länge. Eine anschauliche Darstellung für Knoten ist es, sie auf eine Kreisform visuell abzubilden. Zur besseren Skalierung kann man auch nur einen Pixel auf dem Bildschirm wählen (siehe Abbildung 3.5).

(a) Pixelbasierter Ansatz (b) Kreisbasierter Ansatz

Abbildung 3.5: Knotendarstellung: (a) pixelbasierter vs. (b) kreisbasierter Ansätze

Wie in Abschnitt 2.2 schon diskutiert, wird die gerichtete Kante nicht durch einen Pfeil, sondern durch ein nadelförmiges spitz zulaufendes Dreieck repräsentiert (siehe Abbildung 3.6).

(a) traditionelle Kante (b) Tapered Kante

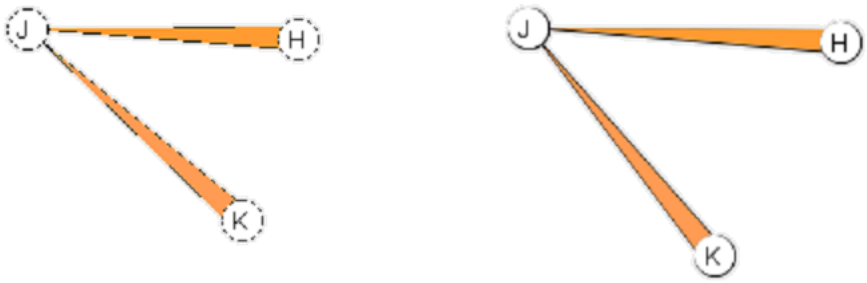
Abbildung 3.6: Kantendarstellung: traditioneller mit Pfeilspize vs. (b) Tapered Ansätze

Die Länge der Kanten soll je nach Bedarf flexibel angepasst werden. Einige häufig verwendete Längenverhältnisse für die allgemeine Analyse sind 12.5%, 25%, 50%, 75%, 90% und 100%. Eine benutzerdefinierte Länge wäre noch bequemer für die Einstellung.

Abbildung 3.6(b) zeigt deutlich, dass die Kante stufenförmige Linien enthält. Bei langen Kanten sieht das nicht gut aus und soll durch Anti-Aliasing entschärft werden.

Abbildung 3.7 zeigt den Vergleich zwischen einem normalen Bild und dem Bild, auf dem ein Anti-Alias Ansatz angewendet wird. Mit diesem Verfahren sehen die Kanten wesentlich schärfer aus, da die bekannten „Bauklötzchen“, die Ecken und Kanten, verschwinden.

Die Beschreibungen (Labels) sollen auch flexibel platziert werden, um Störungen im Diagramm durch Überdeckungen zu reduzieren. Wenn es zahlreiche Knoten und Kanten auf dem Graph gibt, ist die Darstellung von Labels nicht von großer Bedeutung. Im Gegensatz dazu führen die dichten Labels zu einem Überfluss an visuellen Objekten und somit zu Visual Clutter. In diesem Fall sollen die Labels nicht gezeichnet werden, um die Relationen zwischen den Knoten deutlicher und überlagerungsfrei zu zeigen.



(a) Normal (b) Anti-alias

Abbildung 3.7: Visuelle Artefakte: (a) Normal vs. (b) Anti-alias

3.1.3 Interaktion

Heutzutage ist Interaktion zwischen dem Benutzer und dem Programm meistens durch die Maus realisiert. Daher soll der Grapheditor in der Lage sein, auf die Aktion der Maus zu reagieren, um die Knoten zusammen mit Kanten, sowohl eingehende als auch ausgehende Kanten, auf dem Bildschirm hin- und herzuziehen. Weil alle Kanten zu Startknoten und Endknoten gebunden und gerade sind, ist es sinnlos, nur Kanten zu ziehen. Deshalb werden nur die Knoten auswählbar gemacht, wenn der Benutzer mit der Maus daraufklickt und diese zieht. Dadurch wird die Reaktion beschleunigt und die Implementierung erleichtert. Nach dem Ziehen eines Knotens beziehungsweise während dem Ziehen soll der Graph rechtzeitig aktualisiert werden. Deswegen soll nicht nur die Reaktionsfunktion, sondern auch die Funktion, die für die Erstellung von Graphen zuständig ist, sehr effizient sein.

3.1.4 Graphische Darstellung des Kantengewichtes

Das Kantengewicht ist ein wichtiger Parameter für die Kante. Eine visualisierte Darstellung dafür ist die Farbe der Kante. Dadurch wird das Gewicht lebhaft gezeigt, während die numerische Form nur die langweilige Ziffer darstellt. Die Farbe wird dann durch Anwendung der entsprechenden Funktion entsprechend dem Kantengewicht dynamisch berechnet. Das maximale Kantengewicht zeigt sich als rote Farbe, während die Kantengewichte, die näher beim Nullwert liegen, sich mit blauer Farbe zeigen. Wie vorher erklärt wird bei einem Nullwert als Gewicht keine visuelle Darstellung der Kante vorgesehen. Deswegen ist das Minimum von Kantengewichten zwar der Wert 0 und dieser deutet sich nicht mit einer blauen Farbe an, sondern wird ausgegraut dargestellt. Die Farbe vom Hintergrund wird in diesem Fall für den Nullwert eines Kantengewichtes verwendet. Für die Kante, die nur ein Gewicht enthält, reicht eine Farbe für die Kante aus. Für die Kante, die mehrere Kantengewichte enthält, soll das nadelförmige Dreieck, welches die Kante repräsentiert, gleichmäßig in Segmente aufgeteilt werden und jedes Teilstück soll mit der Farbe der zugehörigen Kantengewichtsfarbe gezeichnet werden.

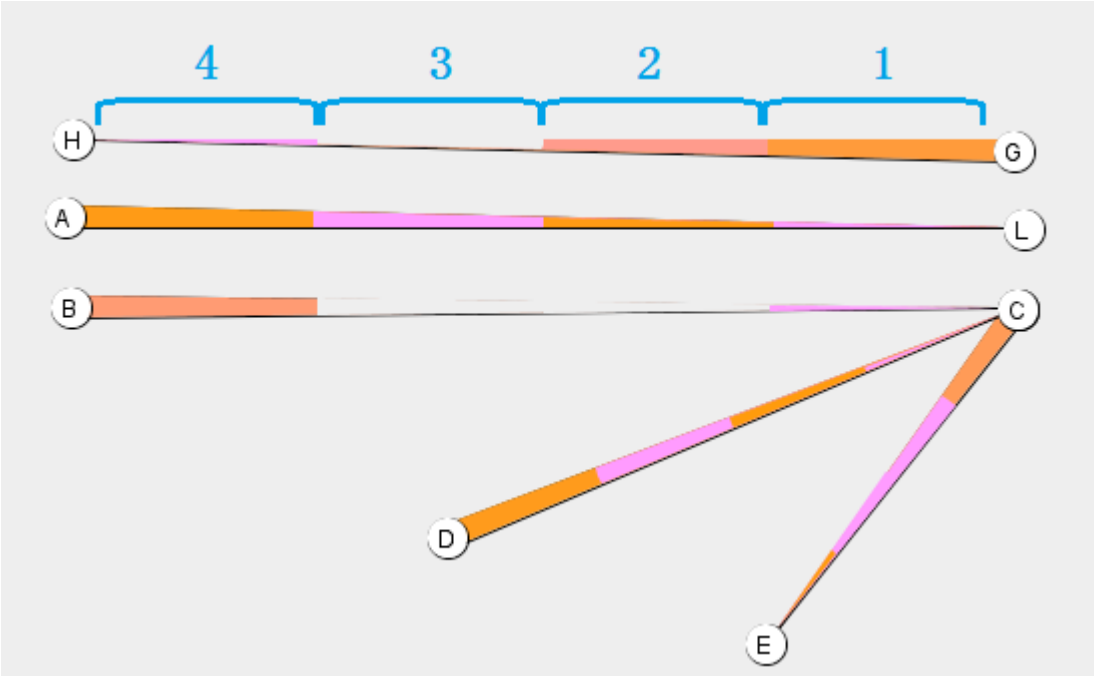


Abbildung 3.8: Kanten mit 4 Gewichten: Das Kantengewicht der Kante GH ist null im dritten Zeitabschnitt, während die andere Kantengewichten größer als null sind.

Abbildung 3.8 zeigt einen Graphen, in dem jede Kante mehrere Kantengewichte enthält. Zum Beispiel gibt es vier Farben auf der Kante von Knoten G nach Knoten H. Das erste Teilstück (1) zeigt das erste Kantengewicht und das dritte Teilstück zeigt das Kantengewicht mit einem Nullwert durch dieselbe Farbe wie der Hintergrund. Es gibt bei der Kante von Knoten B nach Knoten C sogar zwei Nullwertgewichte in der Mitte.

3.1.5 Informationsanzeige

Die Informationsanzeige dient der Unterstützung des Verständnisses eines Graphen durch die Anzeige von Detailinformationen. Während zahlreiche Knoten und Kanten auf dem Graph gezeichnet werden, ist es nicht so einfach, alle eingehenden oder ausgehenden Kanten für bestimmte Knoten zu erkennen. Besonders wenn die Kante mehrere Gewichte enthält, ist es notwendig, neben der Farbe der Kanten auf andere Weise die Kantengewichte anzuzeigen.

3.2 Entwurfsmodell

Nach der Analyse von Anforderungen werden zwei Modelle entwickelt, um die Programmierung durchzuführen: Ein Modell ist das Datenmodell, das die Relationen zwischen Knotendatei und Kantendateien zeigt. Das andere ist das Referenzmodell, das die Beziehungen zwischen den entwickelten Klassen zeigt.

3.2.1 Datenmodell

Die Kantendatei, die nur ein Kantengewicht enthält, kann man als einen Sonderfall der Kantendatei betrachten, die mehrere Kantengewichte enthält. Damit können beide Formate durch eine Liste von Kantengewichten vereinigt werden. Abbildung 3.9 zeigt die Beziehungen zwischen den Knoten und den Kanten.

Abbildung 3.9: Datenmodell: Jede Kante enthält genau einen Startknoten und einen Endknoten.

Es sollen noch andere Eigenschaften ergänzend in die Struktur einfließen: Die Anzahl der Kantengewichte und das maximale Kantengewicht.

3.2.2 Referenzmodell

Um den Grapheditor zu realisieren, sollen einige Klassen entwickelt werden. Das UML-Diagramm in Abbildung 3.10 zeigt eine anschauliche Darstellung der Beziehungen zwischen diesen Klassen. Die Klasse „DrawLinesFrame“ steht in der Mitte und bietet einen Rahmen für das Programm. Alle anderen Funktionsteile werden in verschiedenen Teilen in diesem Rahmen eingehängt, um den gesamten Editor aufzubauen. Die Reaktion auf Mausaktionen wird dann durch den Rahmen realisiert. Die Klasse „FileRead“ beschäftigt sich mit dem Einlesen von Dateien, sowohl dem Einlesen der Knotendatei als auch der Kantendateien. Die Klasse „LinesPainter“ trägt die Hauptaufgaben, um den Graph zu zeichnen. Die Klassen „Node“ bzw. „Edge“ sind die Grundklassen für zwei Datentypen. Während die Klasse „LPanel“ dem Framework hilft, das Bedienfeld zu platzieren, ist die Klasse „MyFileFilter“ extra für das Einlesen von Dateien geschrieben, um alle unerwarteten Dateiformate zu filtern.

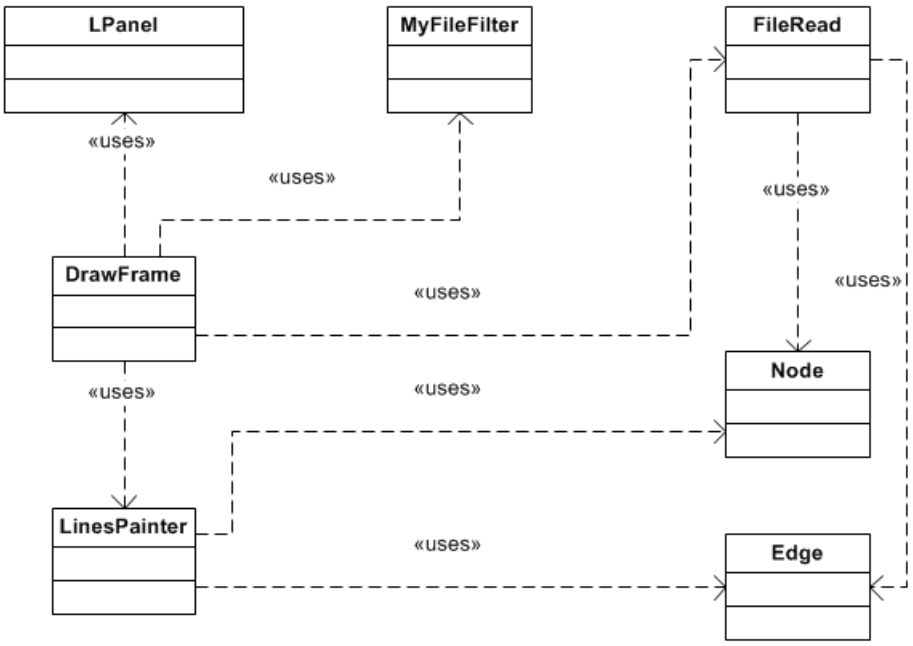


Abbildung 3.10: Referenzmodell: Die Beziehungen zwischen die entwickelten Klasse werden mit Pfeile gezeichnet.

