Visualisierungsinstitut der Universität Stuttgart
VISUS building (Allmandring 19)
70569 Stuttgart

Diplomarbeit Nr. 3531

Die Visualisierung dynamischer Graphen als Small Multiples

Siguang Liang

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Daniel Weiskopf
Betreuer/in:	Dr. Michael Burch
Beginn am:	1. August 2013
Beendet am:	31. Januar 2014
CR-Nummer:	H.5.2, I.3.3

Kurzfassung

In der Entwicklung und Forschung im Gebiet der Informatik gibt es viele unterschiedliche abstrakte und schwer verständliche Daten. Solche Daten und ihre Beziehungen sind schwierig zu verstehen und zu analysieren, wenn man nur die textuelle Form betrachtet. Normalerweise werden solche Daten zur graphischen Repräsentation, d.h. in eine Visualisierung umgewandelt. In dieser Diplomarbeit werden dynamische relationale Daten mit Hilfe einer dynamischen Graphvisualisierung explorierbar gemacht. Dabei stützen wir uns auf das bgraph-Format. Diese Dateien enthalten die Informationen, die dynamische Graphen beschreiben. Ein dynamischer Graph ist eine Folge von statischen Graphen, der die wiederholten Modifikationen eines ursprünglichen Graphen ausdrückt und sich im Lauf der Zeit verändert. Aber diese Informationen über die dynamischen Graphen liegen in abstrakter und schwer zu lesender Text-Form vor. In dieser Diplomarbeit wird ein Programm in der Programmiersprache JAVA entwickelt, das solche Dateien im bgraph-Format einlesen und darstellen kann. Es wird eine Small Multiples Darstellung für die dynamischen Graphen gewählt, während jeder einzelne statische Graph bipartite oder radial dargestellt wird. Um Visual Clutter zu reduzieren, wird die Edge Splatting Technik angewendet. Mit Interaktion kann der Benutzer die Änderungen der dynamischen Graphen gut überblicken und analysieren. In mehreren Fallstudien wird die Nützlichkeit der Technik und des Visualisierungswerkzeuges illustriert. Hier werden dynamische Graphen aus dem Bereich der Softwareentwicklung betrachtet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Einleitung

Relationale Daten treten in vielen Anwendungsgebieten auf. So lassen sich etwa Freundschaftsbeziehungen in sozialen Netzwerken als gerichtete oder ungerichtete Graphen modellieren. Auch Aufrufgraphen in Softwaresystemen stellen gerichtete Graphen dar. Im Bereich der Biologie interagieren Proteine untereinander, was auch zu einem Graphenmodell führt.

Um solche Daten besser und schneller verstehen zu können, bieten sich Visualisierungstechniken an. Diese nutzen die Stärken des menschlichen visuellen Systems und dessen perzeptuelle Fähigkeiten in besonderem Maße aus. Die gewählte Visualisierungstechnik für solche Daten spielt hierbei aber eine entscheidende Rolle. Nicht jede Technik ist gleichermaßen geeignet, um Graphdaten zu interpretieren.

Als visuelle Darstellung von relationalen Daten bieten sich in etwa Knoten-Kanten Diagramme, Adjazenzmatrizen oder Adjazenzlisten an. Diese drei visuellen Metaphern gelten als die traditionellsten. Hierbei kann man sagen, dass die ersten beiden mit Sicherheit am Häufigsten verwendet werden. Adjazenzmatrizen erlauben zwar die Darstellung besonders dichter Graphen, sie haben aber Nachteile bei pfadbezogenen Aufgaben. Knoten-Kanten Diagramme helfen dem Betrachter eher, Pfade zu lesen, aber gerade bei größeren Graphen wird diese Aufgabe auch hier schwierig zu lösen.

In dieser Diplomarbeit nutzen wir zwar Knoten-Kanten Diagramme für die Darstellung statischer Graphen, wir wenden allerdings zusätzlich das Prinzip des Edge Splatting an, um Visual Clutter zu reduzieren, der durch viele Kantenkreuzungen entsteht. Es werden darüberhinaus auch mehrere Graphlayouts getestet, um zu überprüfen, ob es generelle Unterschiede bei der Erkennung von visuellen Mustern in verschiedenen Layouts gibt.

Graphen sind nicht immer statisch. In vielen Anwendungsgebieten verändern sich Graphen über die Zeit. Beispielsweise kommen in sozialen Netzwerken Relationen hinzu, während andere wieder verschwinden. Betrachtet man solch eine Evolution von Graphen, dann spricht man von einem dynamischen Graphen.

Im Rahmen dieser Diplomarbeit wird ein Visualisierungswerkzeug implementiert und beschrieben, welches dynamische Graphen einlesen kann und diese dann als Small Multiples Darstellung visualisieren kann. Mit Interaktionstechniken lassen sich die Daten dann vom Benutzer aggregieren, browsen, filtern und letztlich explorieren.

Im Rahmen dieser Diplomarbeit soll ein interaktives Visualisierungswerkzeug entwickelt werden, das eine Folge von Graphen in einer Small Multiples Darstellung zeigt. Die Graphen sollen zeitlich aggregiert werden können und die Interaktion soll Browsing durch die Zeit unterstützen. Dazu sollen:

I zunächst dynamische Graphen eingelesen werden können,
II solche Daten als Small Multiples dargestellt werden können,
III interaktive graphbezogene Features wie etwa Aggregation und Browsing implementiert werden,
IV weitere Interaktionstechniken eingebaut werden,
V mit weiteren Kantendarstellungen wie etwa Edge Bundling, Edge Splatting und Partial Edges experimentiert werden,
VI eine Fallstudie mit großen dynamischen Graphdaten durchgeführt werden,
VII eine Fallstudie als Test der Bedienbarkeit und der Verständlichkeit durchgeführt werden.

2 Verwandte Arbeiten

„Graph“ ist ein mathematischer Begriff und ist das Hauptforschungsobjekt der Graphentheorie, die wiederum ein Teilgebiet der Mathematik ist. Mit Graphen lassen sich Relationen modellieren. Solche Relationen - ob gewichtet, ungewichtet, gerichtet oder ungerichtet - spiegeln die paarweise Struktur zwischen Objekten wieder.

Der Begriff „Graph“ wird jedoch nicht nur in der Mathematik angewandt, sondern auch in den Gebieten wie etwa der Chemie, der Physik, der Biologie, der Geographie und insbesondere der Informatik. Fast alle Datenstrukturen können als Graph repräsentiert werden. Darüber hinaus kommt der Begriff Graph auch in vielen Teilgebieten der Informatik vor, z.B. bei Netzwerken in der Kommunikation, bei der Datenorganisation, dem Strom der Berechnung usw. In dieser Diplomarbeit wird das Thema des einzelnen statischen Graphen betrachtet und darauf aufbauend das des dynamischen Graphen.

2.1 Statische Graphen

Ein einzelner Graph ist ein allgemeiner Graph, der auch als „statisch“ bezeichnet wird. Ein Graph ist eine abstrakte Struktur, die eine Menge von Objekten zusammen mit den zwischen diesen Objekten bestehenden Verbindungen repräsentiert. Die Abstraktionen der Objekte heißen „Knoten“. Die paarweisen Verbindungen zwischen Knoten werden dabei „Kanten“ genannt. Häufig werden Graphen anschaulich gezeichnet, indem die Knoten durch Punkte und die Kanten durch Linien dargestellt werden. Hierbei kann man die Darstellung der Punkte variieren, d.h. es können Kreise, Dreiecke oder Quadrate als Formen gewählt werden. Auch bei den Kanten kann man auf einfache gerade, gebogene oder orthogonale Linien zurückgreifen. Darüberhinaus können diese auch zugespitzt dargestellt sein wie in der Evaluation von Holten et al [HIWF11, HW09] beschrieben.

Es gibt einige Kategorien von Graphen. Die in der Arbeit auftretenden und behandelten Graphtypen werden unten vorgestellt:

I Ein einfacher Graph G ist ein Tupel (V, E), wobei V eine endliche Menge von Knoten und E eine endliche Menge von Kanten bezeichnet. Dabei ist E eine Teilmenge aller 2-elementigen Teilmengen von V. Zum Beispiel ist (u,v) eine Kante, wobei u und v zwei Knoten aus der Knotenmenge V sind.

II Ein gerichteter Graph (von englisch: directed graph oder digraph) ist auch ein Tupel (V, E). Aber E ist ein geordnetes Knotenpaar, das gerichtete Kante genannt wird. Zum Beispiel bedeutet (u,v) eine gerichtete Kante von u zu v, wobei u und v zu der Knotenmenge V gehören. u heißt Startknoten, v wird Endknoten genannt.

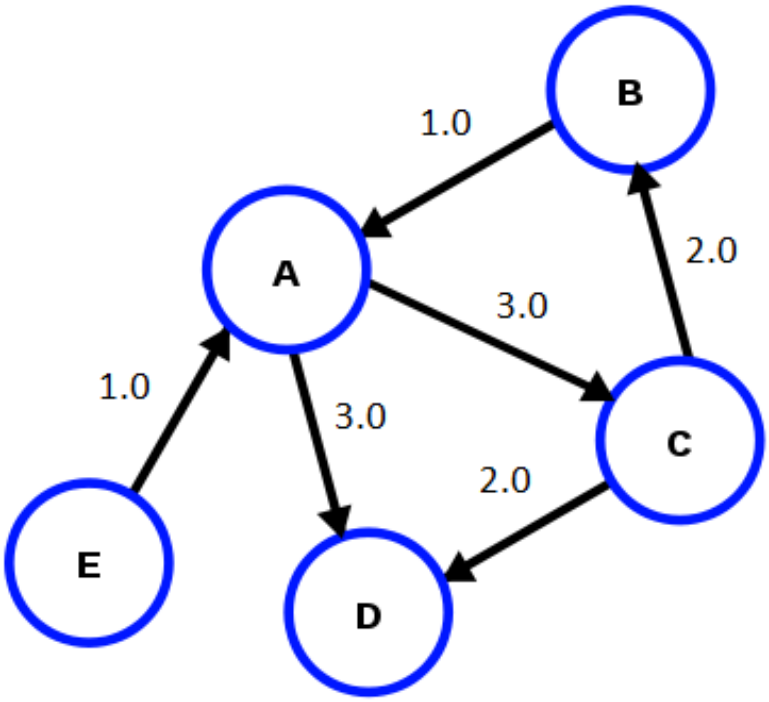


Abbildung 2.1: Ein Beispiel eines gerichteten und kantengewichteten Graphen.

III Ein kantengewichteter Graph, kurz gewichteter Graph, ist auch ein Graph, in dem jeder Kante eine reelle Zahl als Kantengewicht zugeordnet ist. Falls die Knoten eines Graphen gewichtet sind, heißt er knotengewichteter Graph. In dieser Diplomarbeit werden nur kantengewichtete Graphen betrachtet. Ein gewichteter Graph wird als Tupel (V, E, f) dargestellt. f ist eine Funktion von E nach N, wobei N die Menge von Gewichten ist.

In dieser Diplomarbeit sind alle Graphen gerichtete, kantengewichtete Graphen. Die Abbildung 2.1 zeigt ein Beispiel eines gerichteten, kantengewichteten Graphen.

Gerade im Bereich der Visualisierung von statischen Graphen existieren zahlreiche Arbeiten. Im Allgemeinen zielen Graphvisualisierungstechniken darauf ab, lesbare, interpretierbare und ästhetisch gut aussehende Layouts für den Betrachter zu generieren. Ein guter Überblick wird in [DBETT99] gegeben. Auch im Rahmen dieser Diplomarbeit versuchen wir Layouts zu erzeugen, die ästhetischen Graphzeichenkriterien entsprechen [BEW95]. Dies wird allerdings immer schwieriger, wenn die Daten größer werden und auch dynamische Graphen in Betracht gezogen werden.

Die meisten Ansätze versuchen Visual Clutter [RLMJ05] zu reduzieren wie etwa die partiellen Kanten [BFBD10]. Aber auch bestimmte Layoutalgorithmen wie etwa hierarchisches [BVB+11], radiales [BD08] oder kräftebasiertes [Ead84, FR91, KK89] Layout können ohne Kantenreduktion gute Ergebnisse erzielen. Zirkuläres Layout wird auch in dieser Diplomarbeit als eine Alternative betrachtet.

Gerade im Bereich der ästhetischen Kriterien wurden von Purchase et al. [Pur97, PCA02, PCJ96] viele empirische Studien durchgeführt. In ihrer ersten Studie erforschten sie Effekte wie etwa die Symmetrie, Kantenkreuzungen and Kantenknicke als Lesbarkeitskriterien. Sie berichteten, dass die Minimierung von Kantenknickstellen und Kantenkreuzungen die Performanz von Betrachtern stark verbessert. Die Minimierung von Kantenkreuzungen ist hierbei das stärkere Kriterium.

Ware et al. [WPCM02] fand beispielsweise heraus, dass die Kontinuität ebenfalls ein entscheidender Faktor für die Ästhetik ist. Der Winkel zwischen Kantenkreuzungen ist ebenso entscheidend wie die Kreuzung selbst. Diese Ergebnisse wurden durch Eye Tracking Studien von Huang et al. [HE05, HHE05] unterstützt. Sie zeigen, dass kleine Winkel die Augenbewegungen verlangsamen.

Holten et al. [HIWF11, HW09] führte einige Studien durch, um die Performanz verschiedener Kantendarstellungen zu evaluieren. Ihre Ergebnisse zeigten, dass es einen signifikanten Unterschied bezüglich der Vorteile von Tapered-Darstellungen gegenüber Standardpfeilspitzen gibt. In dieser Diplomarbeit werden einerseits die Leserichtung (von links nach rechts) als Indiz für die Kantenrichtung genutzt und andererseits wird Visual Clutter durch Edge Splatting Techniken reduziert.

Abgesehen von Knoten-Kanten Diagrammen lassen sich Graphen auch als Adjazenzmatrizen darstellen. Ghoniem et al. [GFC04] haben in einer Studie herausgefunden, dass Adjazenzmatrizen besser zur Exploration dichter Graphen geeignet sind. Allerdings werden pfad-bezogene Aufgaben schlechter bearbeitet, zumindest für Graphen mit wenigen Knoten. In dieser Diplomarbeit wollen wir nur Knoten-Kanten Diagramme betrachten.

2.2 Dynamischer Graph

Die oben erwähnten Graphen sind „statisch“. Aber die Anwendungen von Graphen in den Gebieten Interaktion, Visualisierung und Algorithmenanimation beziehen sich oftmals auf die „dynamischen“ Graphen. In diesem Gebiet verändern sich fast alle Datenstrukturen (als Graph beschrieben), wenn das Programm ausgeführt wird. Unter dynamischem Graph versteht man eine Abfolge von Graphen, die die wiederholten Modifikationen eines ursprünglichen Graphen sind und sich im Lauf der Zeit verändern, wie $G = (G_1, G_2, \ldots, G_n)$, wobei $G_1, G_2, \ldots, G_n$ eine Folge von Graphen ist. Diese Veränderungen werden durch die Interaktionen vom Benutzer, Algorithmen oder der Determinierung der darunterliegenden Prozesse verursacht.

Ein statischer Graph bezieht sich auf drei Entitäten: Knotenmege V, Kantenmenge E und Gewichtsfunktion f. Wenn irgendeine dieser drei Entitäten sich im Lauf der Zeit verändert, ist der Graph ein dynamischer Graph. Es gibt drei grundlegende Typen von dynamischen Graphen:

I Für einen Knoten-dynamischen Graph wird die Knotenmenge V geändert. Einige Knoten werden hinzugefügt oder entfernt. Bei der Entfernung der Konten werden ihre entsprechenden Kanten auch entfernt.

II Für einen Kanten-dynamischen Graph wird die Kantenmenge E geändert. Die Kanten werden hinzugefügt oder entfernt. Aber bei der Entfernung der Kanten werden die entsprechenden Knoten nicht entfernt.

III Für einen Gewicht-dynamischen Graph wird die Funktion f geändert. Das bedeutet, dass die Gewichte von Kanten geändert werden.

Alle drei Typen von Veränderungen können sich gleichzeitig ereignen. In dieser Arbeit werden die dynamischen Graphen in Bezug auf alle drei Aspekte betrachtet. Die Aggregationen und Zerlegungen von Graphen verursachen die Veränderungen von V, E und f.

Ein dynamischer Graph ist ein Graph, der sich im Laufe der Zeit verändert. Ein dynamischer Graph kann als eine Abfolge von statischen Graphen mathematisch modelliert werden. Für das Graphzeichnen (Graph Drawing) eines statischen Graphen ist es eine wichtige Aufgabe, das Graph-Layout effektiv und effizient auszurechnen. Die Betrachtungsweise in dieser Diplomarbeit wird von TimeArcTrees [GBD09] und TimeRadarTrees [BD08] inspiriert, wobei die Knoten eines Graphen sich auf zwei parallelen Linien oder auf einem Kreis verteilen.

Jede Kante zwischen den Knoten wird durch eine gerade Linie repräsentiert. Wegen der enomen Anzahl der Kanten und vieler Kreuzungen von Kanten verursacht die Visualisierung der Graphen eventuell Visual Clutter, abhängig von der Größe der Graphen und deren Dichte. Unter Visual Clutter versteht man wörtlich einen Wirrwarr in der Darstellung, der dazu führen kann, dass die Visualisierung nicht mehr lesbar ist und deshalb auch nicht mehr interpretiert werden kann. Deshalb wird die Technik „Parallel Edge Splatting“ eingeführt [BVB+11].

In dieser Methode werden Kanten in „Dichtefeldern“ umgewandelt. Zuerst wird jeder statische Graph in viele kleine Gebiete unterteilt. Mit den kleineren Gebieten kann ein Graph leicht in eine Matrix tranformiert werden. Je mehr Kanten ein Gebiet überqueren, umso größer ist das Gewicht des Gebiets. Diese Abfolge von Matrizen verändern sich auch im Laufe der Zeit, wie entsprechend wie vom dynamischen Graph gegeben. Die Matrix ist dann eine time-based Matrix, also zeitbasiert. Dann wird ein „Box Filter“ auf jede Matrix angewandt. Der Gedankengang von Parallel Edge Splatting ist es, die Repräsentation der Dichte der Kanten mit Hilfe von Splatting zu verbessern und so die Lesbarkeit der Graphdarstellung zu erhöhen. Durch solch eine Visualisierung können die Trajektorien der Kanten in einem sehr dichten Gebiet leichter erkannt werden als ohne Edge Splatting.

Neben dem Edge Splatting existieren aber auch noch weitere Darstellungsmöglichkeiten für dynamische Graphen. Die meisten davon basieren auf Knoten-Kanten Diagrammen [Ead92, DBETT99].

Problematisch wird es, wenn die darzustellenden Graphen nicht mehr dünn besetzt sind, sondern zur Klasse der dichten Graphen gehören. Dann entsteht sehr leicht das Problem des Visual Clutters [RLMJ05], was im dynamischen Fall noch verschlimmert wird.

Da Visual Clutter meistens bei Knoten-Kanten Diagrammen besteht, gibt es auch die Möglichkeit, auf Matrix-basierte Darstellungen zurückzugreifen. Solche Ansätze werden beispielsweise in den TimeRadarTrees beschrieben [BD08]. Dort wird auch eine radiale Variante genutzt.

Statische Diagramme für dynamische Graphen haben viele Vorteile gegenüber animierten Diagrammen [TMB02]. Speziell für dynamische Graphen muss man bei animierten Sequenzen immer darauf achten, dass die dynamische Stabilität und die Mental Map erhalten bleibt. Gerade bei der Graphanimation gab es in diesem Bereich viele Arbeiten [DG02, FT08].

Die dynamische Stabilität spielt eine sehr grosse Rolle bei der Visualisierung von dynamischen Graphen. Sie hilft entscheidend, kognitive Anstrengungen zu reduzieren, die der Betrachter aufbringen muss, wenn er die Graphsequenz betrachtet und versucht, Schlüsse daraus zu ziehen.

Auch bei Matrixdarstellungen für dynamische Graphen sollte dies gewährleistet sein [YEL10, BN11, GGK+11, SWS10]. Hier werden die dynamischen Kanten in die entsprechenden Matrixzellen etwa als Balkendiagramme dargestellt. Auch Interaktion mit den Visualisierungen ist hier sehr hilfreich zur Datenexploration.

3 Datenmodell

Ein Datenmodell ist ein Modell der zu beschreibenden und verarbeitenden Daten eines Anwendungsbereichs und ihrer Beziehungen zueinander. Bei der Datenmodellierung gibt es drei Eigenschaften. Erstens soll das Datenmodell die reale Welt anschaulich simulieren. Zweitens soll das Datenmodell von Menschen leicht verstanden werden. Drittens soll das Datenmodell durch den Computer leicht realisiert werden können. In dieser Arbeit basieren die Datenmodelle auf den oben genannten drei Eigenschaften.

3.1 Graph und Netzwerk

Der Knoten eines Graphen ist als ein abstrakter Begriff zu verstehen, der die verschiedenen Objekte repräsentieren kann. In dieser Arbeit werden Knoten als Pfade in einer Hierarchie beschrieben wie z.B. „root/net/sourceforge/cobertura/ant/AntUtil/AntUtil()“. Diese Form ist schwierig zu notieren und zu verstehen, also wird dies im Datenmodell zunächst beschrieben. Als Knoten der Graphen ist diese textuelle Form nicht gut geeignet. In der Arbeit ist die Reihenfolge der Knoten im Graphlayout von der Knotenreihenfolge auf der Textdatei bestimmt.

Statt „root/net/sourceforge/cobertura/ant/AntUtil/AntUtil()“ können wir die Knoten deshalb als durchnummeriert auffassen. Zum Beispiel bezieht sich die Nummer 1 auf den ersten Knoten in der Reihenfolge aller Knoten, die Nummer 2 auf den zweiten Konten, 3 auf den dritten Knoten, usw. Diese Nummern, auch Identifikatoren genannt, sind anschaulich und leicht zu verstehen und können leicht zur Weiterverarbeitung der Daten genutzt werden.

Die Beziehung zwischen zwei Knoten kann dementsprechend durch zwei solcher Nummern repräsentiert werden. Bezüglich des Gewichtes können wir eine Kante durch ein 3-Tupel beschreiben. Zum Beispiel bedeutet (1 3 1.0) eine Kante vom ersten Knoten zum dritten Knoten, deren Gewicht 1.0 ist. Ein Graph besteht aus mehreren Kanten. Das bedeutet, dass eine Gruppe solcher 3-Tupel wie (1 3 1.0) einen Graph darstellt. Es kann die drei oben erwähnten Eigenschaften des Datenmodells besitzen.

3.2 Dynamischer Graph

Ein dynamischer Graph ist eine Abfolge von einzelnen Graphen. Am Anfang sind alle Knoten aller einzelnen Graphen gegeben, die dann in den Vereinigungsgraph transformiert werden. Deshalb ist die Reihenfolge aller Knoten gleichbleibend, was später bei der Visualisierung dafür sorgt, dass die Mental Map erhalten bleibt. Alle Graphen basieren auf diesen am Anfang der

Datei gegebenen Knoten. Die Identifikatoren aller Knoten verändern sich im ganzen Prozess nicht, d.h. über diese eindeutigen Nummern lassen sich die Knoten stets auseinanderhalten und identifizieren. Deshalb wird ein dynamischer Graph einfach durch mehrere Gruppen von 3-Tupeln repräsentiert. Jede Gruppe beschreibt einen einzelnen Graph.

3.3 Eingabedatei

Die Eingabedatei ist eine „.bgraph“ Datei, die beispielsweise durch die Software „UltraEdit“ oder „WordPad“ von Windows geöffnet werden kann. Eine Eingabedatei beinhaltet viele Knoten und die Beziehungen zwischen den Knoten, d.h. die gewichteten Kanten. Alle Knoten und Beziehungen bilden einen oder mehrere Graphen. Die Abbildung 3.1 ist ein Beispiel einer Eingabedatei.

Der erste Teil der Eingabedatei besteht aus mehreren Zeilen wie etwa „root/net/sourceforge/cobertura/ant/AntUtil/AntUtil()“, „root/net/sourceforge/cobertura/ant“ usw. Jede Zeile repräsentiert einen Knoten im Graph. Aber für die konkreten Informationen der Zeilen haben wir kein Interesse, weil diese Informationen keinen Einfluss auf die Generierung der Graphen und deren Layout haben. Wenn wir die erste leere Zeile treffen, bedeutet es, dass der erste Teil über dem „Knoten“ endet, z.B. die leere Zeile in der Abbildung 3.1 zwischen „root/net/-sourceforge/cobertura/ant/CheckTask/setClasspath(org.apache.tools.ant.types.Path)“ und „#cobertura-1.0.0“. Der zweite Teil der Eingabedatei beschreibt die Beziehungen zwischen den Knoten. Diese Beziehungen bilden die Relationen eines Graphen. Die Inhalte zwischen zwei leeren Zeilen stellen jeweils einen Graph dar. Zum Beispiel gibt es vier Zeilen in der Abbildung 3.1 zwischen der ersten und der zweiten leeren Zeile. Diese vier Zeilen beschreiben somit einen einzelnen Graph. Die erste Zeile davon ist „#cobertura-1.0.0“, die eine globale Beschreibung, also der Titel dieses Graphen ist und somit auch keinen Einfluss auf die Generierung und das Layout des Graphen hat. Deshalb kümmern wir uns nicht darum.