3.3 Beschreibung der Funktionalitäten

Wie in Abschnitt 3.1 gezeigt, gibt es hauptsächlich vier Anforderungen. Um diese Anforderungen zu erfüllen, ergeben sich folgende konkrete Funktionalitäten des Grapheditors: Dateienoperation, Graphendarstellung und weitere Steuerungen des Graphs.

3.3.1 Dateienoperation

Eigentlich gibt es bei der Dateienoperation nur zwei wichtige Aktionen: lese die Informationen über Knoten und Kanten von zwei Textdateien oder von einer gemeinsamen Datei (im Format BGraph). Die Informationen werden dann in zwei Listen gespeichert: die Knotenliste und die Kantenliste.

Als grundlegende Methoden der Klasse „FileRead“ gibt es drei Funktionen: „readLabel“, „readGraph“ und „readBGraph“. Die Funktion „readLabel“ liest die Knotendatei, zieht die Informationen über Knoten heraus und speichert diese Informationen in der Knotenliste für weitere Bearbeitungen ab. Die Verteilung der Knoten wird am Anfang mit einer Random-Funktion zufällig erzeugt. Die aktuellen Positionen von Knoten werden auch in der Knotenliste gespeichert.

Die Funktion „readGraph“ liest die Kantendateien, mit einzelnem Kantengewicht oder mehreren Kantengewichten, und fügt die Informationen über Kanten in eine interne Liste von Kanten ein. Die Funktion „readBGraph“ liest die BGraph-Datei ein, die die Labels für Knoten bzw. die Kanten mit Kantengewichten enthält, parst die Inhalte und speichert die Informationen über den Graph auch in der Knotenliste und Kantenliste. Dann kann die Topologie des Graphs durch die beiden Listen vollständig aufgezeigt werden. Die weiteren Eigenschaften, wie zum Beispiel die Anzahl der Kantengewichte und das maximale Kantengewicht, können beim Einlesen der Kantendatei parallel gesucht und festgelegt werden.

3.3.2 Visualisierung

Die Darstellung der Graphen spielt eine zentrale Rolle, um die gerichteten und gewichteten Relationen zu visualisieren. Die Darstellung ist maßgeblich von der Topologie des Graphen abhängig. Die Topologie ist durch zwei interne Listen gespeichert, nämlich die Knotenliste und die Kantenliste. Die Klasse „LinesPainter“ sorgt dafür, dass die vollständigen Informationen über Knoten und Kanten graphisch dargestellt werden können. Dann soll sie die Informationen von beiden Listen effizient auslesen und dann sofort die entsprechende Grafik auf dem Bildschirm zeichnen. Sie soll auch dafür sorgen, dass bei der Reaktion auf Mausaktionen der Graph dynamisch geändert wird, nicht nur die Positionen der Knoten, sondern auch die Länge der Kanten. Deswegen soll der Graph immer aktualisiert werden. Die Hauptoperationen stehen in der Funktion „paint“, die von der Grundklasse „Component“ geerbt und überschrieben wurde. Die Funktion „paint“ wird dann bei der Aktualisierung des Graphs automatisch aufgerufen.

3.3.3 Steuerung

Es gibt zwei Hauptsteuerungen: das Ziehen der Knoten zusammen mit entsprechenden Kanten und die Anpassung der Kantenlänge. Bevor die richtige Steuerung durchgeführt werden kann, sollen die aktuellen Positionen aller Knoten gespeichert werden. Danach können die Kanten mit neu definierter Länge erneut gezeichnet werden. Eine Suchfunktion soll auch dazu dienen, den nächsten Knoten zu finden, der bei der aktuellen Mausposition im Falle eines Mausklicks, platziert ist. Nur der Knoten, der am nächsten am Mauscursor liegt, ist der richtige Knoten, den der Benutzer ausgewählt hat. Weitere Steuerungen, wie zum Beispiel die Anwendung von zufälligen oder radialen Verteilungen, sollen auch im Bedienfeld zur Verfügung gestellt werden.

4 Implementierung

In diesem Kapitel werden zuerst die Auswahl der Entwicklungsumgebung beschrieben und dann die wichtigen Details bzw. Algorithmen zur Programmierung dargestellt. Es wird also die Umsetzung des Konzepts beleuchtet. Hierbei kann aber nicht auf jede einzelne, während der Entwicklung entstandene Klasse eingegangen werden, da das den Umfang der Arbeit unnötig aufblähen würde. Daher wird nur auf die interessantesten und wichtigsten Aspekte eingegangen.

4.1 Auswahl der Entwicklungsumgebung

Java ist eine objektorientierte Programmiersprache und ein Bestandteil der Java-Technologie des Unternehmens Sun Microsystems (seit 2010 Oracle). Sie „besteht grundsätzlich aus dem Java-Entwicklungswerkzeug (JDK) zum Erstellen von Java-Programmen und der Java-Laufzeitumgebung (JRE) zu deren Ausführung. Die Laufzeitumgebung besteht selbst aus der virtuellen Maschine (JVM) sowie den mitgelieferten Bibliotheken der Java-Laufzeitumgebung“ [Jav]. Durch die Trennung von JDK und JRE wird der Quellcode von Java zuerst in einen maschinenverständlichen Code, den sogenannten Java-Bytecode übersetzt und dann zur Laufzeit der Java-Bytecode in endgültigen Maschinencode transformiert. Da die Ausführung des Java-Bytecodes nicht direkt durch Hardware, sondern durch entsprechende Software auf dem Zielsystem geschieht, verdient Java die bekannte Plattformunabhängigkeit. Um diesen Vorteil von Java zu nutzen, wird Java als Entwicklungsumgebung für die Implementierung gewählt.

4.2 Implementierung der Funktionsblöcke

Nach dem Entwurf sollen die drei Hauptteile implementiert werden: die Dateienoperation, die Graphdarstellung und die Steuerung. Aber zuerst dient das Framework (Hauptfenster) als Grundlage für den Grapheditor, worauf weitere Funktionalitäten hinzugefügt werden können.

```
1 f = new JFrame("dynamische Graphvisualisierung");
2 f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Listing 4.1: Erzeugung eines Frameworks

4.2.1 Framework

Abbildung 4.1 zeigt einen Grapheditor mit Rahmen und Bedienfeld. Am Anfang reicht der einfache Rahmen aus. Listing 4.1 zeigt, wie ein einfacher Rahmen erzeugt werden kann. Die weiteren Funktionalitäten werden Schritt für Schritt in den Rahmen hinzugefügt.

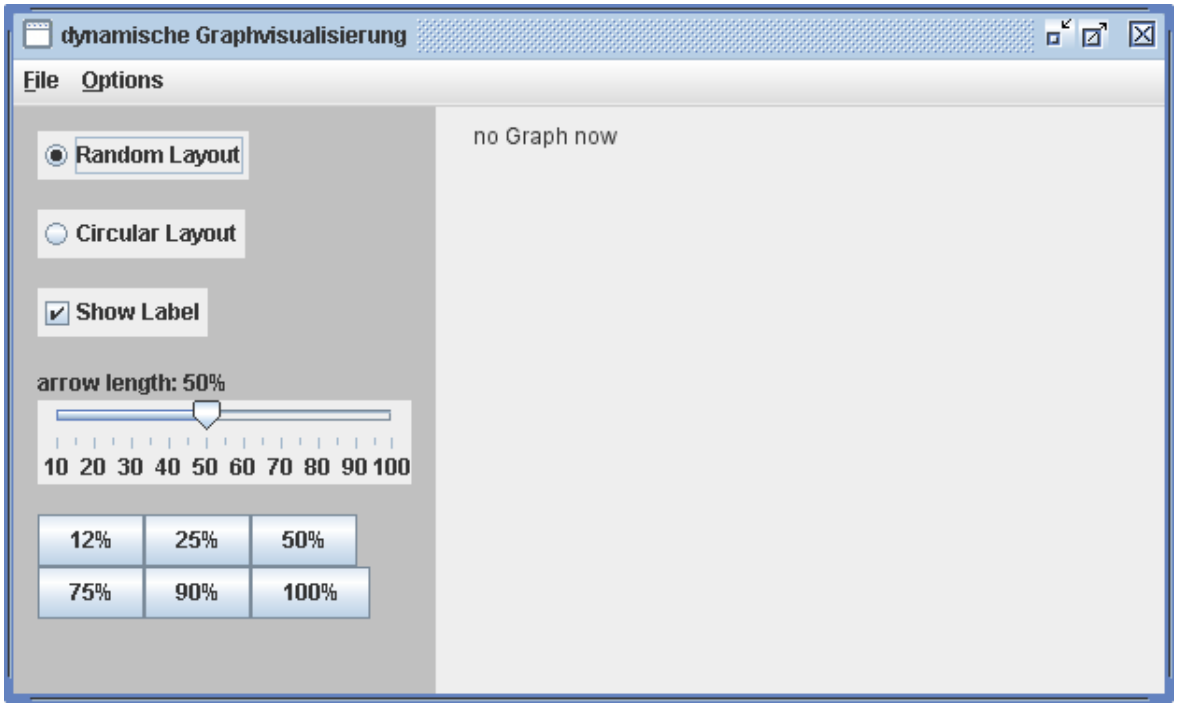


Abbildung 4.1: Grapheditor: Das Hauptfenster mit Bedienfeld ohne Graph darzustellen

4.2.2 Dateienoperationen

Wie in Abschnitt 3.1.1 gezeigt, gibt es zwei Varianten von Daten: Die Daten von Knoten und Kanten werden entweder in zwei Textdateien getrennt gespeichert oder in einer Datei (BGraph-Format). Jedes Dateiformat soll durch eine entsprechende Lesefunktion bearbeitet werden können. Dann wird eine Menüleiste in den Rahmen integriert, um die Entscheidungen des Benutzers zu treffen, welche Formate von Dateien eingelesen werden sollen. Listing

```
1 //add "Open Label" menu in "File"
2 menuOpenLabel = new JMenuItem("Open Label");
3 menuOpenLabel.addActionListener(this);
4 menuOpenLabel.setEnabled(true);
5 menu.add(menuOpenLabel);
```

Listing 4.2: Hinzufügen der Menüelemente

4.2 zeigt, wie man ein neues Element in der Menüleiste hinzufügt und den Zugriff auf die Funktionalitäten des Programms ermöglicht.

Durch die Funktion addActionListener wird das Datenelement bei dem entsprechenden ActionListener angemeldet, der die Behandlungen in die Funktion actionPerformed weiterleitet. Drei Menüelemente für das Einlesen von Daten werden dann im Rahmen hinzugefügt. Abbildung 4.2 zeigt die Menüs.

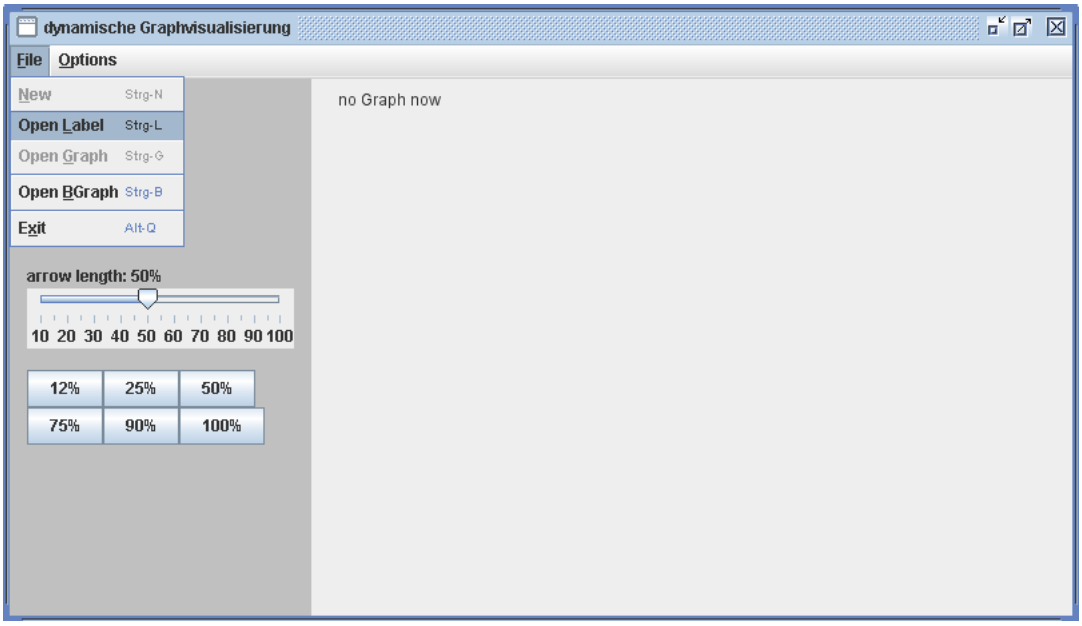


Abbildung 4.2: Menü: Man kann durch die Menüs entweder zuerst eine Knotendatei und danach eine Kantendatei einladen oder direkt eine BGraph-Datei einladen.

Open Label ist zuständig für das Lesen der Knotendatei, während Open Graph für das Lesen von Kantendateien zuständig ist. Open BGraph ist dann nur für die Dateien verantwortlich, die sowohl Knoten als auch Kanten in einem BGraph-Format enthalten.

Ein Dialogfenster für das Öffnen von Dateien wird mit der Klasse javax.swing.JFileChooser als ein vorgefertigter Dialog zur Verfügung gestellt. Dieser bietet einen Datei-Explorer und Buttons zum Öffnen und Abbrechen. Damit der JFileChooser effizienter eingesetzt