Ab der zweiten Zeile sind die Inhalte wichtig. Die zweite Zeile ist „1 5 1.0“, die die drei Zahlen 1, 5 und 1.0 enthält. 1 bezieht sich auf den ersten Knoten, nämlich die erste Zeile in der Eingabedatei „root/net“. 5 bezieht sich auf den fünften Knoten, nämlich die fünfte Zeile in der Eingabedatei „root/net/sourceforge/cobertura/ant/AntUtil“. 1.0 bedeutet das Gewicht einer Kante. Diese Zeile beschreibt eine Kante vom ersten Knoten „root/net“ zum fünften Knoten „root/net/sourceforge/cobertura/ant/AntUtil“, deren Gewicht „1.0“ ist.

Die dritte Zeile „2 7 1.0“ ist ähnlich. Sie bedeutet eine Kante vom zweiten Knoten „root/net/-sourceforge“ zum siebten Knoten „root/net/sourceforge/cobertura/ant/CheckTask/execute()“, deren Gewicht 1.0 ist.

Die vierte Zeile ist eine Kante vom sechsten Knoten zum vierten Knoten, deren Gewicht 1.0 ist. Dann trifft man auf die zweite leere Zeile. Sie stellt das Ende des ersten Graphen dar. Danach beginnt der zweite Graph, der bis zu der nächsten leeren Zeile geht.

Bitte beachten Sie, dass der zweite Graph keinen „Titel“ hat. Nicht alle Graphen haben einen solchen „Titel“. Durch den zweiten Teil der Eingabedatei (die Informationen in Form „1 5 1.0“) werden die Graphen gezeichnet.

```
root/net
root/net/sourceforge
root/net/sourceforge/cobertura
root/net/sourceforge/cobertura/ant
root/net/sourceforge/cobertura/ant/AntUtil
root/net/sourceforge/cobertura/ant/AntUtil/AntUtil()
root/net/sourceforge/cobertura/ant/CheckTask/execute()
.....
root/net/sourceforge/cobertura/ant/CheckTask/setClasspath(org.apache.tools.ant.types.Path)

#cobertura-1.0.0
1 5 1.0
2 7 1.0
6 4 1.0

166 162 1.0
166 211 1.0
166 213 1.0

#cobertura-1.1.3

#cobertura-1.2.2
54 100 1.0
57 101 1.0
175 177 1.0
180 178 1.0
.....
```

Abbildung 3.1: Die Inhalte einer Eingabedatei wie sie in textueller Form vorliegt

Bitte beachten Sie, dass das Gewicht im Beispiel eine Zahl mit nur einer Dezimalstelle ist, wie etwa 1.0. Aber theoretisch sind alle Zahlen vom Typ „double“ für das Gewicht möglich. Zum Beispiel sind in der ersten Eingabedatei die Gewichte in der Form „1.0, 2.0. . . “ vorliegend. Vielleicht sind in der zweiten Eingabedatei die Gewichte in der Form „1.01, 2.11. . . “ vorliegend. Weil die Gewichte die Farben der Kanten in der Visualisierung beeinflussen, muss das Problem bei der Programmierung beachtet werden.

In der Abbildung 3.1 können wir sehen, dass der dritte Graph nur einen „Titel“ hat. Der Graph beinhaltet nur eine Zeile „#cobertura-1.1.3“. Oben und unten sind zwei leere Zeilen. Es ist ein „leerer“ Graph. Wenn solche leeren Graphen gelesen werden, werden ihre Startpunkte auf 0, Endepunkte auf 0 und Gewichte auf 0.0 eingestellt.

4 Visualisierungstechnik

Unter Visualisierung versteht man im Allgemeinen, die abstrakten Daten und ihre Zusammenhänge in einer graphischen bzw. visuell erfassbaren Form zu zeigen. Tatsächlich ist Visualisierung eine Interpretation der Ausgangsdaten in visuelle Medien, so dass die Prozesse und die schwer formulierbaren Zusammenhänge der abstrakten Daten verständlich werden. In der Arbeit wird Informationsvisualisierung (oder „Datenvisualisierung“) verwendet, die sich mit den computer-unterstützten Methoden die große Menge von Daten zur graphischen Repräsentation beschäftigt. Das Ziel ist hierbei, die Interaktion zwischen Mensch und Computer zu verbessern. Durch die bildliche Darstellungsmethode kann der Betrachter einer Visualisierung baldmöglichst einen Überblick über die in den Daten beinhaltenen Informationen bekommen, die Daten auswerten und aus ihnen neue Erkenntnisse gewinnen.

4.1 Einzelner Graph

Ein einzelner Graph ist ein allgemeiner, statischer Graph. Um einen solchen Graphen zu visualisieren, brauchen wir nur seine wichtigen Elemente klar zu repräsentieren, nämlich seine Knoten, seine Kanten und seine Kantengewichte (siehe Abschnitt 2.1 auf Seite 11). Wir benutzen die Programmiersprache JAVA [Ora13], um eine interaktive Visualisierungstechnik zu implementieren. JAVA ist eine objektorientierte Programmiersprache und hat die folgenden Vorteile: Es ist robust und sicher, architekturneutral und portabel, interpretierbar, parallelisierbar und dynamisch [Ged13]. Zwei Klassen „Graph“ und „Item“ werden zuerst als die Datenstrukturen, die die Informationen über Graphen speichern, selbst definiert.

„Item“ hat drei Eigenschaften: Startknoten, Endknoten und Gewicht. Die originalen Texte wie „9 14 1.0“ werden vom Programm zuerst herausgezogen und analysiert. Dann werden die drei Zahlen 9, 14 und 1.0 jeweils in den drei Eigenschaften von „Item“ gespeichert. „Graph“ besteht aus mehreren „Item“. Ein einzelner Graph wird in der Datenstruktur „Graph“ gespeichert. Graph ist ein abstrakter mathematischer Begriff. Wenn er auf Papier gezeichnet wird, ist die Verteilung der Knoten theoretisch beliebig. Es gibt keine Regel, wie ein Graph gezeichnet und gelayoutet werden muss, wo seine Knoten sich befinden müssen. Für verschiedene Graphen sind auch deren Knotenzahlen unterschiedlich. Das Problem ist, dass wir uns eine Form für die Repräsentation ausdenken müssen. In dieser Form können viele verschiedenen Knoten von unterschiedlichen Graphen deutlich gezeigt werden, um dem Betrachter eine Datenanalyse zu erleichtern.

Meine Lösung ist, die Knoten auf die Kanten einer vorgegebenen Form gleichmäßig zu verteilen, z.B. auf ein Rechteck oder einen Kreis. Mit den Methoden „.drawRect()“ und „.drawPolygon()“

des JAVA AWT [Ora13] ist dies leicht zu realisieren. Die Position eines jeden Punktes auf der entsprechenden Form ist auch leicht berechenbar. Falls es eine Kante zwischen zwei Knoten gibt, brauchen wir nur eine Linie zwischen den zwei Konten mit der Methode „.drawLine()“ zu zeichnen. Der RGB-Farbwert in JAVA ist durch solche Zahlen bestimmt. Deshalb können die unterschiedlichen Kantengewichte auch mit RGB-Farben repräsentiert werden. Wir brauchen uns nur eine Funktion auszudenken, die die Abbildung von Zahlen im Datentyp „double“ in einen entsprechenden RGB transformiert, d.h. ein entsprechendes Farbmapping.

4.2 Dynamischer Graph

Ein dynamischer Graph ist eine Folge von Graphen (siehe Abschnitt 2.2 auf Seite 13). Diese Abfolge von Graphen repräsentiert die Veränderung des originalen Graphen im Laufe der Zeit. Tatsächlich besteht ein dynamischer Graph aus mehreren einzelnen Graphen. Um einen guten Überblick zu bekommen, sollen diese Graphen in einer graphischen Benutzeroberfläche des Programms gleichzeitig gezeigt werden. Falls zu viele Graphen angezeigt werden, kann der Benutzer auch auswählen, wie viele Graphen auf einmal in der Benutzeroberfläche angezeigt werden sollen. Falls alle Graphen nicht einmal angezeigt werden können, soll das Programm mit einer „Durchblätter-Funktion“ ausgestattet sein. Um solch eine Funktion zu realisieren, können wir „CardLayout“ von JAVA AWT anwenden. Das CardLayout enthält mehrere „JPanel“s. „JPanel“ von JAVA Swing [Ora13] funktioniert als „Leinwand“, auf der die Abfolge der einzelnen Graphen gezeichnet wird. Das Programm akzeptiert eine Eingabezahl des Benutzers, die die Anzahl der Graphen in einem JPanel bestimmt. Darüber hinaus gibt es noch die Interaktionen zwischen dem Benutzer und dem Programm, welche die dynamischen Graphen direkt manipulieren. Zum Beispiel kann das Programm mehrere ausgewählte Graphen zu einem Graph zusammenaggregieren und diesen Aggregations-Graph auch später wieder „On-Demand“ zerlegen. Mit Hilfe der Interaktionen kann ein Benutzer die Veränderungen der dynamischen Graphen besser und deutlicher beobachten und analysieren.

4.3 Visualisierung der Graphen

Heutzutage ist das *Graphical User Interface* (GUI) eine populäre Methode, mit Software zu interagieren. Aus der Perspektive eines Benutzers lassen GUIs die Software einfacher auf ein Problem anwendbar machen. Das ist auch der Grund, warum die Anwendungen der GUI sich schnell steigern. Weil GUIs immer wichtiger werden, verbringen die Entwicklern auch immer mehr Arbeit an gutem Code, um eine GUI zu implementieren, etwa ausgestattet mit interaktiven Merkmalen. Man meint, dass ungefähr 60% des gesamten Source Codes eines Programms für die Komponenten der graphischen Benutzeroberfläche zuständig sind [Mem02]. GUI Programme, unähnlich wie die traditionelle Software, sind auf „Event“-basierende Systeme gestützt. Eine GUI akzeptiert eine Reihe an Ereignissen, die vom Benutzer oder vom System erzeugt werden [Mem02]. Durch das GUI Programm kann jeder Graph in mehreren Formen repräsentiert werden: die Form „Rechteck“ und „Kreis“ beispielsweise wie sie in dieser Diplomarbeit gezeigt werden.

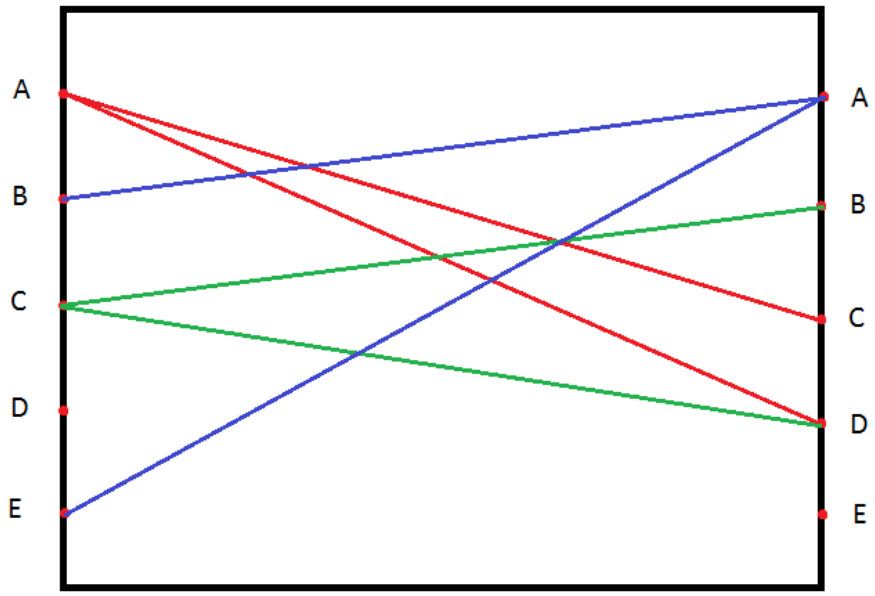


Abbildung 4.1: Die Form Rechteck von Graph.

4.3.1 Visualisierung als rechteckiges Layout

Die Graphen in „.bgraph“ Eingabedateien werden zuerst in Form „Rechteck“ gezeigt. Die Abbildungen 2.1 ist ein gerichteter, kantengewichter Graph. Jetzt möchten wir den Graph in Form Rechteck repräsentieren. Für einen gerichteten Graph hat die Kanten Richtungen. Es gibt Startknoten und Endknoten. (u,v) und (v,u) sind zwei unterschiedliche Kanten. Für einen kantengewichteten Graph haben die Kanten verschiedene Gewichte. In der Form Rechteck müssen diese zwei Aspekte gezeigt. Die Abbildung 4.1 ist die Visualisierung der Form Rechteck. Zuerst gibt es ein Rechteck. Alle Startknoten befinden sich auf der linken Kante des Rechtecks. Alle Endknoten sind auf der rechten Kante. Eine Kante von Graph ist eine Linie von der linken Kante zu der rechten Kante. Die verschiedenen Gewichte werden durch die unterschiedlichen Farben repräsentiert. Je kleiner das Gewicht ist, desto blauer ist seine Farbe. Je Größer das Gewicht ist, umso roter ist seine Farbe. Z.B. in der Abbildung 2.1 gibt es eine Kante von A zu C und das Gewicht ist 3.0. Dann in der Abbildung 4.1 gibt es eine rote Linie vom Knoten A auf der linken Kante des Rechtecks zum Knoten C auf der rechten Kante.