```
1  public void actionPerformed(ActionEvent e) {
2  // TODO: Add your code here
3  String command = e.getActionCommand();
4  ...
5  //action for the menu "File/Open Label"
6  if (FILE_OPEN_LABEL.equals(command)){
7    openLabel();
8  }
9  ...
```

Listing 4.3: Behandlung des ActionListeners

werden kann, wird ein FileFilter verwendet. Damit können wir z.B. die Auswahl von Dateien auf bestimmte Endungen wie „.txt“ oder „.bgraph“ beschränken. Abbildung 4.3 zeigt die Auswahl einer BGraph-Datei.

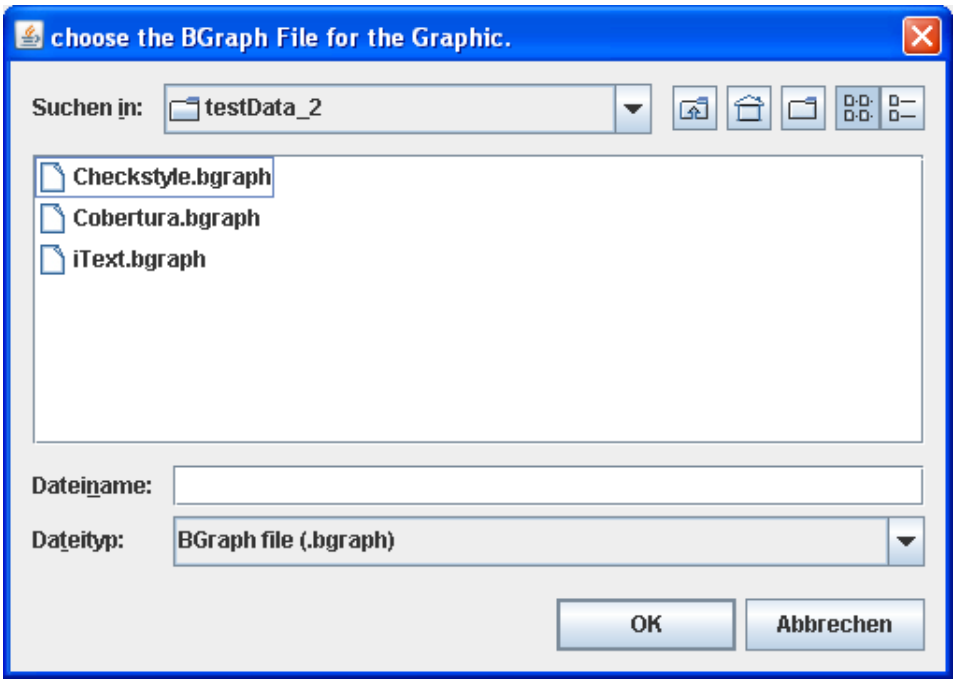


Abbildung 4.3: fileChooser: Bei der Auswahl einer BGraph-Datei werden die Dateien, die die Endung „.bgraph“ haben, im fileChooser sichtbar gemacht.

Wie gesagt, werden alle Reaktionen im Menü durch die Funktion actionPerformed zusammen verarbeitet und dann nach und nach ihre Befehle weitergeleitet. Dadurch wird die richtige Funktion mit dem Menüelement verknüpft. Listing 4.3 zeigt die allgemeine Behandlung des ActionListeners.

```
1  private void openLabel() {
2    ...
3    if (result == JFILEChooser.APPROVE_OPTION) {
4      file = fileChooser.getSelectedFile();
5      fileName = file.getAbsolutePath();
6
7      // read the Label file
8      txtFile = new FileRead();
9
10     // read the list of nodes
11     nodes = txtFile.readLabel(fileName);
12     ...
13   }
14   ...
15 }
```

Listing 4.4: Die Funktion openLabel

```
1 nodes = txtFile.readLabel(fileName);
2 edges = txtFile.readGraph(fileName);
```

Listing 4.5: Einlesen der Knoten- und Kantendatei

Die Funktion openLabel ist dann so aufgebaut, dass zuerst durch den FileChooser die richtige Datei ausgewählt wird, danach werden dann die Informationen in der Liste gespeichert. Die Struktur der Funktion openLabel sieht wie in Listing 4.4 aus:

Nur wenn eine Bestätigung durch fileChooser vom Benutzer ankommt, wird die echte Leseoperation durchgeführt. Der Prozess beim Einlesen von Kanten ist ähnlich wie beim Einlesen der Knotendatei (siehe Listing 4.5).

Die wesentlichen Funktionen readLabel und readGraph sind die Methoden der Klasse FileRead. Da das Format der Knotendatei festgelegt ist, ist die Implementierung dafür ganz einfach. Mit einem Scanner und einer Schleife werden die Informationen über Knoten dann in der Liste nodes gespeichert (siehe Listing 4.6).

```
1 while(scan.hasNext()){
2   id = scan.nextInt();
3   label = scan.next();
4   nodes.add(new Node(id,label));
5 }
```

Listing 4.6: Schleife zum Knoten einlesen

```
1  while (scan.hasNextLine()){
2    String str = scan.nextLine();
3    String[] numList = str.split("\\s");
4    start = Integer.parseInt(numList[0]);
5    end = Integer.parseInt(numList[1]);
6    float [] weightList = new float [numList.length - 2];
7    for(int i=2;i<numList.length;i++){
8      float num = Float.parseFloat(numList[i]);
9      weightList[i-2] = num;
10   }
11   Edge newEdge = new Edge(start, end, weightList);
12   newEdge.setWeightNo(weightList.length);
13   edges.add(newEdge);
14 }
```

Listing 4.7: Schleife zum Kanten einlesen

Für die Kantendatei ist die Anzahl der Gewichte unbekannt und die Anzahl soll zuerst beim Einlesen dynamisch festgelegt werden. Dann kann ein passendes Array definiert werden, um alle Kantengewichte zu speichern. Auf diese Weise können sowohl die Kantendatei, in der jede Kante nur ein Kantengewicht enthält, als auch die Kantendatei, in der jede Kante mehrere Kantengewichte enthält, gelesen werden. Eine erweiterte Schleife dafür sieht wie in Listing 4.7 gezeigt aus.

Beim Einlesen der BGraph-Datei wird die Datei zwei Mal durchgelesen, um die Informationen über Knoten bzw. Kanten zu trennen. Beim ersten Durchlesen werden die Labels von Knoten bzw. die Nummer der Zeile als ID des Knotens in der Knotenliste gespeichert. Die IDs werden von 0 aufwärts durchnummeriert. Dabei wird die Anzahl des Zeitabschnittes durch die Kennzeichnung von Leerzeilen festgelegt.

Beim zweiten Durchlesen werden nur die Informationen über Kanten bearbeitet. Weil zu jedem Zeitabschnitt eine Kante nur ein Kantengewicht enthält, soll ab dem zweiten Zeitabschnitt von den Kanten geprüft werden, ob dieselbe Kante im vorläufigen Zeitabschnitt auftritt. Wenn ja, dann soll das neue Kantengewicht in der Liste der Kantengewichte eingefügt werden. Wenn nicht, dann soll eine neue Kante in der Kantenliste eingefügt werden, deren Liste für Kantengewichte den 0 Wert für die vorläufigen Zeitabschnitte hat.

Durch die oben genannten Prozesse werden dann die Informationen über Knoten und Kanten eingelesen und in zwei Listen getrennt gespeichert.

4.2.3 Canvas

Nach der Erzeugung von Knotenliste und Kantenliste kann der Graph aufgebaut und dann durch die Klasse LinesPainter gezeichnet werden (siehe Listing 4.8). Eine Aktualisierung des Graphs wird dann automatisch durchgeführt.

```
1 lines.setGraph(edges, nodes);
```

Listing 4.8: Verknüpfung der Listen und dem dynamischen Graph

```
1 @Override
2 public void paint(Graphics g_in) {
3   ...
4 }
```

Listing 4.9: Funktion: Paint

„lines“ ist ein Objekt der Klasse LinesPainter und ist zuständig für das Zeichnen des Graphs. Nach der Übertragung von Knotenliste und Kantenliste werden alle Informationen über den Graph an das Objekt „lines“ weitergebeben. Dann werden alle Zeichenoperationen innerhalb der Funktion paint durchgeführt, die dieselbe Funktion von java.awt.Component überschreibt. Eine Annotation @Override soll dann dafür eingesetzt werden, um damit Warnungen und Fehlermeldungen zu unterdrücken [ann] (siehe Listing 4.9).

Wie in Abschnitt 3.1.2 gezeigt, soll die Kantenglättung verwendet werden, um die bekannten Bauklötzchen zu beseitigen. In Java wird die Anti-Aliasing Technik für Grafikprogrammierung benutzt. Das Einschalten von Anti-Aliasing ist wie in Listing 4.10 gezeigt [ant].

Wie zuvor diskutiert, sind alle Kanten an Startknoten und Endknoten gebunden. Deswegen kann beim Zeichnen des Graphs die Liste von Kanten nur einmal durchgelesen werden, um den gesamten Graph zu zeichnen. Der Prozess fürs Zeichnen wird wie in folgendem Pseudo-Code im Stil von JAVA gezeigt (siehe Listing 4.11):

Zuerst werden die Listen von Knoten und Kanten überprüft, ob sie die richtigen Daten enthalten. Wenn die Daten dort gespeichert sind, kann die Schleife in Zeile 2 durchgeführt werden, um alle Kanten durchzulesen. In der Schleife werden die Operationen von Zeile 3 bis 8 nacheinander durchgeführt, um die Informationen herauszuziehen und dann werden diese Daten graphisch dargestellt.

In Abschnitt 3.1.2 wird die graphische Darstellung von Kanten und Knoten illustriert. Bei der Implementierung werden ein paar Tricks verwendet, um diese Anforderungen zu erfüllen und den Graph somit effizient zu zeichnen.

```
1 // Kantenglaettung
2 Graphics2D g = (Graphics2D)g_in;
3 g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
4                    RenderingHints.VALUE_ANTIALIAS_ON);
```

Listing 4.10: Anti-Aliasing

```
1  if(edges != null && nodes != null){
2    for(Iterator itr = edges.iterator();itr.hasNext();){
3      getEdge();
4      getStartNode();
5      getEndNode();
6      drawEdge();
7      drawStartNode();
8      drawEndNode();
9    }
10 }
```

Listing 4.11: Pseudo-Code im Stil von JAVA: Zeichenprozess

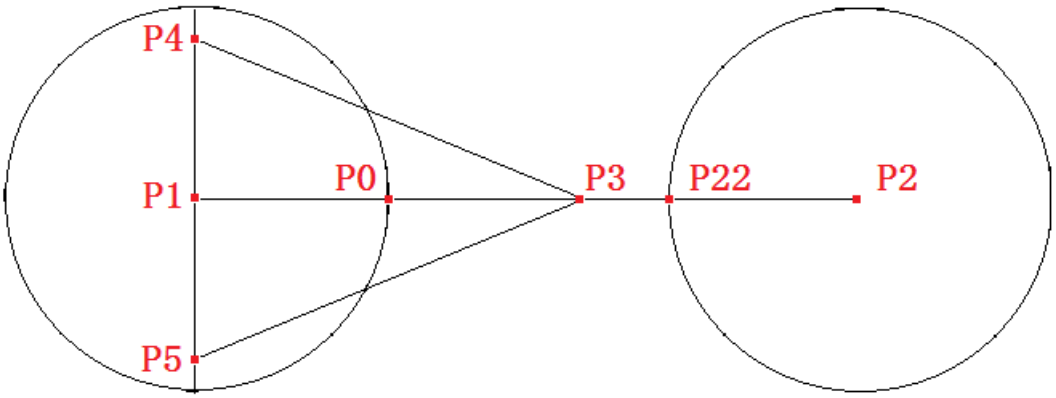


Abbildung 4.4: Knoten und Kante: Eine partielle Kante zwischen Knoten $P1$ und Knoten $P2$ wird durch ein paar Punkte festgelegt und dann gezeichnet.

Abbildung 4.4 zeigt eine Kante vom Punkt $P1$ zum Punkt $P2$. Wenn eine solche Kante gezeichnet wird, werden folgende Schritte durchgeführt:

1. Suche zwei Punkte $P4$ und $P5$ innerhalb des Kreises von $P1$ mit Radius des Knotenkreises, damit die Linienlänge zwischen P4 und P1 genauso lang ist wie die Linienlänge zwischen $P1$ und $P5$. Außerdem sollen die Punkte $P1$, $P4$ und $P5$ in eine gerade Linie bewegt werden. Die gerade Linie $P4P5$ und die gerade Linie $P1P2$ schneiden sich rechtwinklig.
2. Nach der aktuellen Skala der Kante berechnet man die Position für den Punkt $P3$, damit die Länge zwischen $P0$ und $P3$ mit der definierten Länge identisch ist. Wenn die Skala gleich 0% ist, liegt $P3$ genau auf dem Punkt $P0$. Wenn die Skala gleich 100% ist, dann liegt $P3$ genau auf dem Punkt $P22$. Das bedeutet, dass die richtige Länge für die Kante von $P1$ nach $P2$ sich um zweimal den Radius des Knotenkreises verkleinert.

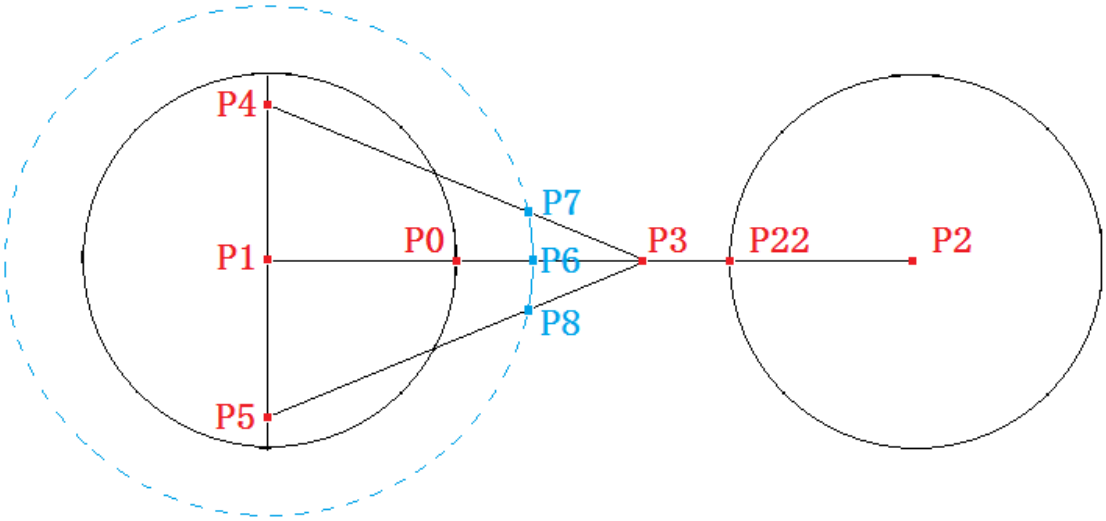


Abbildung 4.5: Kante mit mehreren Gewichten: Die partielle Kante wird nach Anzahl der Kantengewichten unterteilt und die Kurve wird durch zwei gerade Linien ersetzt.

3. Zeichne eine gerade Linie zwischen $P4$ und $P3$, $P5$ und $P3$ und fülle die Farbe, die vom Kantengewicht herrührt, innerhalb des resultierenden Dreiecks $(P4 - P3 - P5 - P4)$. Dann zeichne zwei Kreise mit Mittelpunkt der Strecke von $P1$ nach $P2$, um die interne Linie zu überdecken.

4. Schließlich werden die textuellen Beschreibungen darauf gezeichnet (optional).

Für die Kante, die mehrere Kantengewichte enthält, sollen noch zusätzliche Punkte berechnet werden. Angenommen, dass der Radius des Knotenkreises R ist, die Anzahl der Kantengewichte $N(N > 1)$ ist und die Länge für die Linie zwischen $P0$ und $P2$ L ist. Dann wird die Kante in N Teile gleichmäßig unterteilt. Genau wie vorher wird nur die Linie zwischen $P0$ und $P22$ unterteilt. Das heißt, der Radius vom berechneten Kreis (Blauer Kreis in Abbildung 4.5) ist $R + L/n$.

Durch die Schnittstellen können die Positionen der Punkte $P6$, $P7$ und $P8$ berechnet werden. Statt der Kurven von Punkt $P7$ nach Punkt $P8$ werden zwei gerade Linien von $P7$ nach $P6$ und von $P6$ nach $P8$ annähernd gezeichnet. Dann wird das Polygon $P7 - P3 - P8 - P6 - P7$ mit der Farbe gefüllt, die vom zweiten Kantengewicht abhängig ist. Dadurch wird die Kante in zwei Teile eingefärbt. Wiederhole diese Schritte, jedes Mal erhöht sich der Radius um L/n,

bis der Radius mit $R + (n-1)L/n$ identisch ist. Dann werden die weiteren Kantengewichte auf die Kante gezeichnet und eingefärbt.

Die Farbe des Kantengewichtes verläuft von blau nach rot und ändert sich linear. Das heißt, das maximale Kantengewicht zeigt sich mit roter Farbe und das minimale Kantengewicht (nicht gleich 0 sondern nahezu 0) mit blauer Farbe. Die Farben der Kantengewichte dazwischen werden dann nach dem RGB-Model berechnet. „Dem Konstruktor werden die Rot-, Grün- und Blauanteile als Parameter übergeben. Die angegebenen Werte müssen alle im Bereich zwischen 0 und 255 liegen“ [RGB]. Damit wird die Farbe wie folgt berechnet:

$$col = weight / maxWeight$$

$$c = Color((int)(255 * col), 0, (int)(255 * (1 - col)))$$

Wie in Abschnitt 3.1.4 vorgestellt, wird für die Farbe des Nullwertes eines Kantengewichtes die Farbe des Hintergrundes verwendet. Die entsprechende Farbe ist wie folgt definiert:

$$c = Color(238, 238, 238)$$

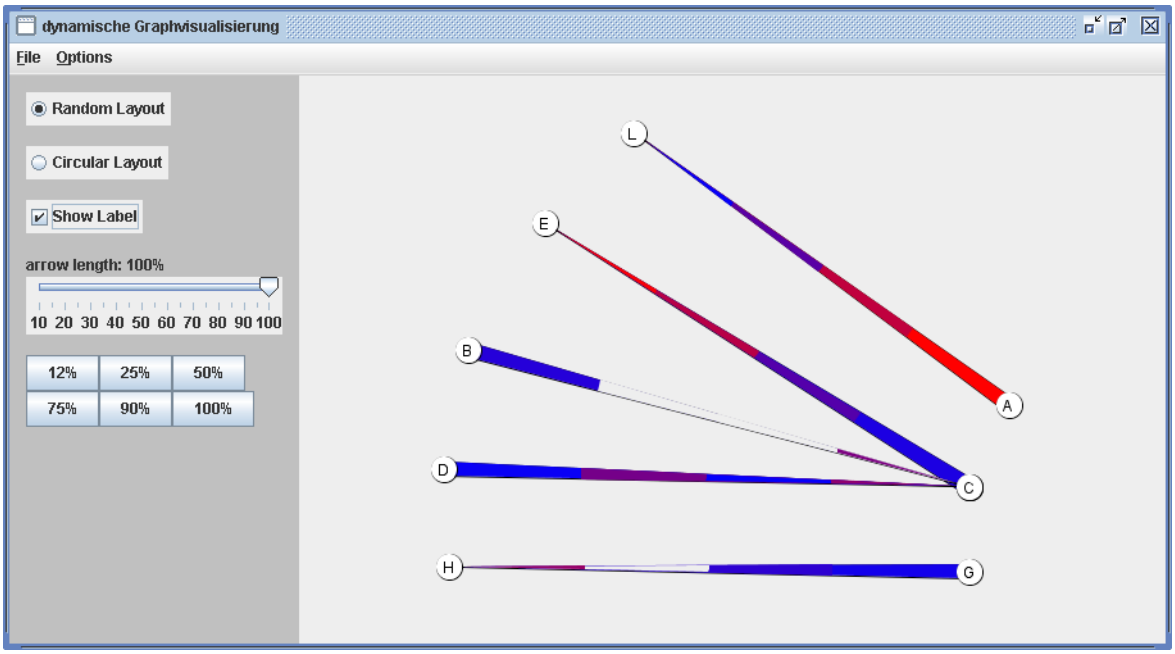


Abbildung 4.6: Farbverlauf in den Kanten: Mit Farbe wird die Änderung der Kantengewichten in verschiedenen Zeitabschnitten klar dargestellt.

Abbildung 4.6 zeigt einen Graph mit dynamischen Kanten. Es gibt vier Zeitabschnitten und damit soll jede Kante in vier Teilstücke unterteilt werden. Während die Kantengewichte der Kante von Knoten A nach L immer kleiner werden, steigen die Kantengewichte der Kante von Knoten C nach E. Auf der Kante von Knoten B nach C gibt es zwei Teilstücke,

deren Farbe gleich mit der Farbe des Hintergrundes ist. Dadurch wird gezeigt, dass es keine Verbindungen zwischen Knoten B und C in diesen Zeitabschnitten gibt.

In der Praxis gibt es die Möglichkeit, dass die meisten Kantengewichte sehr klein sind, während einige davon erheblich groß sind. In diesem Fall werden die meisten Kanten mit blauer Farbe gezeichnet und die Unterschiede dazwischen verdeckt. Eine besser Lösung dafür ist die Umrechnung der Kantengewichten mit dem Logarithmus. Dann wird die Linearität der Kantengewichte aufrechterhalten und die Tendenz der Kantengewichte wird klar dargestellt.

Für die Darstellung von Knoten können auch verbessert werden, indem die Radien der Knoten von Summe der ausgehenden Kantengewichte abhängen. Je mehr ausgehende Kantengewichte ein Knoten besitzt, umso wichtiger ist der Knoten und soll damit größer dargestellt werden. Die Knoten, die gar keine ausgehende kante besitzen, spielen eine triviale Rolle und können mittels eines kleinen Kreis dargestellt werden.

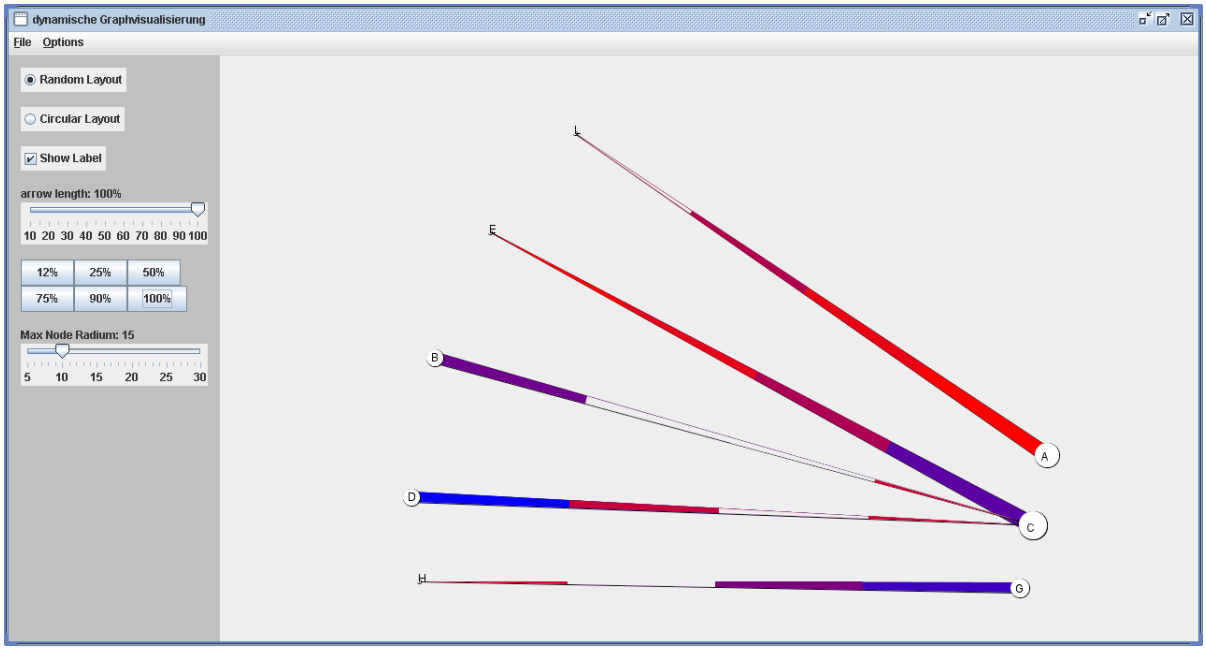


Abbildung 4.7: Farbverlauf in den Kanten: Mit Logarithmus werden die Kantengewichte umrechnet, um die kleinen Unterschiede zu vergrößern. Die Radien der Knoten sind nach der Summe aller ausgehenden Kantengewichte berechnet.

Abbildung 4.7 zeigt die verbesserte Darstellung von Kantengewichte und Knoten. Aber man muss bei der Verwendung von Logarithmus aufpassen, dass die Kantengewichte, die genau 1 sind, werden dann nach 0 umrechnet und dann die Farbe von Hintergrund dafür verwendet. In diesem Fall ist es schwer zu unterscheiden, ob die Kanten in diesem Zeitabschnitt gibt.

4.2.4 Bedienfeld

„Ein Bedienfeld (Panel) ist eine leere Fläche (gedanklich eine Metallplatte eines Gerätes mit Anzeigen und Schaltern), die weitere Steuerelemente aufnehmen kann (Container-Element), deren Größe berechnet und danach diese selbständig arrangiert“ [Bed]. Im Grapheditor bietet das Bedienfeld die häufig verwendeten Funktionen an (siehe Abbildung 4.8).

Der Grapheditor stellt momentan zwei Verteilungen von Knoten zur Verfügung: zufällige oder kreisförmige Verteilung. Man kann aber nur eine davon auswählen, weil zu einem Zeitpunkt nur eine Verteilung von Knoten verwendet wird. Dann zeigen sich die beiden Verteilungen als JRadioButton innerhalb einer Gruppe. Wenn später noch weitere Verteilungen angeboten werden, können diese in der Gruppe hinzugefügt werden. Die Anzeige von Beschreibungen ist eine „Ja oder Nein“ Auswahl und dafür reicht ein Kontrollkästchen CheckBox aus.

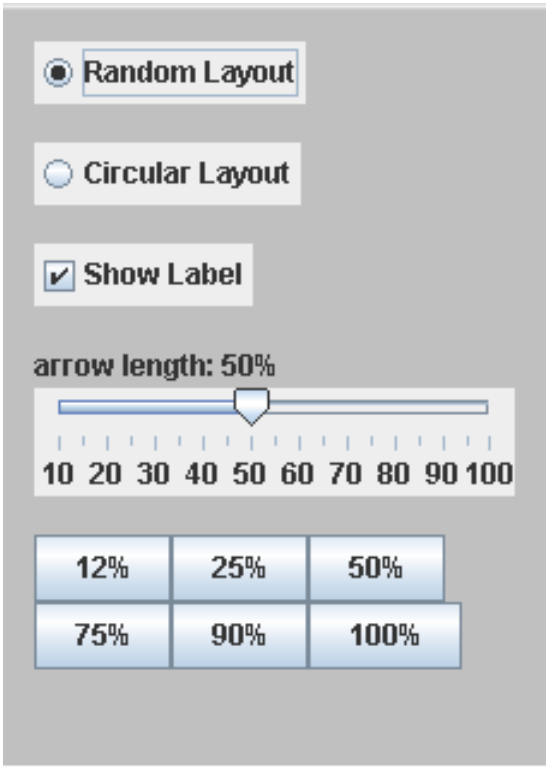


Abbildung 4.8: Bedienfeld des Grapheditors: Nur die am häufigsten verwendeten Funktionen werden im Bedienfeld platziert.

Wie im vorläufigen Kapitel erläutert, ist die Anpassung der Kantenlänge eine Hauptaufgabe des Grapheditors. Deswegen gibt es sogar zwei Werkzeuge im Bedienfeld, um die Anforderungen zu erfüllen. Während der Schieberegler (eng. Slider) die Möglichkeit bietet, die Kantenlänge schnell und flexibel einzustellen, bietet die Gruppe von Knöpfen die Auswahl