Um die oben erwähnte Umwandlung zu realisieren, wird die Java GUI angewandt, z.B. Swing und AWT. Nach der Behandlung von Eingabedatei können wir wissen, wie viele Knoten ein Graph hat und das maximale Gewicht des Graphen. Mit der Klasse „Graphics“ kann das

Rechteck gemalt. Weil JAVA GUI ein Bild nach „Pixeln“ malt, wissen wir die Koordinationen aller Pixeln vom Rechteck. Das bedeutet, dass wir die Koordinationen aller Startknoten und Endknoten wissen können. Dann können wir alle Knoten des Graphen auf der linken und rechten Kante des Rechteck verteilen. Für jede Kante können wir auch eine Linie von Startknoten zu Endknoten malen. Die einfachste Methode ist, dass jedes Pixel des Rechtecks einen Knoten repräsentiert. Z.B. hat ein Graph 500 Knoten. Dann ist die Höhe des Rechteck 500 Pixeln. Jedes Pixel bedeutet ein Knoten. Falls es eine Kante vom 200ten Knoten zum 300ten Knoten, wird eine Linie vom 200ten Pixel auf dem linken Kante zum 300ten Pixel auf dem rechten Kante des Rechteck gemalt.

Die Kanten mit verschiedenen Gewichten haben unterschiedliche Farben. Weil für verschiedene Eingabedateien ihre Intervalle von Kantengewichten unterschiedlich sind, kann ich keine Kantenfarbe mit die konkreten Gewichte einfach festlegen. Deshalb bestimme ich die Farben durch das Verhältnis von Kantengewicht und dem maximalen Gewicht(Kantengewicht/max-Gewicht). Wenn das Verhältnis zu 0% neigt, tendiert die Farbe zu Blau. Falls das Verhältnis zu 100% neigt, tendiert die Farbe zu Rot. Für das Programm werden 19 Farbenpolitiken entworfen. Ein zweidimensionales Array mit RGB-Werten wird angewandt. Die erste Dimension bestimmt, welche Farbenintervall ausgewählt wird. Die zweite Dimension entscheidet mit dem Verhältnis, welche Farbe in diesem Intervall zurückgegeben wird(Siehe die Klasse Colors in Abschnitt 5.3.2 auf Seite 34).

4.3.2 Visualisierung als kreisförmiges Layout

Die Graphen in den „.bgraph“ Eingabedateien werden auch in verschiedenen Formen wie etwa dem „Kreislayout“ dargestellt. Ein Beispiel ist in der Abbildung 2.1 zu sehen. Jetzt möchten wir den Graph auch in der Form eines „Kreislayout“ realisieren. In dieser Form müssen alle Knoten eines Graphen auf einen Kreisring verteilt werden. Falls es eine Kante zwischen zwei Knoten gibt, wird eine Linie als ihre Verbindung eingezeichnet. Die Abbildung 4.2 ist die Visualisierung eines solchen Kreislayouts. In dieser Form können die Richtungen der Kanten zwar nicht gezeigt werden, aber die Häufigkeit und die Dichte der Kanten ist dennoch erkennbar. In der Abbildung 2.1 gibt es beispielsweise eine Kante von A nach C. Darüber hinaus sieht man in der Abbildung 4.2 auch eine Linie zwischen A und C. Aber die Richtung können wir hier in dieser Abbildung nicht erkennen.

Die Idee beim Kreislayout ist zwar sehr ähnlich wie beim Rechtecks-Layout, aber dennoch gibt es Unterschiede. Die Zeichenkomplexität ist höher beim Kreislayout. Alle Knoten müssen equidistant auf einen Kreisring projiziert werden, während bei einem Rechteck diese nur auf zwei vertikale parallele Linien gezeichnet werden.

Diese Funktion wird in der Methode „Mittelpunktposition + Offset“ realisiert. Das Offset wird mit den Funktionen `sin` und `cos` ausgerechnet, zum Beispiel die Position vom Mittelpunkt eines Kreises (u,v). Der Graph habe 500 Knoten. (X_i, Y_i) ist die Koordinate des i-ten Knoten. Dann kann seine Koordinate durch X_i = u + (Radius + cos((2*π*i)/Knotensumme)) und Y_i = v + (Radius + sin((2*π*i)/Knotensumme)) berechnet werden. Die Koordinaten aller Knoten werden in zwei Arrays abgespeichert. Eines speichert alle X-Koordinaten, das andere speichert

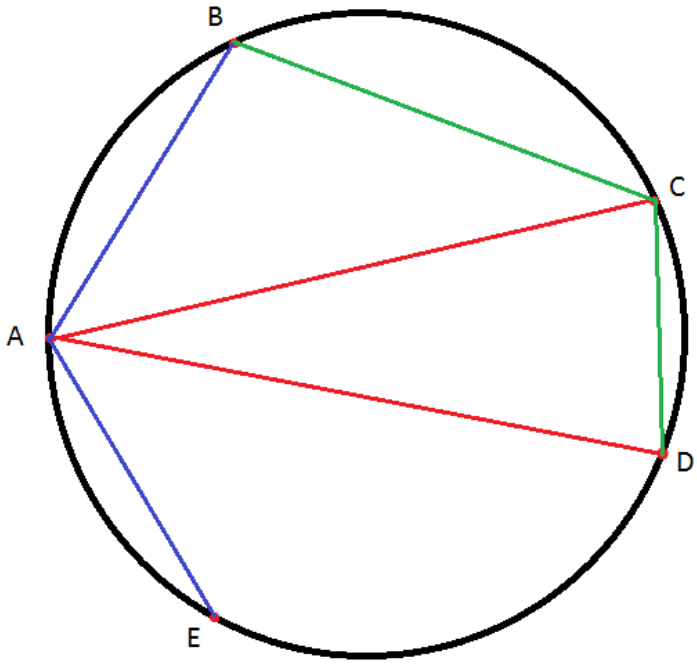


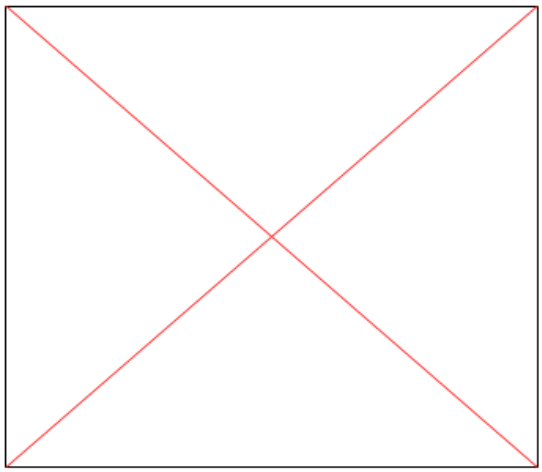
Abbildung 4.2: Ein Graph im Kreislayout.

alle Y-Koordinaten. Durch diese 2 Arrays und die Methode „Graphics.drawPolygon()“ kann das Kreislayout für alle Knoten gezeichnet werden. Die anderen Aufgaben, z.B. Kanten zu zeichnen und die Kantenfarben auszuwählen, sind ähnlich zu realisieren wie beim Rechteckslayout, das in Abschnitt 4.3.1 auf Seite 23 vorgestellt wird.

4.3.3 Matrix

Nachdem die Graphen in den Formen „Rechteck“ in Abschnitt 4.3.1 auf Seite 23 und „Kreis“ in Abschnitt 4.3.2 auf der vorherigen Seite visualisiert worden sind, können diese auch in Form einer „Matrix“ gezeigt werden. Zuerst wird jeder Graph in eine Matrix umgewandelt. Dann werden neue Graphen entsprechend der Sortierung in den bereits existierenden Matrizen generiert.

Das Rechteck-Layout und das Kreis-Layout wird mit Swing und AWT dargestellt. In Swing und AWT werden die Größe und Position eines Bildes durch Pixel bestimmt. Für eine „Matrix“-Visualisierung ist die Situation ähnlich. Wir teilen ein Rechteck- oder Kreis-Layout in m*n „kleine Stücke“. Diese m*n Stücke bilden eine m*n Matrix. Die Farben der Pixel im originalen Graph werden durch die Gewichte bestimmt. Deshalb können die Werte jeder Matrixeinheit



a)

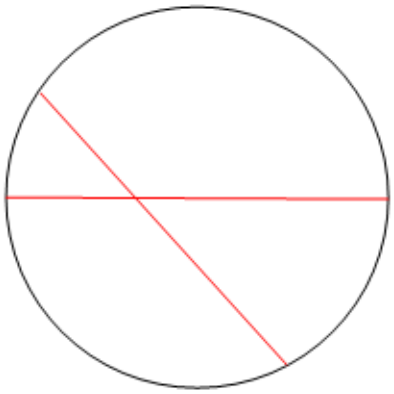
1.0	0.0	0.0	0.0	1.0
0.0	1.0	0.0	1.0	0.0
0.0	0.0	2.0	0.0	0.0
0.0	1.0	0.0	1.0	0.0
1.0	0.0	0.0	0.0	1.0

b)

Abbildung 4.3: a) Ein Rechtecks-Layout, das zwei Kanten besitzt. Die Gewichte aller Kanten sind 1.0. b) Die aus dem Graph enstandene Matrix. Alle Einheiten, deren Werte 1.0 sind, bilden auch zwei „Linien“ in der Matrix. Aber der Wert vom Schnittpunkt ist in der Summe dann 2.0.

auch durch die Gewichte ausgerechnet werden. Zum Beispiel ist die Abbildung 4.3 eine Umwandlung eines Rechteck-Layouts in eine Matrix. Der Rechteck-Graph in Abbildung 4.3a) hat zwei Kanten, deren Gewichte beide 1.0 sind. Diese zwei Kanten überschneiden sich. Falls eine Linie(Kante) eine Einheit der Matrix überquert, ist der Wert der Einheit 1.0. Die Einheiten, deren Gewichte 1.0 sind, bilden zwei Linien in der Matrix. Der Wert ihres Schnittpunktes ist die Summe der zwei Kantengewichte, nämlich 1.0 + 1.0 = 2.0. Die Abbildung 4.4 zeigt ein Beispiel für die Umwandlung eines Kreis-Layouts in eine Matrix. Es ist ein ähnliches Szenario wie in Abbildung 4.3. Das Kreislayout hat zwei Kanten, die sich auch überschneiden, wie in der Abbildung 4.4a) zu sehen ist. Die Knoten werden in der Matrix der Abbildung 4.4b) auf einem „Kreis“ verteilt. Der Wert des Schnittpunktes der zwei Kanten ist auch die Summe der Gewichte.

Offensichtlich, je feiner ein Graph unterteilt wird, umso genauer ist auch die Matrix. Zum Beispiel möchten wir eine m*m Matrix `arr` bekommen. Wir wissen die Dimension des originalen Graphen, zum Beispiel n*n Pixel. Dann wird für jedes Pixel (x,y) im originalen Graphen seine Position durch das Verhältnis der Dimensionen von Matrix und Graph, nämlich m/n, berechnet. Die Einheit arr[y*(m/n)][x*(m/n)] ist das gewünschte Ergebnis. Bitte beachten Sie, dass die Suffixe von Matrix und Pixel umgekehrt sind. Die y-Dimension des Pixels wird angewandt, die x-Dimension der Matrix zu berechnen. Die x-Dimension der Pixel ist für die Berechnung der y-Dimension der Matrix zuständig. Wenn wir die Matrix erhalten, ist es relativ einfach, ein Bild gemäß der Matrix zu zeichnen. Die Einheiten der Matrix werden als neue „Pixel“ behandelt. Durch die Methode „Graphics.fillRect()“ können m*m kleine Rechtecke



a)

0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0
1.0	2.0	1.0	1.0	1.0
0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	1.0	0.0

b)

Abbildung 4.4: a) Ein kreisförmiges Layout, das auch zwei Kanten zeigt. Ihre Gewichte sind auch 1.0. b) Die aus a) entstandene Matrix. Alle Einheiten, deren Werte 1.0 sind, bilden zwei „Linien“ in der Matrix. Der Wert vom Schnittpunkt ist auch hier 2.0.

gezeichnet werden. Der Wert der Einheit bestimmt dann die Farbe. Mit dieser Strategie wird dann ein „Matrix-Graph“ produziert.

4.3.4 Splatting

Jetzt wird der Begriff der „Interpolation“ eingeführt. In der numerischen Mathematik versteht man unter dem Begriff Interpolation eine Mehtode, neue Datenpunkte im Bereich einer diskreten Reihe von bekannten Datenpunkten zu konstruieren. Jede Einheit der in Abschnitt 4.3.3 auf Seite 25 erhaltenen Matrix soll interpoliert werden. Zum Beispiel wird in der Abbildung 4.5a) die Einheit $a_{(i,j)}$ interpoliert. Zuerst werden der Wert von $a_{(i,j)}$ und die Werte aller 8 umgebenden Einheiten akkumuliert. Der neue Wert von $a_{(i,j)}$ ist dann der Durchschnittswert all dieser. Die zugehörige Formel ist

$$a_{(i,j)} = (a_{(i-1,j-1)}+a_{(i-1,j)}+a_{(i-1,j+1)}+a_{(i,j-1)}+a_{(i,j)}+a_{(i,j+1)}+a_{(i+1,j-1)}+a_{(i+1,j)}+a_{(i+1,j+1)})/9$$

Es ist ein Box Filter, der grundsätzlich ein „average-of-surrounding-pixel“ Typ Image-Filter darstellt [TA07]. In der Abbildung 4.5a) ist die allgemeine Situation dargestellt. Die Einheit befindet sich innerhalb der Matrix. Falls die bearbeitete Einheit sich auf Kanten befindet, umgeben sie nur 5 Einheiten. Der Durchschnittswert soll ihre Summe geteilt durch 6 sein. Die Abbildung 4.5b) zeigt, dass die interpolierte Einheit sich auf der linken Kante befindet. Wenn die behandelte Einheit sich auf der oberen, unteren oder rechten Kante befindet, sind die

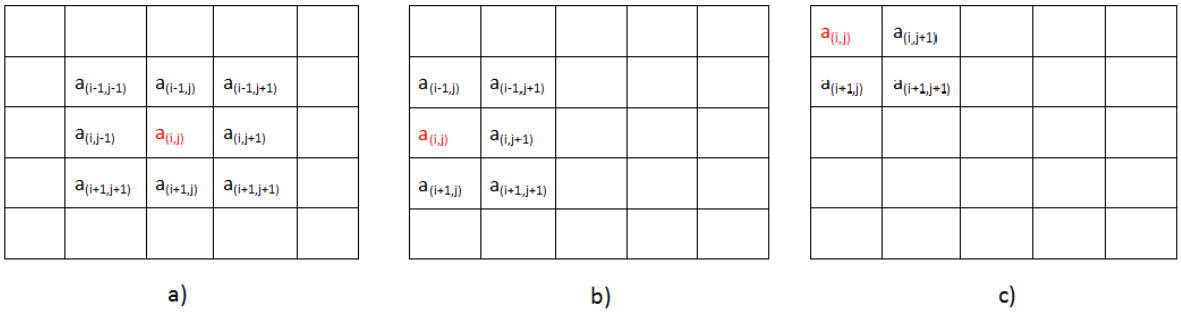


Abbildung 4.5: a) Interpolation der allgemeinen Einheit. Box Filter in einem 3*3 Gebiet. b) Die Einheit befindet sich auf der Kante. Box Filter in einem 2*3 Gebiet. c) Die Einheit befindet sich in der Ecke. Box Filter in einem 2*2 Gebiet.

Situationen ähnlich wie in Abbildung 4.5b). Die Abbildung 4.5c) zeigt, dass die interpolierte Einheit sich in der Ecke befindet. Bei der Situation wird sie von 3 anderen Einheiten umgeben. Ihre Summe soll durch 4 geteilt werden. Falls die Einheit in den anderen 3 Ecken liegt, sind die Situationen auch ähnlich wie in der Abbildung 4.5c) dargestellt. Der Unterschied liegt nur in den verschiedenen Suffixen der Einheiten.

Nach der Interpolation-Operation bekommen wir eine neue Matrix für jede in Abschnitt 4.3.3 auf Seite 25 beschriebene Matrix. Durch die neue Matrix kann ein „Splatting-Graph“ gezeichnet werden. Die Methode entspricht der in Abschnitt 4.3.3 auf Seite 25 beschriebenen.

4.4 Interaktion

In einem Visualisierungsprogramm ist die Interaktion zwischen Benutzer und Computer wichtig. Nachdem die Graphen in den obigen Formen gezeigt geworden sind, muss die Interaktion auch realisiert werden, ansonsten wäre der Betrachter auf statische Abbildungen eingeschränkt. Generell gibt es zwei Typen von Interaktionen in der Arbeit: Aggregation und Zerlegung von Graphen.

4.4.1 Aggregation

Mehrere Graphen können zu einen neuen Graph „aggregiert“, d.h. zusammengefasst werden. Es ist egal, ob diese aggregierten Graphen kontinuierlich oder diskontinuierlich sind. Nach der Aggregation werden alle aggregierten Graphen entfernt. Aber der neu hergestellte Aggregations-Graph ersetzt die Position des ursprünglichen Graphen in diesen aggregierten Graphen. Zum Beispiel gibt es eine Reihe von Graphen (G_1, G_2, ..., G_n). Jetzt werden G_2, G_4 und G_5 zusammengefasst und ein neuer Aggregations-Graph G_{neu} wird daraus gebildet. Dann werden G_2, G_4 und G_5 entfernt. G_{neu} wird auf die Position des früheren G_2 eingefügt. Die Inhalte und

Positionen aller anderen Graphen verändern sich nicht. Jetzt ist die Reihenfolge der Graphen wie folgt: (G_1, G_{neu}, G_3, G_6, G_7, ..., G_n).

Eine Eingabedatei (siehe die Abbildung 3.1) beinhaltet mehrere Graphen. Jeder Graph besteht aus Items, wie z.B. „1 5 1.0“. Der Startknoten „1“ und der Endknoten „5“ sind tatsächlich die echten Positionen der Startknoten und Endkonten in der Eingabedatei. Die Zahl „1“ bedeutet, dass der Startknoten die erste Zeile in der Eingabedatei ist. Die Zahl „5“ repräsentiert, dass der Endknoten die fünfte Zeile ist. Für alle in einer Eingabedatei enthaltenen Graphen sind die Indizes ihrer Startknoten und Endkonten gleich. In einem Graph bedeutet „1“ die erste Zeile, während in einem anderen Graph die Zahl „1“ auch die erste Zeile in dieser gleichen Eingabedatei repräsentiert. Alle aggregierten Graphen befinden sich in einer Eingabedatei. Deshalb können die Items dieser Graphen einfach kombiniert werden, wenn die Graphen zusammengefasst werden. Alle Zahlen im neu erhaltenen Graph repräsentieren die entsprechenden richtigen Zeilen. Zum Beispiel enthält G_1 die Zeile „1 5 1.0“. G_2 umfasst „2 7 1.0“. Jetzt werden G_1 und G_2 zusammengefügt. Wir können die Zeilen „1 5 1.0“ und „2 7 1.0“ in den neuen Graph einfach einbetten. Die Zahlen „1“, „5“, „2“ und „7“ im neuen Graph orientieren sich nach den richtigen Positionen, nämlich die erste, fünfte, zweite und siebte Zeile. Tatsächlich sind die Prozesse der Aggregation so: Zuerst werden die Items (Siehe Abschnitt 3.3 auf Seite 18) aller Graphen herausgezogen. Dann wird mit allen herausgezogenen Items ein neuer Graph generiert, der gleich dem neu erhaltenen Aggregations-Graph ist.

Jetzt berücksichtigen wir die folgende Situation. G_1 beinhaltet ein Item „1 5 1.0“, während G_2 ein Item „1 5 2.0“ enthält. Offensichtlich haben diese zwei Items die gleichen Startknoten und Endkonten. Aber ihre Gewichte sind ungleich. Wenn der neue Aggregations-Graph gezeichnet wird, wird die zuletzt gezeichnete Linie die erst gezeichnete Linie überdecken. Um das Problem zu lösen, wird jeder Graph vor dem Zeichnen zuerst separat behandelt. Jedes Item wird mit allen anderen Items verglichen. Falls mehrere Items die gleichen Startknoten und Endknoten aber ungleiche Gewichte haben, wird nur ein Item mit den Startknoten und Endknoten bleiben. Sein Gewicht ist das maximale Gewicht zwischen allen Gewichten dieser Items. Im oben erwähnten Beispiel wird das Item „1 5 2.0“ im neuen Graph bleiben.

4.4.2 Zerlegung

Ein Aggregations-Graph besteht aus mehreren Graphen, entweder originalen Graphen oder anderen Aggregations-Graphen. Nur die Aggregations-Graphen (siehe Abschnitt 4.4.1 auf der vorherigen Seite) können in eine Reihe von Graphen zerlegt werden, die vorher den zerlegten Aggregations-Graphen erzeugt haben. Darüber hinaus muss die Sequenz der neu erhaltenen Graphen gleich der Sequenz in der originalen Reihenfolge sein. Wie das Beispiel in Abschnitt 4.4.1 auf der vorherigen Seite zeigt, wird der Graph G_{neu} jetzt zerlegt. Das Ergebnis muss G_2, G_4 und G_5 sein. Ihre Sequenz muss deshalb (G_1, G_2, G_3, G_4, G_5 G_6, G_7 ..., G_n) sein. Aus diesem Grund müssen alle Graphen zuerst nummeriert werden. Für jeden Aggregations-Graph müssen die Indizes auch notiert werden, um herauszufinden, welche Graphen ihn ausmachten. Dann können bei der Zerlegung die richtigen Graphen wiederhergestellt werden. Zum Beispiel wenn der Graph G_{neu} zerlegt wird, werden G_2, G_4 und G_5 erhalten. G_{neu} wird aus der Graphenfolge entfernt. Jetzt ist die Folge der Graphen wie folgt: (G_1, G_3, G_6, G_7,

..., G_n). Dann wird G_2 extrahiert. Der Graphindex ist 2, also größer als der Index von G_1 und kleiner als der Index von G_3. Dann wird G_2 zwischen G_1 und G_3 eingefügt. Falls der Index größer als alle Indizes in der Reihenfolge ist, wird der Graph am Ende der Reihenfolge eingefügt. Jetzt ist die Reihenfolge: (G_1, G_2, G_3, G_6, G_7, ..., G_n). Das Programm wiederholt diese Schritte für G_4 und G_5. Zum Schluss wird das Ergebnis (G_1, G_2, G_3, G_4, G_5, G_6, G_7, ..., G_n) erhalten. Durch den Vergleich der Indizes können die wiederhergestellten Graphen an den richtigen Positionen wieder eingefügt werden.

5 Implementierung

„*Design before coding*“ [Roy70]. Bevor wir eine Software entwickeln, sollen wir zuerst die sachlichen und technischen Anforderungen über diese Arbeit systematisch untersuchen, z.B. welche Funktionen verlangt werden und was diese Funktionen genau tun. 40%-60% der Probleme in Softwareprojekten werden in der Phase der Anforderungsanalyse verursacht [AL09]. Um die Anforderungen möglichst ausführlich zu erfassen, um den Entwurf und die Systemarchitektur des Programms möglichst genau festzulegen, gibt es die Phase des „Rapid Prototype Model“.

5.1 Rapid Prototype Model

Wenn ein Maschinenbauingenieur eine Designaufgabe bekommt, wird er normalerweise ein Baumuster nach den Anforderungen und seinen eigenen Verständnissen in kurzer Zeit zuerst machen und gibt das Baumuster zu Kunden. Falls das Baumuster kein Problem darstellt, wird die Serienproduktion erst nach der Bestätigung des Kunden zugelassen. Dieses Baumuster stellt einen Prototyp dar. Bei der Methode „Rapid Prototype Model“ im Bereich Informatik wird zuerst ein Prototyp hergestellt, der die Hauptanforderungen des Kunden widerspiegeln kann. Der Prototyp ist nicht detailliert, aber er kann schnell implementiert werden. Dadurch können die Kunden einen Überblick über das zukünftige System bekommen, so dass die Kunden urteilen können, welche Funktionen die Kundenanforderungen zufriedenstellen und welche Aspekte noch verbesserungswürdig sind. Nachdem der Prototyp festgelegt wurde, wird der Entwickler das auf den Prototyp basierende System weiterentwickeln. Weil der Prototyp abhängig von der Entwicklungsgeschwindigkeit ist und nicht sehr detailliert ist, durchläuft er in der folgenden Entwicklung vielleicht viele Veränderungen. Letztendlich wird der alte Prototyp sogar weggeworfen und das eigentliche System komplett neu implementiert. Trotzdem ist das „Rapid Prototype Model“ eine gute Methode, um den Überblick über das System schnell zu bekommen und um die Kundenanforderungen genau zu ermitteln. Die Abbildung 5.1 zeigt den Prototypen, wie er als Programm in der zugrundeliegenden Diplomarbeit entworfen wurde.