```
1 public Component addComponent(Component component, int width, int anchor){...}
```

Listing 4.12: Funktion: addComponent

an Skalen. Eine zusätzliche Anzeige für die aktuelle Skala in Prozent hilft bei der Einstellung der Kantenlänge.

Um das obige in Abbildung 4.8 gezeigte Bedienfeld zu implementieren, sollen zuerst alle entsprechenden Komponenten im Bedienfeld eingebaut werden und dann das Bedienfeld in der linken Seite des Editors angezeigt werden. Nach der Definition eines Bedienfeldes ist es klar, dass alle Elemente darin vom Bedienfeld selbst angeordnet werden. Dann ist das Aussehen des Bedienfeldes nicht festgelegt, sondern vom Hauptfenster abhängig. Um eine feste Aussicht des Bedienfeldes zu gewinnen, wird eine eigene Klasse dafür entwickelt.

Wie im Abschnitt 3.2.2 gezeigt, ist die Klasse Panel für die Anzeige des Bedienfeldes zuständig. Um alle Komponenten in ordentlichen Reihen im Bedienfeld anordnen zu können, wird ein bestimmtes Layout verwendet, und zwar das GridBagLayout. Die entsprechenden Beschränkungen für das Layout GridBagConstraints werden auch dort definiert. Darin gibt es auch eine wichtige Funktion, die für alle Komponenten von Java geeignet ist, um die Komponenten richtig und bequem im Bedienfeld hinzuzufügen. In Listing 4.12 sieht man diese Funktion mit Parametern.

Während der Parameter component die hinzu zufügende Komponente zeigt, markieren die Parameter width und anchor die Position dafür. Der Wert von width ist entweder 0 oder 1: bei 0 zeigt dies an, dass die Komponente die letzte Komponente in einer Zeile ist und bei 1 nicht. Der Wert von anchor hat drei Möglichkeiten: -1, 0 und 1, die mit West, Mittel und Ost korrespondierend sind. Mit dieser addComponent-Funktion werden dann die benötigten Funktionen für das Einfügen von anderen Gestaltungselementen, wie zum Beispiel JButton, JRadioButton, JSlider und JCheckBox, auf eine ähnliche Weise neu definiert. In Listing 4.13 werden die durch die addComponent-Funktion neu definierten Funktionen gezeigt.

Im Hauptrahmen DrawLinesFrame wird dann das Bedienfeld eingebaut.

Wie in Listing 4.14 gezeigt, wird das Bedienfeld in Richtung Nordwest des Grapheditors platziert und bleibt während dem Programmlauf dort im Bedienfeld. Die Antwort auf die Komponenten im Bedienfeld sind auf die gleiche Weise wie bei Menü implementiert. Die Funktion actionPerformed sorgt dafür und bindet die grundlegenden Behandlungsfunktionen mit den Komponenten zusammen.

4.2.5 Knotenlayout

Die von Dateien extrahierten Informationen über den Graph enthalten nur die gerichteten und gewichteten Relationen, keine spezifischen Informationen über die Knotenverteilung. Deswegen soll der Grapheditor eine Knotenverteilung zur Verfügung stellen, um den Graph richtig anzuzeigen. Eine intuitive Knotenverteilung ist die zufällige Verteilung. Aber es

```
1  public JButton addJButton(String buttonString,
2                            int width, int anchor){
3    return (JButton)(addComponent(
4                  new JButton(buttonString), width, anchor));
5  }
6
7  public JRadioButton addJRadioButton(String buttonString,
8                            int width, int anchor){
9    return (JRadioButton) (addComponent (
10                 new JRadioButton(buttonString), width, anchor));
11 }
12
13 public JSlider addJSlider(int min, int max, int value,
14                           int width, int anchor){
15   return (JSlider)(addComponent(
16                 new JSlider(min,max,value), width, anchor));
17 }
18
19 public JRadioButton addJRadioButton(String buttonString,
20                           int width, int anchor){
21   return (JRadioButton) (addComponent (
22                 new JRadioButton(buttonString), width, anchor));
23 }
```

Listing 4.13: Neu definierte Funktionen

```
1 LPanel p = new LPanel();
2 ...
3 //add components here
4 JPanel controls_superpanel = new JPanel();
5 controls_superpanel.add("North", p);
6 p.finish();
7 contentPane.add("West", controls_superpanel);
```

Listing 4.14: Einfügen des Bedienfeldes

gibt wahrscheinlich Knotenüberlappungen oder die Verteilung ist nicht so geeignet für die Anzeige des Graphs mit partiellen Kanten. Wie in Abschnitt 2.1.5 gezeigt, ist die radiale Verteilung eine bessere Lösung dafür. Bei dieser Verteilung sind alle Knoten gleichmäßig auf einem Kreis verteilt und damit keine Störung untereinander. Alle Kanten liegen dann innerhalb des Kreises und sind deutlich sichtbar, auch wenn nur partielle Kanten gezeichnet werden.

Random Layout

Die Implementierung der zufälligen Verteilung ist einfach. Vorher sind die Informationen über Konten in der Liste „nodes“ gespeichert worden. Dann kann die Liste von Knoten nur

```
1 for(Iterator itrNode = nodes.iterator();itrNode.hasNext();){
2   Node node = (Node) itrNode.next();
3   double x,y;
4   x = Math.random() * 1000;
5   y =  Math.random() * 800;
6   node.setX(x);
7   node.setY(y);
8 }
```

Listing 4.15: Zufällige Verteilung

einmal durchgelesen und die Koordinate für jeden Knoten mit einer zufälligen Funktion erzeugt werden. Listing 4.15 zeigt die Kernschleife, um die Positionen der Knoten innerhalb des rechteckigen Bereiches von Punkt $(0,0)$ bis Punkt $(1000,800)$ zufällig zu generieren. Abbildung 4.9 zeigt eine solche zufällige Verteilung von 12 Knoten im Grapheditor.

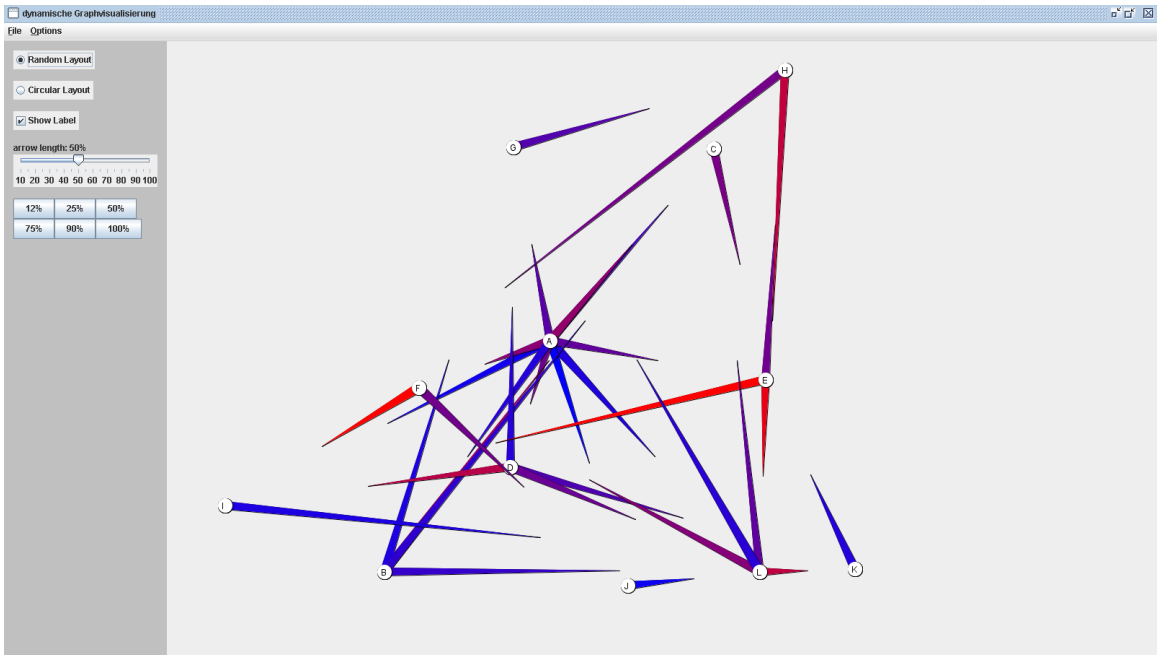


Abbildung 4.9: Zufällige Verteilung von 12 Knoten: Die Position jedes Knotens wird zufällig generiert.

Circular Layout

Bei der radialen Verteilung soll zuerst der Kreis nach der Anzahl der Knoten gleichmäßig unterteilt werden. Weil die Winkel im Kreis genau doppelt so groß wie PI (360 Grad) sind, ist die Verteilung nach Winkeln leicht zu realisieren. Die Polarkoordinaten sind dann einfach

zu berechnen und danach in kartesische Koordinaten umzurechnen. Eine anschauliche Darstellung der Transformation zwischen Polarkoordinaten und kartesischen Koordinaten ist in Abbildung 4.10 gezeigt.

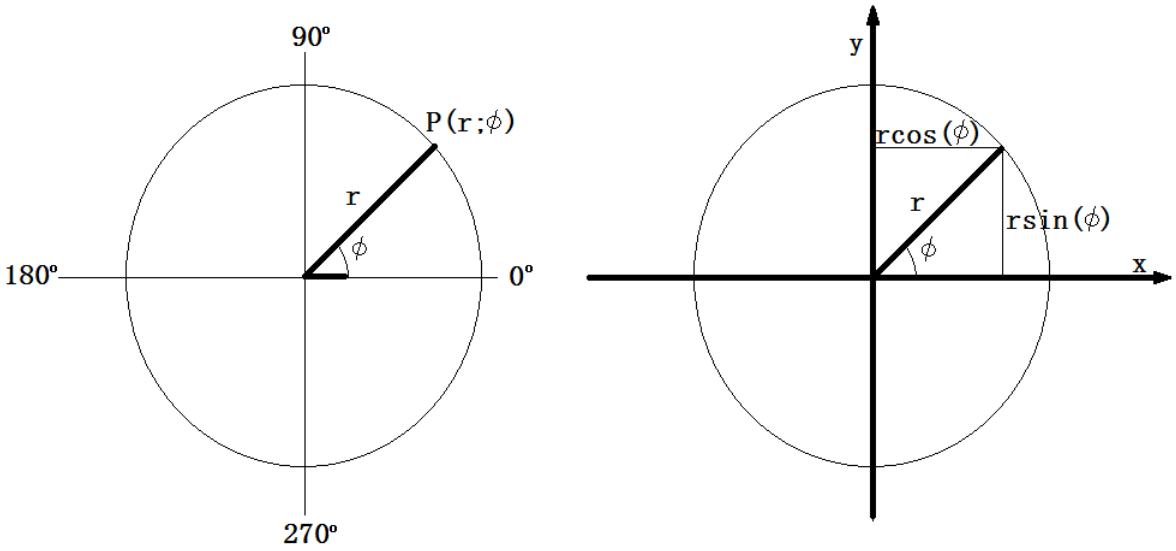


Abbildung 4.10: Ebene Polarkoordinaten (mit Winkelangaben in Grad) und ihre Transformation in kartesische Koordinaten

Die Umrechnung von Polarkoordinaten in kartesische Koordinaten geschieht dann wie folgt:

$$\begin{aligned} x &= r * \cos\varphi \\ y &= r * \sin\varphi \end{aligned} \tag{4.1}$$

Im Programm wird der Punkt $(600, 400)$ als Mittelposition des Kreises gewählt und ist der Radius des Kreises mit 360 Pixel. Dann wird der erste Knoten immer am Punkt $(960, 400)$ platziert. Zu einer festen Anzahl von Knoten ist die kreisförmige Verteilung dann auch festgelegt und damit der entsprechende Graph fest. Die Implementierung ist sehr ähnlich wie bei einer zufälligen Verteilung. Man soll die Anzahl der Knoten dazu verwenden, um die richtigen Winkel zu berechnen. Das Winkelmaß erhöht sich jedes Mal um $2.0 * Math.PI/nodesNo$, um zur nächsten Punktposition zu gelangen. Listiing 4.16 zeigt den Prozess, in dem alle Knoten auf dem Kreis unterteilt werden. Eine beispielhafte Darstellung von 12 Knoten im Grapheditor wird in Abbildung 4.11 gezeigt.