Das Interface des Prototyps besteht aus drei Bereichen: der linke Bereich, der zentrale Bereich und der rechte Bereich, wie in Abbildung 5.1 illustriert. Der linke Bereich umfasst die Funktionsknöpfe. Diese Knöpfe realisieren einige Funktionen, zum Beispiel das Öffnen einer Eingabedatei, den Überblick mehrerer Graphen zeigen und wie viele Graphen in einer Seite gezeigt werden, um nur einige zu nennen. Falls es mehrere Seiten gibt, kann der Benutzer auch mit den Knöpfen „PREVIOUS“ und „NEXT“ durch die Graphen durchblättern. Der rechte Bereich umfasst einige Informationen, die auf die geöffnete Eingabedatei und den im zentralen Gebiet ausgewählten Graphen bezogen sind. Alle Graphen werden im zentralen Bereich gezeigt. Der Benutzer kann auswählen, wie viele Graphen in einer Seite gezeigt werden sollen. Wenn ein

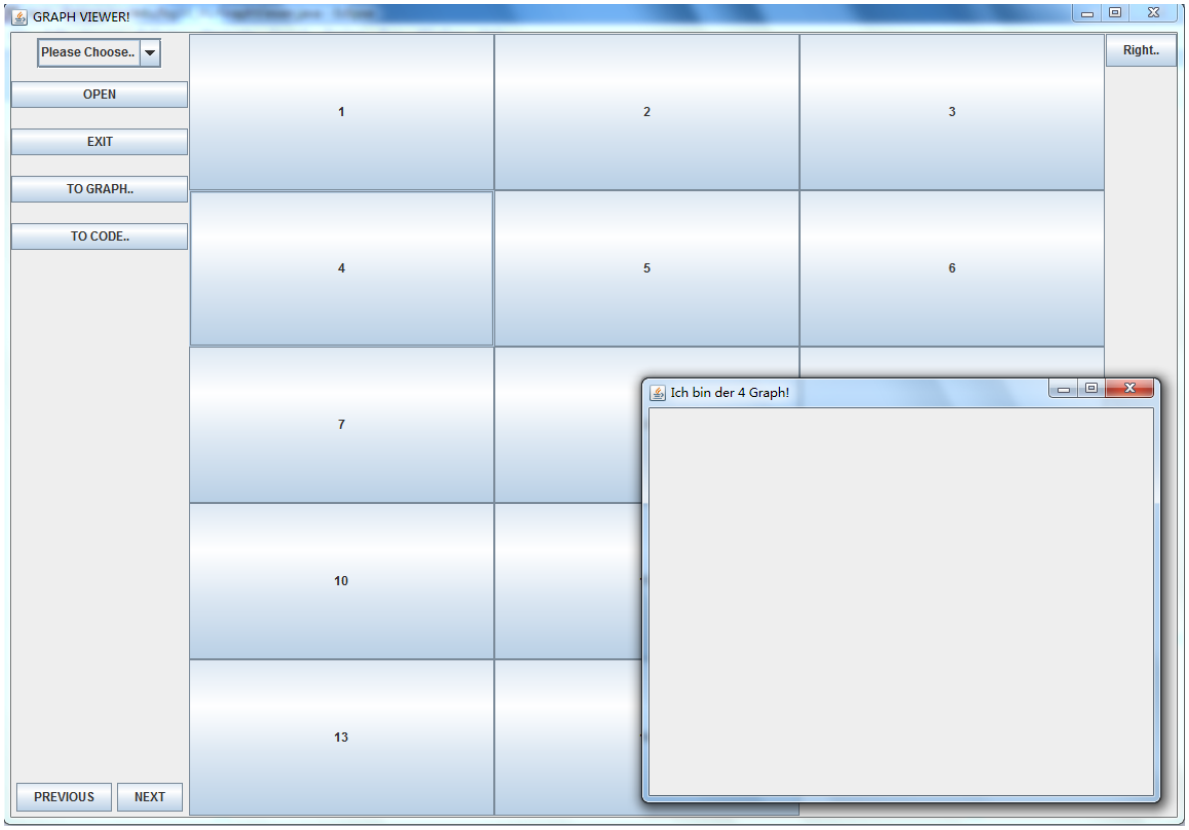


Abbildung 5.1: Prototyp

Graph angeklickt (selektiert) wird, springt ein neues Fenster auf, das den konkreten selektierten Graphen in vergrößerter Form anzeigt. Das grobe Layout des Programms wird in Abbildung 5.1 illustriert.

5.2 Incremental Model

Das incremental Model ist ein iteratives Modell in der Software Entwicklung. Software wird als eine Reihe von relativen Schritten (Increments) betrachtet. Bei jedem Schritt von Iterationen wird eine Erweiterung hinzugefügt. Deshalb ist das Incremental Model als Typ auch ein evolutionäres Modell (Evolutionary Model), das sich durch diese Iterationen Schritt für Schritt vervollkommnet. In dieser Arbeit wird ein solches Incremental Model angewandt. Das Programm wird nach den folgenden Schritten entwickelt:

I Das Programm kann zuerst eine „.bgraph“ Eingabedatei einlesen und ihre Inhalte im zentralen Bereich anzeigen.
II Dann werden alle in der Eingabedatei enthaltenen Graphen im zentralen Bereich gezeigt.

III Die Graphen besetzen vielleicht mehrere Seiten. Der Benutzer kann auswählen, wie viele Graphen in einer Seite gezeigt werden.
IV Die Funktion des Durchblätterns wird realisiert.
V Die Graphen werden mit zwei Formen dargestellt, die Formen „Rechteckslayout“ und „Kreislayout“. Der Benutzer kann die Graphen zwischen den beiden Darstellungsformen variieren.
VI Wenn ein Graph (als Knopf realisiert) im zentralen Bereich angeklickt wird, wird der konkrete Graph in einem vergrößerten Popup-Fenster dargestellt.
VII Die Farben der Kanten im Graph werden durch die Gewichte bestimmt. Verschiedenen Farbschemata können realisiert werden.
VIII Mehrere Graphen können zu einem Aggregations-Graphen kombiniert werden. Ein Aggregations-Graph kann auch wieder zu mehreren Graphen zerlegt werden. Die Kombinations- und Zerlegungssequenzen sind egal, aber die Reihenfolge der originalen Graphen wird stets beibehalten.
IX Jeder Graph in beiden Darstellungsformen Rechtecks- und Kreislayout wird in eine Matrix umgewandelt, d.h. die Pixelinformationen werden somit kodiert. Mit Hilfe der Matrizen werden schließlich die Matrix-Graphen als pixelbasierte und nicht linienbasierte Darstellung gezeichnet.
X Auf jede solche Matrix wird die „Splatting“-Operation angewandt. Durch die neu erhaltene Matrix wird somit ein „Splatted“ Graph erzeugt, der dann als Dichtefeld visualisiert werden kann.
XI Die konkreten Informationen über den ausgewählten Graph werden im rechten Bereich angezeigt. Diese Details-on-Demand Funktion ist enorm wichtig, um dem Betrachter die Graphsequenz leichter interpretieren zu lassen und um visuelle Muster auf die zugrundeliegenden Daten zuzuordnen.

Die Abbildung 5.2 zeigt das Interface des Visualisierungsprogrammes.

5.3 UML

Die Unified Modeling Language (UML) ist eine visuelle Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von Software und Systemen [G$^+$]. UML ist heute die dominierende Sprache, um Software und Systeme zu modellieren. In diesem Abschnitt werden einige UML-Diagramme, die Teile des Progammes illustrieren, beschrieben.

5.3.1 Anwendungsfalldiagramm

Ein Anwendungsfalldiagramm ist eine Diagrammart basierend auf UML. Es zeigt eine Sicht für den Benutzer auf das erwartete Verhalten und die Anforderungen eines Systems. Anwendungsfalldiagramme haben zwei wichtigen Elemente: den Anwendungsfall selbst und die Akteure. Anwendungsfälle halten fest, was ein System tun soll. Akteure spezifizieren, wer (im Sinne einer Person) oder was (im Sinne eines anderen Systems) etwas mit dem System tun soll [RS07]. Die Abbildung 5.3 zeigt das Anwendungsfalldiagramm dieses Programms. Der Benutzer gibt

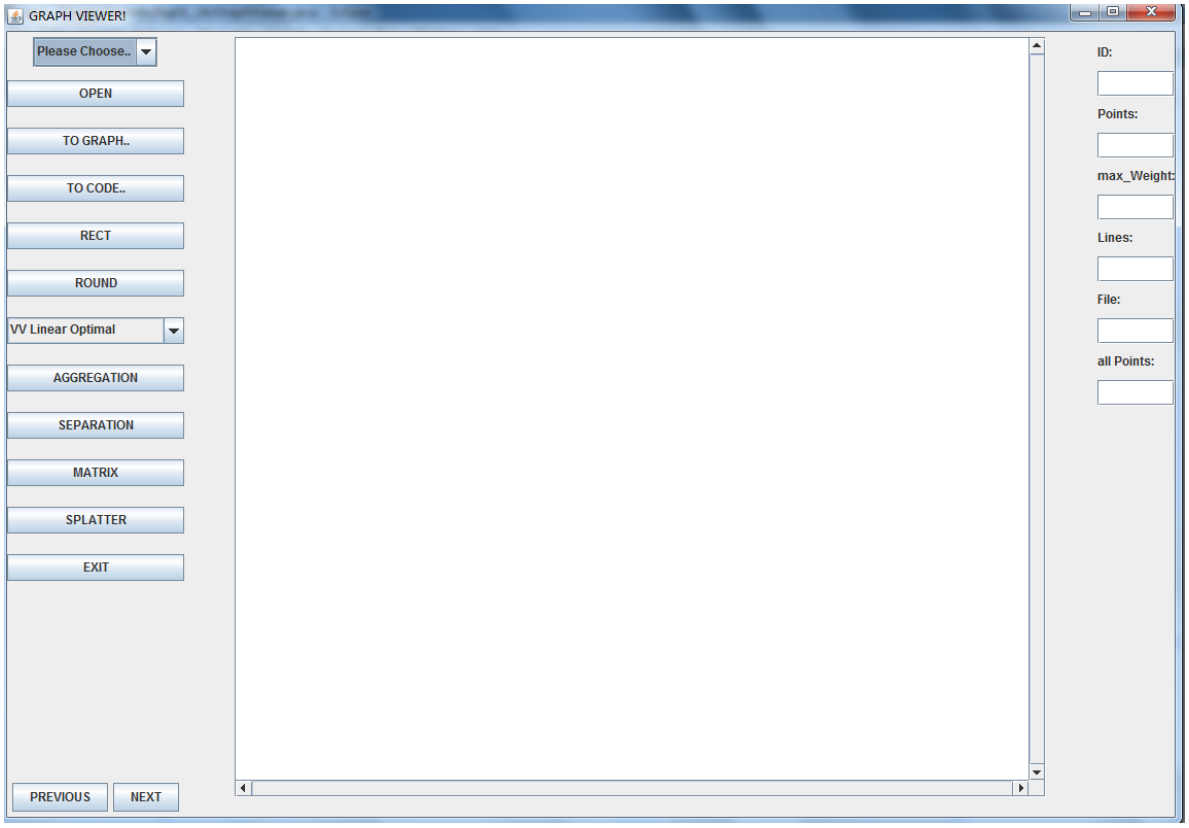


Abbildung 5.2: Interface des Visualisierungsprogrammes.

eine Eingabedatei ein. Das Programm behandelt die Datei und zeigt alle darin umfassten Graphen in beiden Darstellungsformen Rechtecks- und Kreislayout. Der Benutzer kann die Visualisierung zwischen den beiden Darstellungsformen hin- und herwechseln. Dann wird die Matrix für jeden Graph ausgerechnet und der entsprechende Matrix-Graph angezeigt. Danach wird ein Box-Filter angewandt und die „Splatting“-Funktion realisiert. Jede Visualisierung, die in diesem Prozess generiert wird (z.B. Rechteck- und Kreislayout, Matrix-Graphen und Splatting), wird zum Benutzer zurückgegeben. Dadurch können die Interaktionen zwischen Benutzer und System durchgeführt werden. Der Benutzer kann damit einen klareren Überblick über die Änderungen der dynamischen Graphen bekommen und die Abfolge der Graphen besser analysieren.

5.3.2 Klassendiagramm

Statt der traditionellen prozessorientierten Programmierung wird die objektorientierte Programmierung immer breiter angewandt. Unter Objektorientierung versteht man eine Sichtweise auf komplexe Systeme, bei der ein System durch das Zusammenspiel kooperierender Objekte beschrieben wird. In objektorientierter Programmierung werden die Dateien und die Ope-

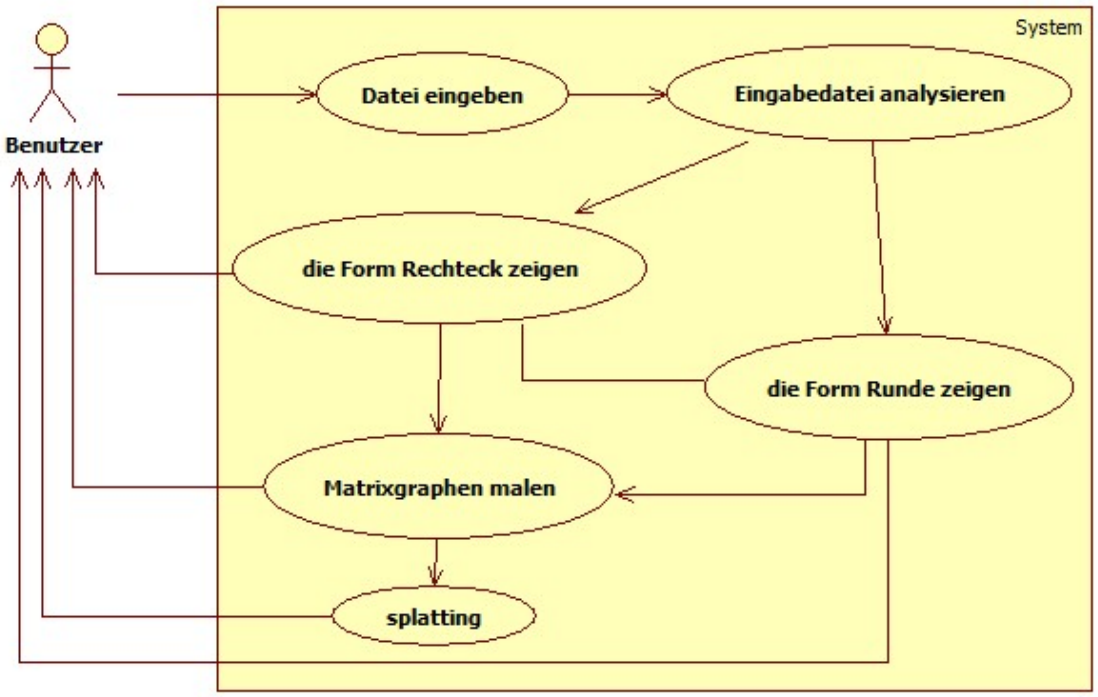


Abbildung 5.3: Das Anwendungsfalldiagramm des Programms.

rationen in dem sogenannten „Objekt“ (Object) gekapselt. Die Kommunikationen zwischen Objekten werden durch „Messages“ realisiert. Der Mechanismus „Objekt + Message“ ersetzt den Gedankengang „Datenstruktur + Algorithmus“.

Ein Klassendiagramm ist ein Strukturdiagramm basierend auf UML zur graphischen Darstellung (Modellierung) von Klassen, Schnittstellen sowie deren Beziehungen. In der Objektorientierung sind die Klassen abstrakte Oberbegriffe, die die gemeinsame Struktur und das Verhalten von Objekten beschreiben. Die Klassen abstrahieren dabei die Objekte. Die Abbildung 5.4 zeigt das Klassendiagramm des Programms.

Die Klasse „GraphViewer“ ist die Main Function. In der Klasse wird das graphische Interface aufgebaut, nämlich linker Bereich, zentraler Bereich, rechter Bereich. Deshalb wird das „Border-Layout“ angewandt. Der „West“ Bereich und „East“ Bereich enthalten die Funktionsknöpfe, während der „Center“ Bereich die GUI beschreibt, nämlich den allgemeinen Visualisierungsbereich. In der Klasse werden die Funktionsknöpfe und ihre entsprechenden „ActionListener“ definiert. Die Klasse GraphViewer ruft andere Klassen auf. Der linke Bereich beinhaltet die Funktionsknöpfe über die „Eingabe“.

Von oben bis unten ist zuerst eine Dropdown-Liste zu sehen, dadurch kann der Benutzer auswählen, wie viele Graphen im zentralen Bereich auf einmal gezeigt werden sollen. Dies wird durch eine „JCombobox“ von JAVA realisiert. Ihr entsprechender ActionListener muss auch in

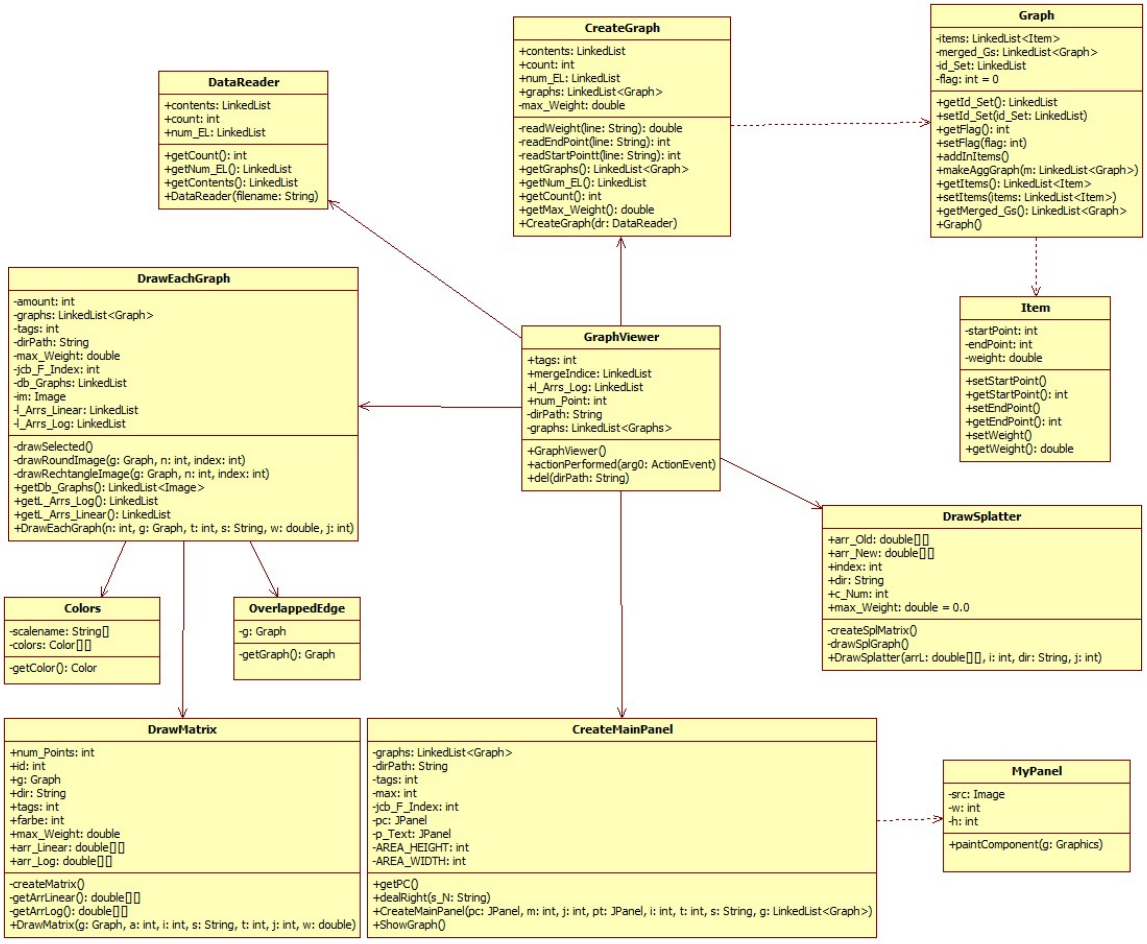


Abbildung 5.4: Das Klassendiagramm des Programms.

der Klasse definiert werden. Wenn die ausgewählte Zahl der JCombobox jedes mal verändert wird, ruft ihr ActionListener die Klasse „CreateMainPanel“ auf, um den zentralen Bereich neu zu zeichnen. Dann erscheint der Knopf „OPEN“, der eine Eingabedatei öffnen und seine Inhalte im zentralen Bereich anzeigen soll. Sein ActionListener ruft zuerst die Klasse „DataReader“ auf. Die Klasse „DataReader“ analysiert die Eingabedatei, bekommt die Informationen über die Eingabedatei und schiebt die Inhalte der Eingabedatei in eine „JTextArea“, die ein Panel im zentralen Bereich beschreibt. Dann wird die Klasse „CreateGraph“ angerufen. Diese Klasse liest die von der Klasse „DataReader“ analysierten Informationen ein, stellt die in der Eingabedatei enthaltenen Graphen durch diese Informationen her und speichert alle Graphen in einer „LinkedList“. Zum Schluss wird die LinkedList als eine „LinkedList graphs“ zurückgegeben. Danach wird die Klasse „DrawEachGraph“ aufgerufen, die das konkrete Bild von jedem Graph in beiden Darstellungsformen „Rechteckslayout“ und „Kreislayout“ zeichnet. Dann wird die

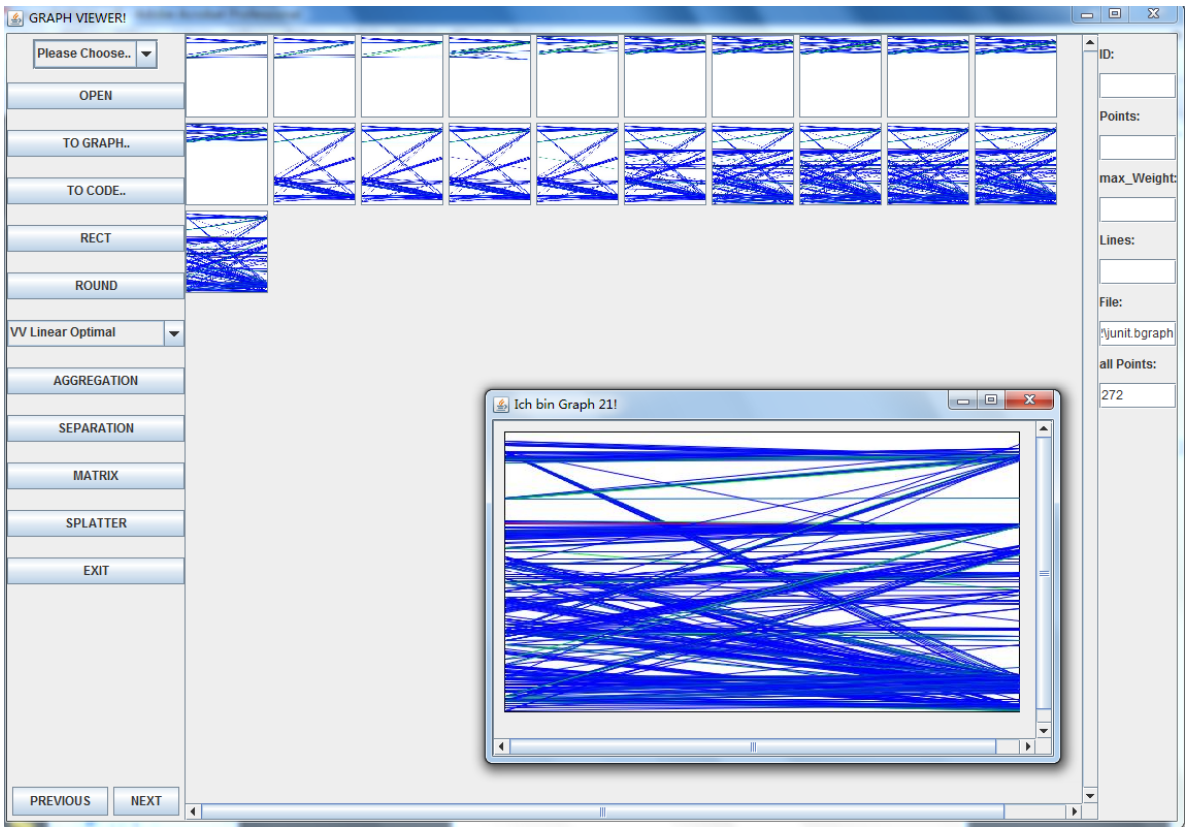


Abbildung 5.5: Die Visualisierung der Darstellungsform „Rechteckslayout“

Klasse „CreateMainPanel“ aufgerufen, die das Layout der Graphen im zentralen Bereich generiert.

Unter „OPEN“ sind zwei Knöpfe „TO GRAPH..“ und „TO CODE..“ platziert. Wenn der Benutzer „TO GRAPH..“ anklickt, werden die Graphen im zentralen Bereich angezeigt. Wenn der Benutzer „TO CODE..“ anklickt, werden die Inhalte der Eingabedatei in Text-Form im zentralen Bereich dargestellt. Dann gibt es die Knöpfe „RECH“ und „ROUND“. Durch diese beiden Knöpfe kann der Benutzer die Layouts der Graphen zwischen den Darstellungsformen „Rechteckslayout“ und „Kreislayout“ variieren. Die Abbildung 5.5 zeigt eine Visualisierung in der Darstellungsform Rechteckslayout. Wenn der Knopf „ROUND“ geklickt wird, wird die Visualisierung wie die in Abbildung 5.6 illustriert erzeugt, die die Graphen im Kreislayout zeigt. Ihre Actionlisteners rufen die Klasse „DrawEachGraph“ auf, um jede konkrete Visualisierung eines einzelnen Graphen zu zeichnen. Eine Varialble „int tags;“ wird hier definiert und an die Klasse DrawEachGraph übertragen. Wenn tags=1, zeichnet die Klasse DrawEachGraph Graphen im „Rechteckslayout“. Wenn tags=2, zeichnet sie Graphen im „Kreislayout“.