```
1  int nodesNo = nodes.size();
2  int i=0;
3  double x,y, angle;
4  for(Iterator itrNode = nodes.iterator(); itrNode.hasNext();){
5
6    Node node = (Node) itrNode.next();
7    angle = 2.0 * i * Math.PI / nodesNo;
8    x = 600 + 360 * Math.cos(angle);
9    y = 400 + 360 * Math.sin(angle);
10   //update the coordinates
11   node.setX(x);
12   node.setY(y);
13
14   i++;
15 }
```

Listing 4.16: Radiale Verteilung

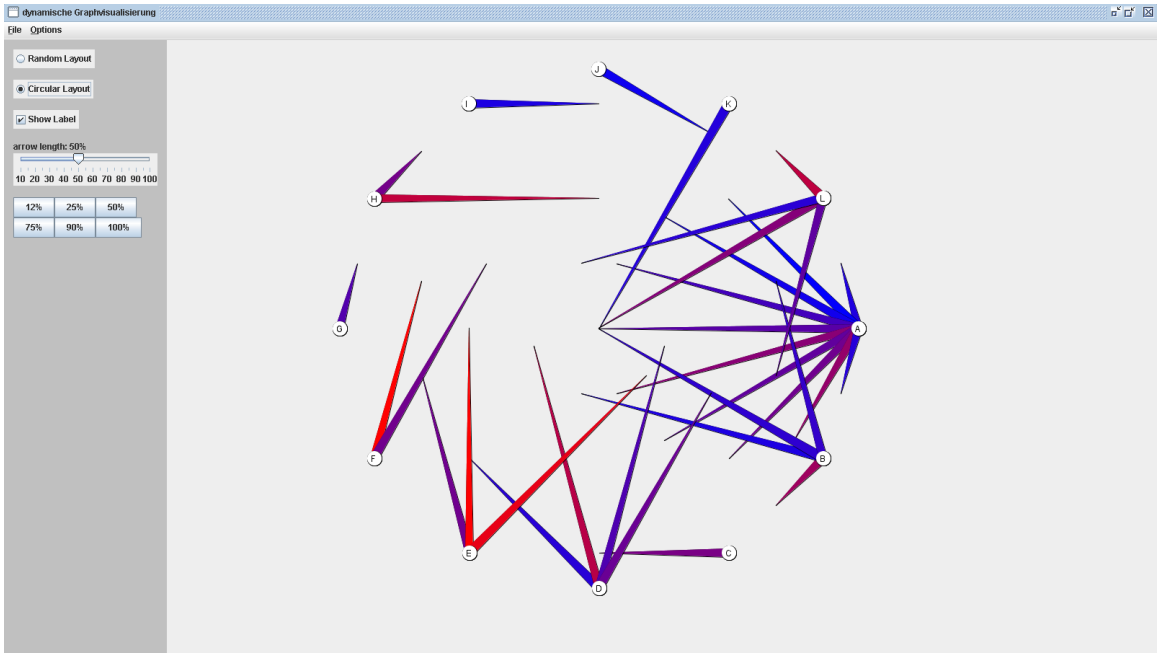


Abbildung 4.11: Radiale Verteilung von 12 Knoten: Die 12 Knoten werden gleichmäßig auf dem Kreis platziert und die Position jedes Knotens bleibt in diesem Fall fest.

5 Fallstudie

Im vorausgegangen Kapitel wurden der Entwicklungsprozess des Graphvisualisierungswerkzeugs ausführlich beschrieben. Um die Leistungsfähigkeit des Visualisierungswerkzeugs zu testen, wird im Rahmen einer Fallstudie ein dynamischer gerichteter und gewichteter Graph mittels des Werkzeuges dargestellt.

```
Cobertura/de
Cobertura/de/kclee
Cobertura/de/kclee/clemens
Cobertura/de/kclee/clemens/cclutil
   ⋮
Cobertura/net/sourceforge/cobertura/util/TypeHelper
Cobertura/net/sourceforge/cobertura/util/Version
Cobertura/net/sourceforge/cobertura/webapp
Cobertura/net/sourceforge/cobertura/webapp/FlushCoberturaServlet

11 12 3.0
11 13 2.0
11 15 2.0
11 19 3.0
   ⋮
154 71 1.0
158 137 1.0
161 12 1.0
161 23 1.0
161 143 1.0

0 1 1.0
0 2 1.0
0 3 1.0
0 4 1.0
0 5 1.0
   ⋮
127 100 1.0
127 101 1.0
127 102 1.0
127 124 2.0
152 56 2.0

48 50 2.0
48 53 1.0
48 54 1.0
48 56 1.0
48 59 1.0
   ⋮
74 59 1.0
74 60 1.0
74 61 1.0
74 62 1.0
```

Abbildung 5.1: Testdatei: Migration.bgraph

In der Fallstudie wird eine relationale Datei untersucht. Es geht um die Migrationsdaten, das heißt, wie viele Menschen von einem Land in ein anderes pro 10 Jahre auswandert. Der Dateneinsatz ist viel groß und soll klein gemacht werden, um die Beziehungen und die

Änderungen deutlich zu zeigen. Hier wird zuerst nur ein Land „China“ davon ausgewählt. Das bedeutet, dass die Informationen die Tendenz der Migration von Chinesen in der letzten Jahrzehnten darstellen sollen. Später werden ein paar weitere Länder (Angola, Australien, Großbritannien, Deutschland und USA) separat ausgewählt und dann die Dateien werden zusammengesetzt, um die Tendenz der Migration in diesen Länder darzustellen. Wenn alle Länder ausgewählt sind, ist die Zusammensetzung die originale Migrationsdatei.

Die Migrationsdaten sind in eine BGraph-Datei gespeichert. Abbildung 5.1 zeigt die Inhalte der Migrationsdatei. Es gibt insgesamt 254 Zeilen im Prolog der Datei und 5 Zeitabschnitten. Der Prolog vorne sind die Labels in der gleichen Reihenfolge wie die Nummern der Knoten. Das bedeutet, dass es 254 Länder oder Regionen gibt. Und die fünf Zeitabschnitte sind $1960 - 1969, 1970 - 1979, 1980 - 1989, 1990 - 1999$ und $2000 - 2009$.

In der erste Datei sind alle Startknoten derselbe Knoten, und zwar das Land China. Jede Kante zeigt einen Auszug von China nach einem anderen Land, mit dem Kantengewicht die Anzahl der Migration in den entsprechenden Jahrzehnten.

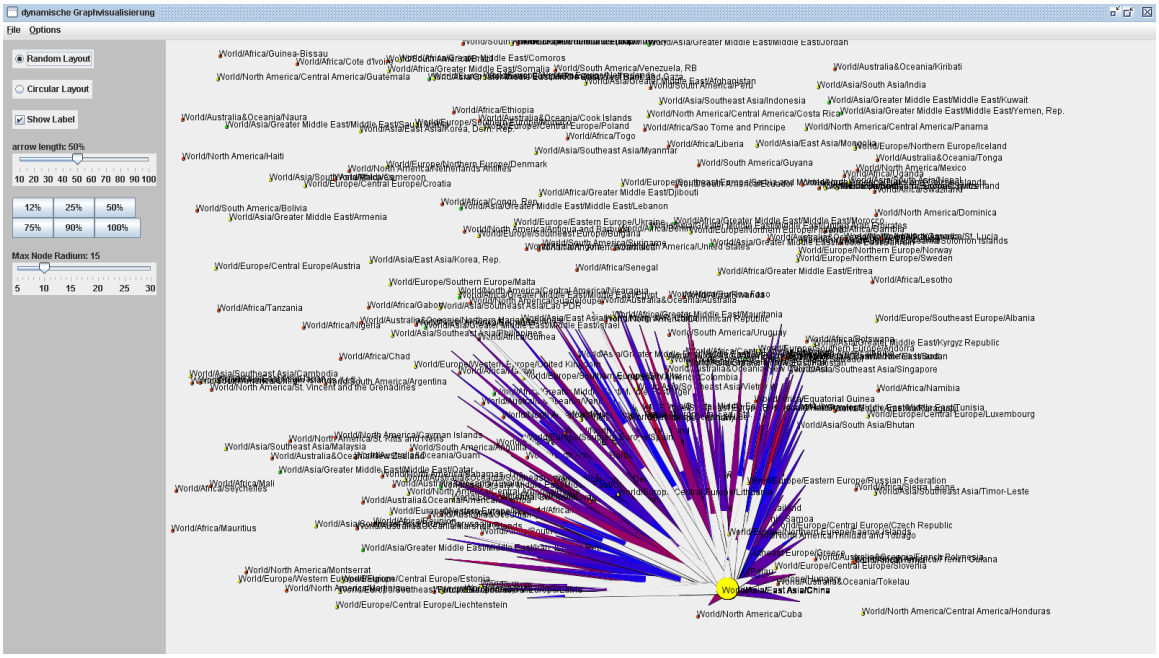


Abbildung 5.2: Zufällige Verteilung mit Label: Die Positionen der Knoten werden zufällig generiert und die Labels sind auch dargestellt.

Nach dem Einlesen der Migrationsdatei wird zuerst eine zufällige Verteilung von Knoten dargestellt (siehe Abbildung 5.2). Jeder gerichteter Kante entspricht genau die Beziehung der Migration von China nach anderen Ländern. Dann ist die Visualisierung der Relation schon realisiert. In Abbildung 5.2 gibt es viele Labels, die die Beobachtung von Kanten behindern. Dann kann die Anzeige für Label ausgeschaltet werden.

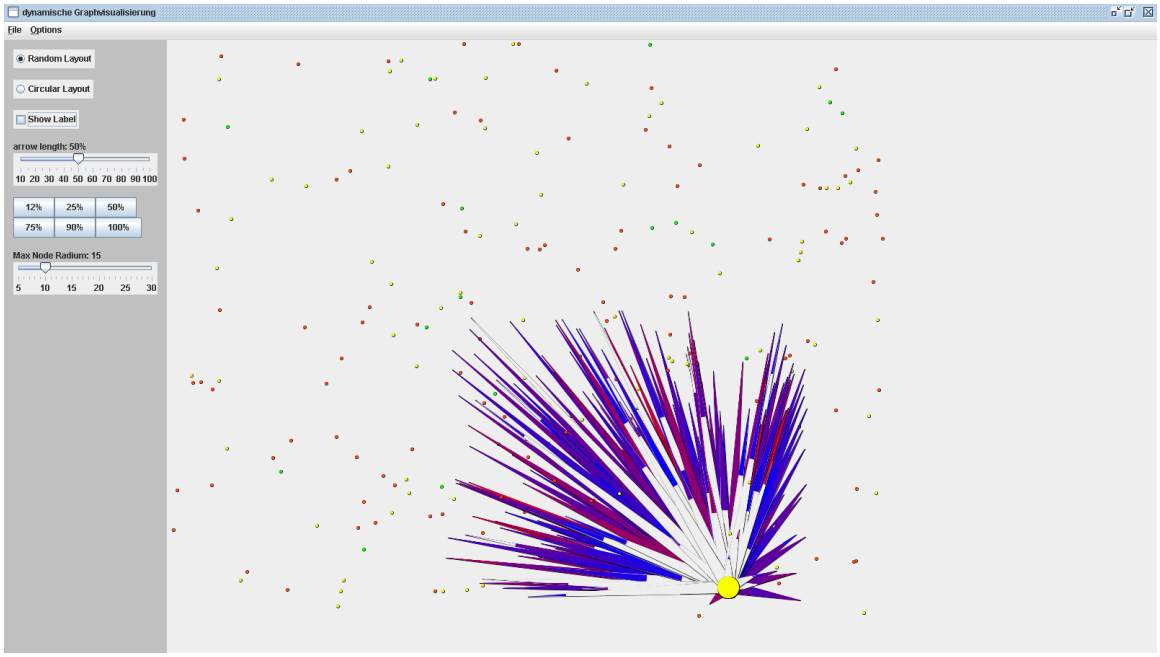


Abbildung 5.3: Zufällige Verteilung ohne Label: Nur der Graph mit Knoten und Kanten wird auf dem Bildschirm gezeichnet. Die Labels für Knoten kommen im Graph nicht vor.

Abbildung 5.3 zeigt denselben Graph aber ohne Label wie Abbildung 5.2. Nun wird die radiale Verteilung von Knoten verwendet, um die Störungen von Kanten mit einander zu reduzieren. Dann wird der Graph wie in Abbildung 5.4 gezeigt. Die Kantenlänge können eingestellt werden. Abbildung 5.5 zeigt denselben Graph wie Abbildung 5.4, aber die Länge der partiell gekennzeichneten Kanten ist von 50% auf 75% verlängert. Dann sind die Farbe von Kanten mehr deutlich als vorher und damit sind ein paar Kanten sehr aufmerksam, die helle besonderes rote Farben besitzen.

Wenn man mit der Maus einen Knoten zieht, wird der Graph auch aktualisiert. In Abbildung 5.6 werden zwei Kanten mit attraktiven Farben nach den Ziehungen von Knoten nach vorne gebracht. Die Knoten können weiter gezogen werden, um die richtige Position zu finden, damit können die gesuchten Kanten vor den Augen gelegt werden. Doppeltklickt auf den Knoten, wird die Anzeige des Labels für diesen Knoten ein- oder ausgeschaltet. Abbildung 5.7 zeigt eine Variante vom Graph in Abbildung 5.6, mit drei Label für Knoten darauf.

Durch die Änderung der Farben von Kanten kann sofort festgestellt, dass die Anzahl der Migration von China nach Indonesien immer sich verringert. In Gegensatz steigt die Anzahl der Migration von China nach Hong Kong, mit der Ausnahme in den Jahrzehnten von 1970 bis 1979.

Die Untersuchung für andere Länder können auf diese Weise durchgeführt. Wenn die Migrationsinformationen von sechs Länder in einer Datei zusammengesetzt, ist der Graph

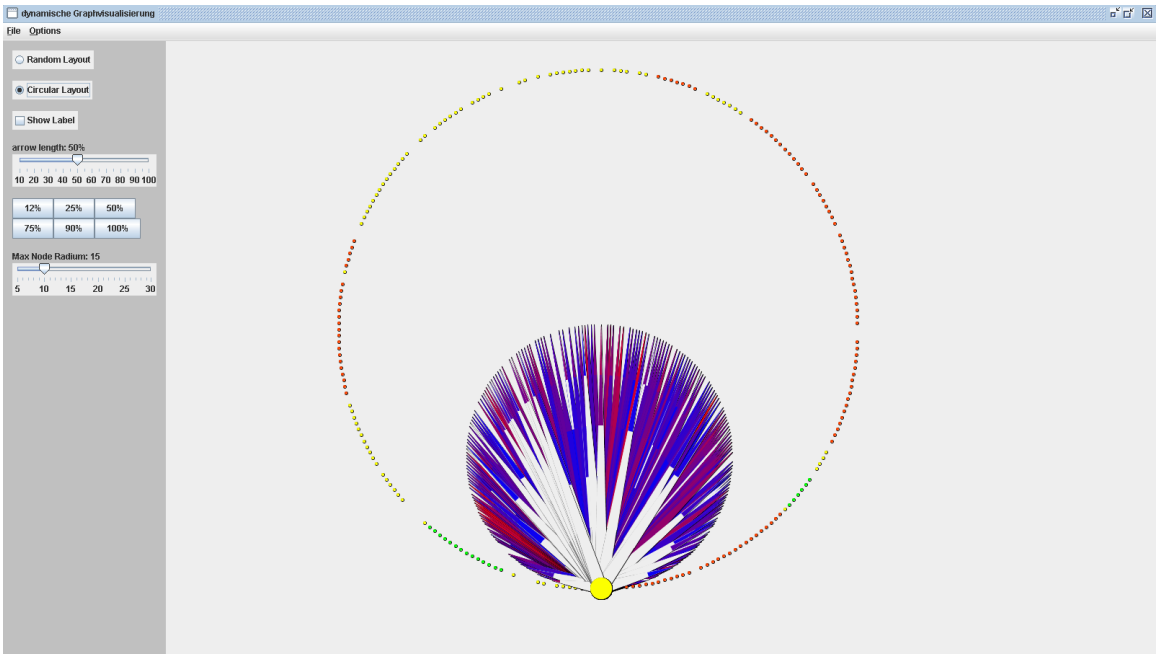


Abbildung 5.4: Radiale Verteilung: Die Positionen der Knoten werden nach Anzahl der Knoten berechnet und bleiben in der radialen Verteilung fest. Die Kantenlänge ist zur Zeit auf 50% eingestellt.

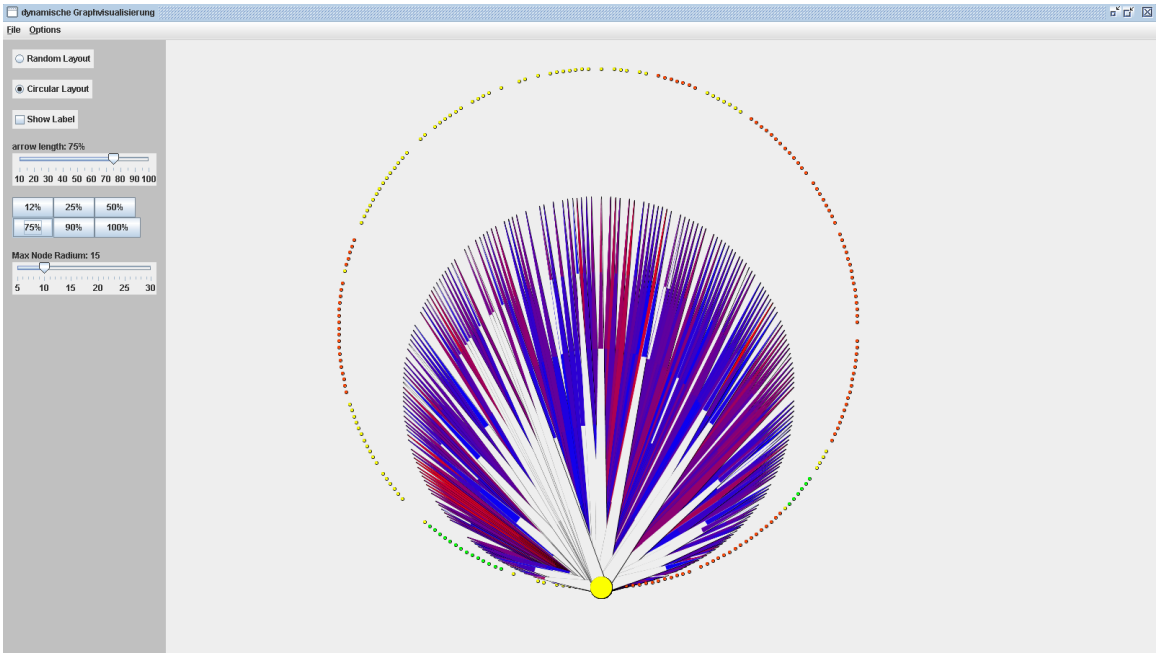


Abbildung 5.5: Radiale Verteilung: Die Kantenlänge wird auf 75% eingestellt.

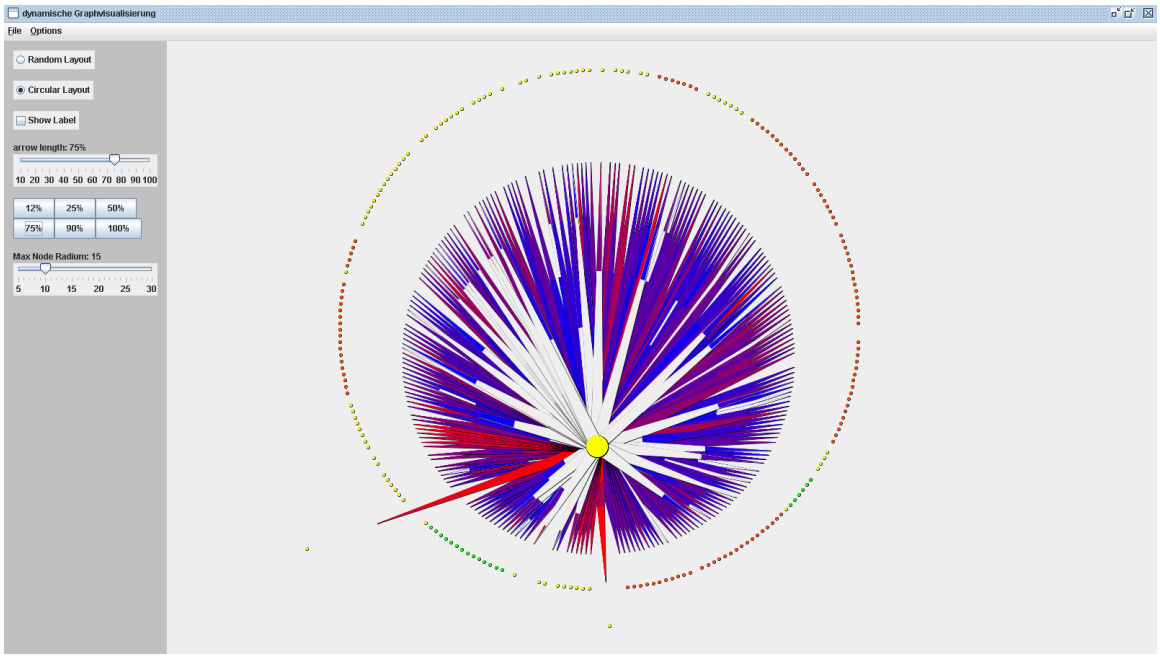


Abbildung 5.6: Ziehung der Knoten: Alle Knoten können jeder Zeit vom Benutzer mit der Maus gezogen werden. Die entsprechenden Kanten werden dann verlängert oder verkürzt. Aber die Skala ist genau wie die Einstellung im Bedienfeld.

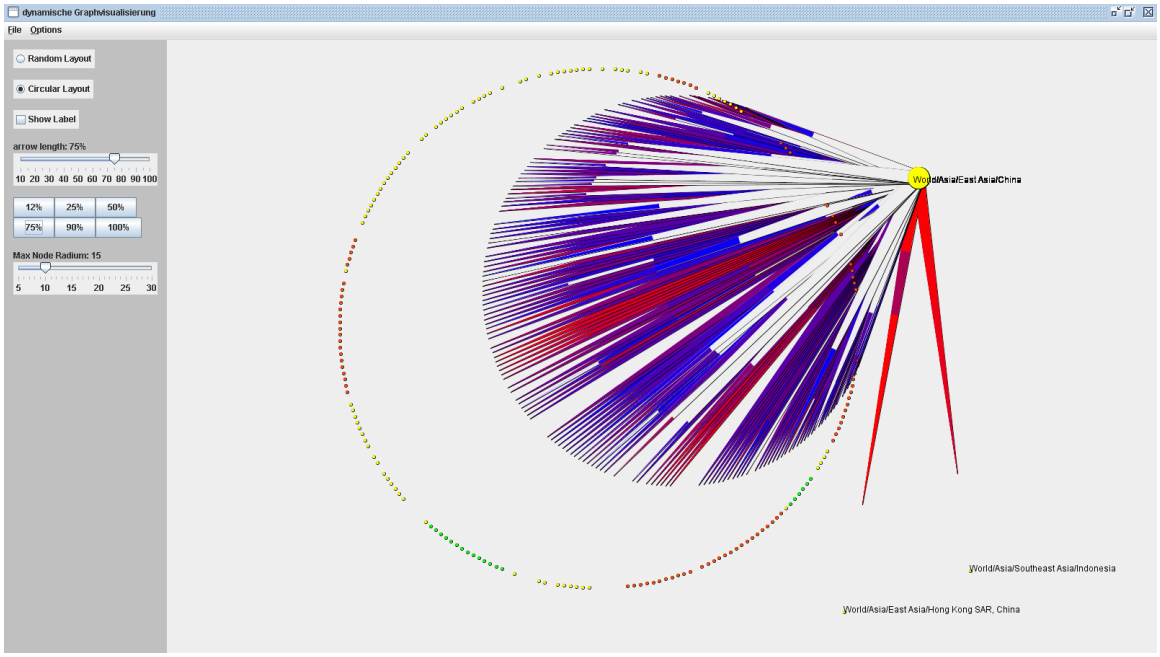


Abbildung 5.7: Ziehung der Knoten: Nur die gewünschten Labels der Knoten werden auf dem Graph gezeichnet.

nicht so einfach wie vor. Viele Kreuzungen werden produziert und dann zu Visual Clutter führen (siehe Abbildung 5.8).

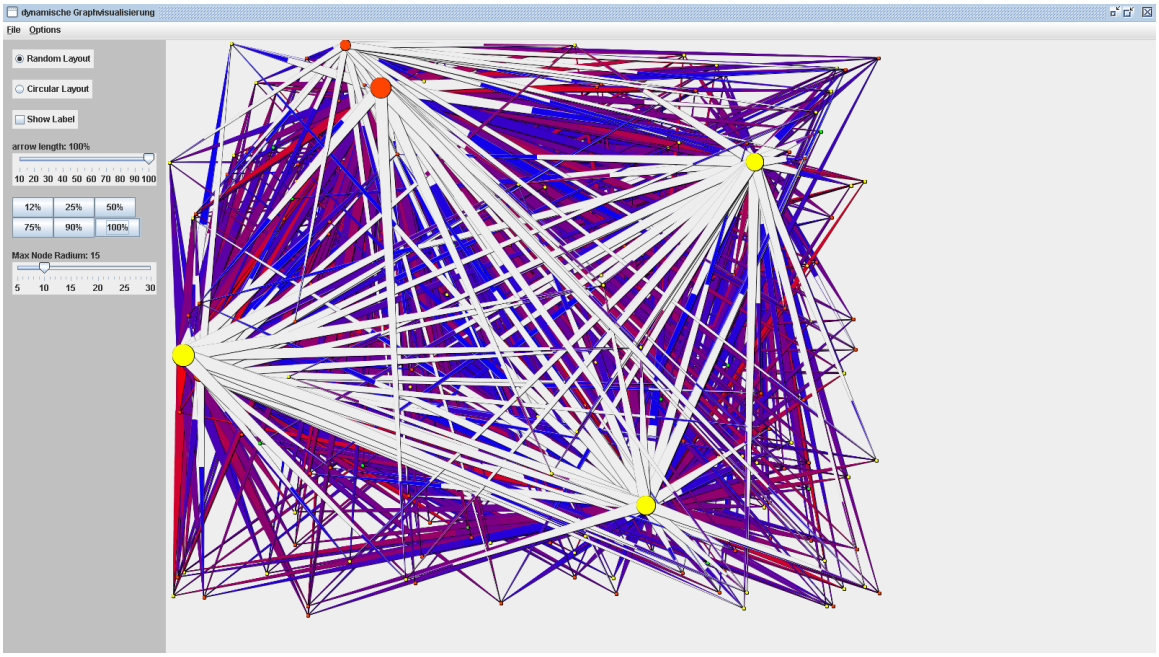


Abbildung 5.8: Migration von sechs Ländern: Der Graph verwendet eine zufällige Verteilung mit vollständiger Kantenlänge. Es gibt hier zahlreiche Kreuzungen und Überlappungen.

Durch die Anwendung der radialen Verteilung werden alle Knoten ordentlich platziert und die „Visual Clutter“ Wirkung wird dadurch abgeschwächt (siehe Abbildung 5.9). Stellt die Kantenlänge mit dem Schiebregler ein, können die Kreuzungen stark reduziert werden. In Abbildung 5.10 werden die sechs zu untersuchenden Länder viel mehr deutlich als in Abbildung 5.9 gezeigt.

In der Fallstudie ist das gezeigt, dass das Visualisierungswerkzeug bei der Untersuchung von Relation mit vielen Funktionalitäten unterstützt. Nicht nur die Visualisierung der Daten, sondern auch die weitere Verarbeitung der Daten können darin realisiert werden. Die Darstellung von Kantengewichten spielt eine wichtige Rolle, weil die Farbveränderung immer zuerst bemerkt werden kann. Die radiale Verteilung und die Einstellung der Kantenlänge dienen dazu, um die Kreuzungen besonders die dadurch entstehende „Visual Clutter“ Wirkung zu reduzieren.

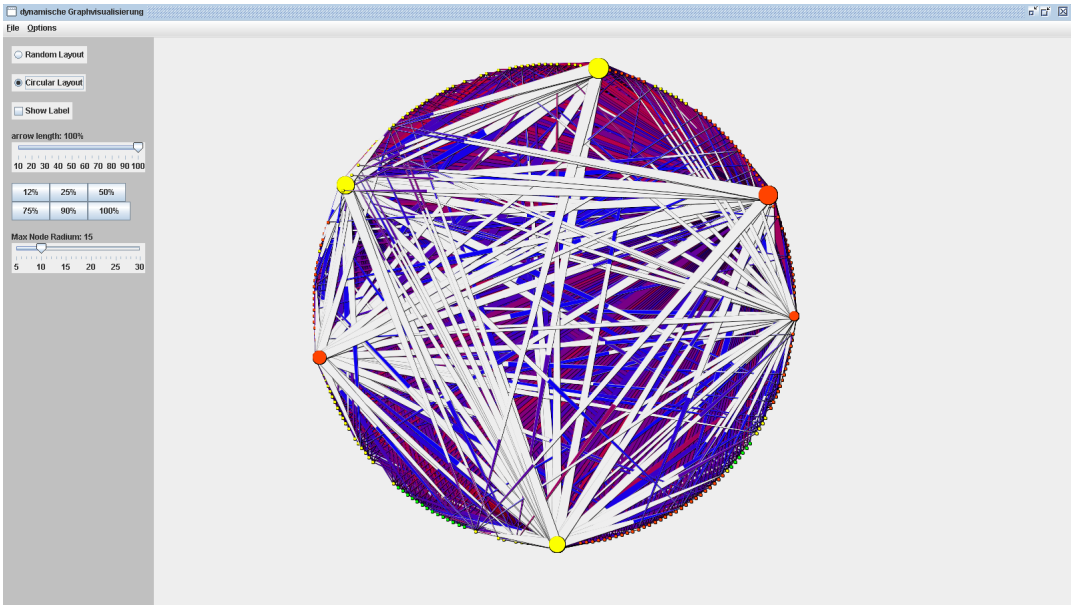


Abbildung 5.9: Migration von sechs Ländern: Der Graph verwendet die radiale Verteilung mit vollständiger Kantenlänge. Die Beziehungen von den Knoten wird auch durch Kreuzungen gedeckt.

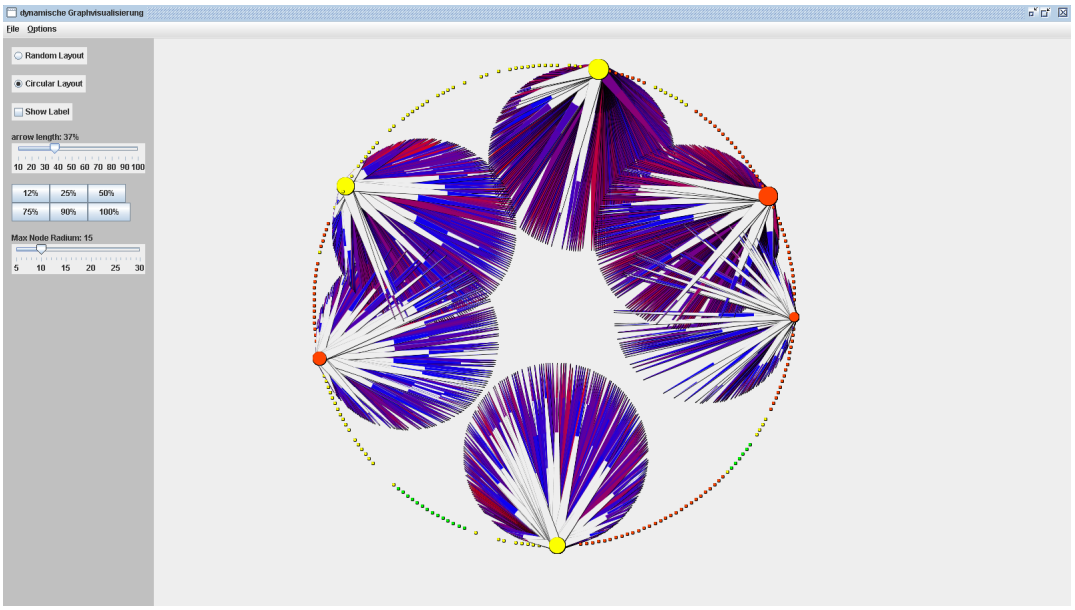


Abbildung 5.10: Migration von sechs Ländern: Durch die Einstellung der Kantenlänge auf 37% wird der Graph, insbesondere die sechs wichtigen Knoten, mehr deutlich gezeigt.

6 Zusammenfassung und Ausblick

Als Abschluss wird die Diplomarbeit nochmal zusammengefasst und dann ein Ausblick für die weitere Entwicklung ausgegeben.

6.1 Zusammenfassung

In der vorliegenden Arbeit wird ein interaktives Graphvisualisierungswerkzeug entwickelt. Dieses Visualisierungswerkzeug dient zur Untersuchung der Darstellung dynamischer gerichteter und gewichteter Graphen. Beim Knoten-Kanten Diagramm führen meist viele Kantenkreuzungen zu Visual Clutter. Partiell gezeichnete Kanten bieten dafür eine elegante Lösung, damit die Kantenkreuzungen deutlich reduziert werden. Zudem werden dynamische Katengewichte dadurch in ein Knoten-Kanten Diagramm integriert, dass die partiellen Kanten in farbkodierte Teilstücke untergliedert werden, indem alle Kanten mit einer Zeitachse versehen werden und der Gewichtsverlauf somit immer vom Start- zum Zielknoten intuitiv verstanden werden kann.

Das Graphvisualisierungswerkzeug besteht hauptsächlich aus zwei Komponenten, nämlich dem Bedienfeld und der Canvas. Das Bedienfeld ist nur für die Steuerungen zuständig. Der Canvas ist für die graphische Darstellung der Daten bzw. für die Interaktion zwischen dem Benutzer und dem Programm zuständig. Als Voraussetzung des Programms dient das Einlesen von Knoten-Kanten-Dateien. Nur die Dateien, die in Abschnitt 3.1.1 dargestellt sind, können vom Programm einwandfrei eingelesen werden.

6.2 Ausblick

Das Graphvisualisierungswerkzeug stellt ein paar Funktionalitäten zur Verfügung, um die Untersuchung der relationalen gerichteten und gewichteten Daten zu unterstützen. Je mehr Funktionalitäten das Programm realisiert, desto mehr Beitrag zur Untersuchung solcher dynamischer Graphdaten gibt es. Die weiteren Funktionalitäten, wie zum Beispiel, die Skala des gesamten Graphs, die Auswahl der Kanten nach Kantengewichten und die Auswahl des Teilgraphs, können in Zukunft implementiert und ins Programm hinzugefügt werden.

Dennoch ist zu bedenken, dass der Bedarf des Benutzers in ersten Linie besteht. Jede neu entwickelte Funktionalität soll den Anforderungen des Benutzers entsprechen. Außerdem steigt die Komplexität des Programms, wenn immer mehr Funktionen hinzugefügt werden.

Die Robustheit des Programms soll immer sorgfältig gewährleistet werden. Die Fehlerbehandlungsmechanismen können auch in Zukunft im Programm eingeführt werden. Damit würde die Leistung des Programms verbessert.

Literaturverzeichnis

[ann] Annotion in JAVA. URL http://de.wikipedia.org/wiki/Annotation_(Java). (Zitiert auf Seite 39)

[ant] Java: Antialiasing. URL http://www.leepoint.net/notes-java/GUI-lowlevel/graphics/graphics2D/65aliasing.html. (Zitiert auf Seite 39)

[BBE+95] R. A. Becker, R. A. Becker, S. G. Eick, S. G. Eick, A. R. Wilks, A. R. Wilks. Visualizing Network Data. *IEEE Transactions on Visualization and Computer Graphics*, 1:16–28, 1995. (Zitiert auf den Seiten 19 und 20)

[Bed] Benutzeroberflaeche (GUI, Graphical User Interface). URL http://www2.f1.htw-berlin.de/scheibl/netMF/index.htm?./GUI/Panel.htm. (Zitiert auf Seite 44)

[BVKW11] M. Burch, C. Vehlow, N. Konevtsova, D. Weiskopf. Evaluating partially drawn links for directed graph edges. In *Proceedings of the 19th international conference on Graph Drawing*, GD'11, S. 226–237. Springer-Verlag, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-25878-7_22. URL http://dx.doi.org/10.1007/978-3-642-25878-7_22. (Zitiert auf Seite 19)

[HW09] D. Holten, J. J. van Wijk. A user study on visualizing directed edges in graphs. In *CHI*, S. 2299–2308. ACM, 2009. (Zitiert auf den Seiten 12 und 21)

[Jav] Java (Programmiersprache). URL http://de.wikipedia.org/wiki/Java_(Programmiersprache). (Zitiert auf Seite 33)

[PD10] B. Preim, R. Dachselt. *Interaktive Systeme. Band 1: Grundlagen, Graphical User Interfaces, Informationsvisualisierung*. Springer, Berlin, 2010. (Zitiert auf den Seiten 15 und 19)

[RGB] Farben und Farbmodelle. URL http://www.dpunkt.de/java/Programmieren_mit_Java/Grafikprogrammierung/14.html. (Zitiert auf Seite 42)

[Zep04] K. Zeppenfeld. *Lehrbuch der Grafikprogrammierung: Grundlagen, Programmierung, Anwendung*. Spektrum, Heidelberg, 2004. (Zitiert auf Seite 15)

Alle URLs wurden zuletzt am 03. 12. 2012 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Xiaobo Gan)