Dann gibt es eine weitere Dropdown-Liste, die 19 Farbschemata beinhaltet. In ihrem Action-Listener wird das ausgewählte Farbschema an die Klasse DrawEachGraph übertragen und die Klasse CreateMainPanel wird danach aufgerufen. Der Benutzer kann mehrere Graphen

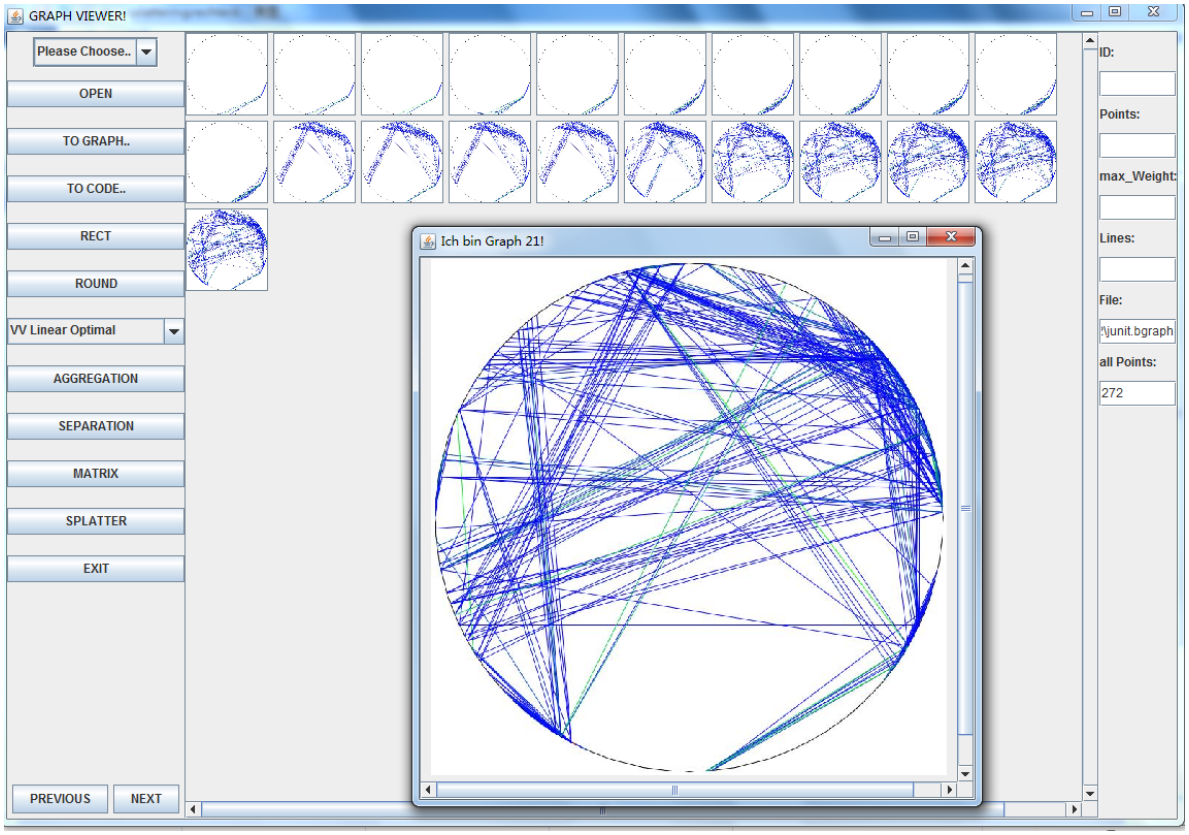


Abbildung 5.6: Die Visualisierung der Darstellungsform „Kreislayout“

auswählen (durch Rechtsklick mit der Maus) und den Knopf „AGGREGATION“ anklicken. Dann werden alle ausgewählten Graphen zu einem Aggregations-Graphen zusammengefasst, wie in der Abbildung 5.7 dargestellt. Der neue Graph 2 ist aus den originalen Graphen 2, 9 und 19 zusammengesetzt. Eine LinkedList „mergeIndice“, die die ausgewählten Graphen speichert, wurde ebenfalls mit ihrem zugehörigen ActionListener implementiert. Die aggregierten Graphen werden aus der alten LinkedList entfernt. Die Reihenfolge der Entfernungsoperation soll beim Graph mit dem kleinsten Index beginnen und zum Graph mit dem größten Index laufen. Falls der Graph mit dem kleinem Index zuerst entfernt wird, werden die Positionen aller hinteren Graphen sich natürlich verändern.

Zum Beispiel werden die Graphen 2 und 7 kombiniert. Falls der Graph 2 zuerst entfernt wird, wird der Graph 7 natürlich zum sechsten Graph. Deshalb werden die Graphen in der LinkedList „mergeIndice“ nach den Indizes von groß zu klein neu angeordnet. Dann kann das Programm die Graphen durch „mergeIndice.get(i)“ in der gewünschten Sequenz wieder extrahieren. Der neu erhaltene Aggregations-Graph wird in der Position des Graphen eingefügt, deren Index am kleinsten zwischen allen entfernten Graphen ist. Der neue Aggregations-Graph trägt auch eine Information darüber, aus welchen Graphen er zusammengesetzt wurde. Er muss die Indizes dieser Graphen speichern, weil wir Aggregations-Graphen auch irgendwann wieder interaktiv

Algorithmus 5.1 Separation Algorithmus

```
procedure SEPARATION(Graph g_Div)
    LinkedList < Graph > graphs ← alle Graphen;
    LinkedList < Graph > gs_Merged ← die SubGraphen von g_Div;
                                                    // g_Div: der zerlegte Graph.
    for all Graph(i) ∈ gs_Merged do
        for all Graph(j) ∈ graphs do
            if (Graph(i)'s Index) < (Graph(j)'s Index) then
                den Graph(i) in der Graph(j)'s Position einfügen;
            end if
            if alle Graph(j)s werden vergleicht then
                den Graph(i) am Ende von graphs einfügen;
            end if
        end for
    end for
end procedure
```

zerlegen können müssen. Für einen originalen Graphen ist dies trivial. Wir brauchen nur seinen Index zu speichern. Falls er ein Aggregations-Graph ist, d.h. ein kombinierter Graph bestehend aus Aggregations-Graphen, brauchen wir nur den kleinsten Index vom Aggregations-Graphen zu speichern. Weil der neu erzeugte Graph immer den vorderen Graph ersetzt, ist der kleinste Index seine echte Position. Die neu erhaltene LinkedList „graphs“ wird an die Klassen DrawEachGraph und CreateMainPanel übertragen und ein neues Layout wird im zentralen Bereich generiert.

Dann gibt es den Knopf „SEPARATION“. Wenn der Benutzer einen Aggregations-Graph ausgewählt hat und den Knopf klickt, wird der Graph in mehrere Graphen zerlegt, die vorher den Aggregations-Graphen dargestellt haben. Die Visualisierung wird jetzt wieder zurück verwandelt, wie in der Abbildung 5.5 illustriert. In seinem ActionListener soll das Programm zuerst beurteilen, ob der ausgewählte Graph ein Aggregations-Graph ist. Falls nein, stoppt der Prozess und es erscheint ein Hinweis. Falls ja, wird das Programm den Prozess weiter durchführen. Die Positionen der generierten Graphen werden durch die Indizes bestimmt. Jeder Graph beinhaltet eine LinkedList, die die Indizes der „Subgraphen“ speichert. Für den originalen Graphen beinhaltet seine LinkedList trivialerweise nur einen Index. Für den Aggregations-Graph wird nur der kleinste Index in seiner LinkedList benötigt, denn der kleinste Index beschreibt seine „echte“ Position. Weil die Indizes in jeder LinkedList degressiv geordnet werden, brauchen wir nur den letzten Index in der LinkedList durch „.getLast()“ auszulesen. Der Algorithmus 5.1 beschreibt das Zerlegungsverfahren.

Unter dem Knopf „SEPARATION“ ist der Knopf „MATRIX“ platziert. Falls die Graphen in „Rechtecks“-Form im zentralen Bereich angezeigt werden und der Knopf „MATRIX“ angeklickt wird, werden alle Graphen zu Matrixgraphen ins Rechteckslayout umgewandelt und im zentralen Bereich angezeigt, wie die Abbildung 5.8 illustriert. Falls die Graphen in „Kreis“-Form gezeigt werden und der Knopf „MATRIX“ geklickt wird, werden alle Graphen zu Matrixgraphen in

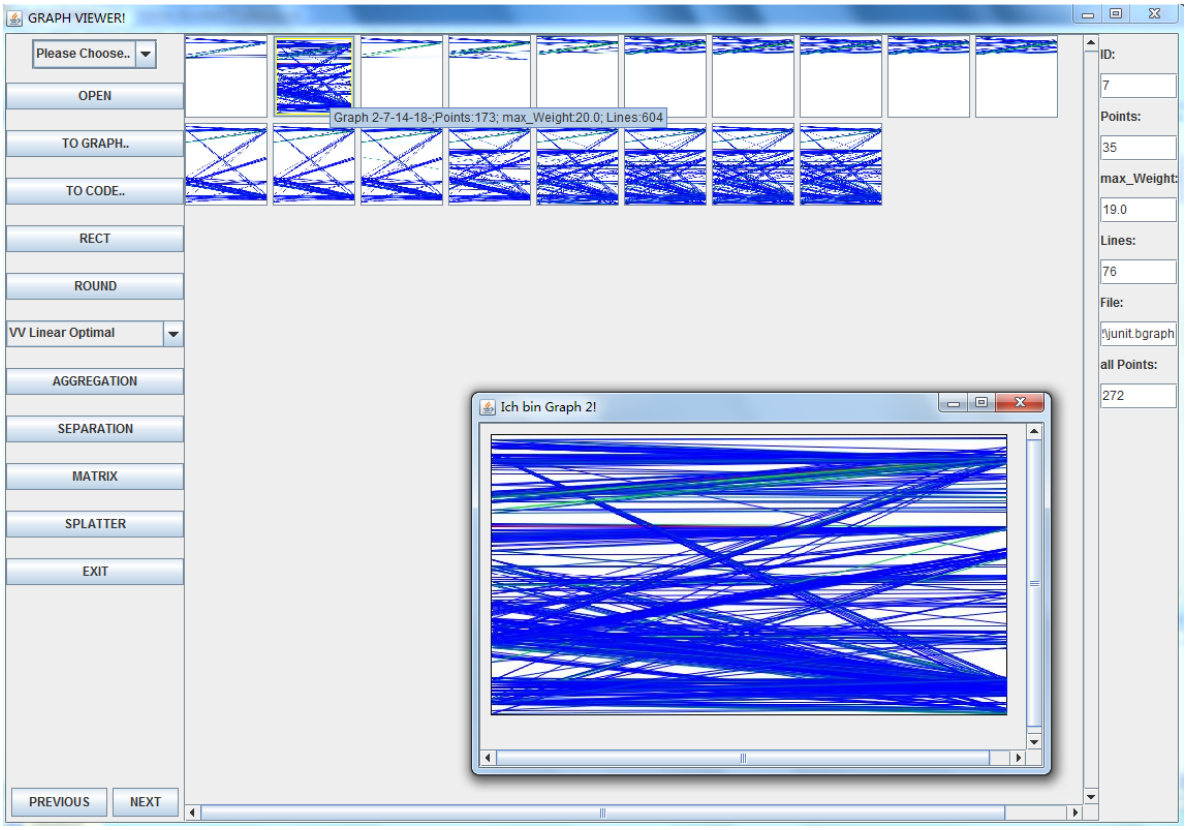


Abbildung 5.7: Der Graph nach der Aggregations-Operation.

der Darstellungsform „Kreislayout“ umgewandelt, wie die Abbildung 5.8 zeigt. Das Programm entscheidet durch die Variable „int tags“, welche Typen von Graphen zu zeichnen sind. Deshalb muss die Variable „int tags“ im ActionListener eingestellt werden. Falls tags=1, bedeutet es, dass die Graphen jetzt in der Darstellungsform Rechteckslayout vorliegen. Dann wird tags=3 zugewiesen und zu „DrawEachGraph“ übergetragen. Das bedeutet, dass das Programm die Matrixgraphen in der Rechtecksform zeichnen wird. Falls tags=2, bedeutet es, dass die Graphen in der Darstellungsform Kreislayout vorliegen. Dann wird tags=4 zugewiesen und demzufolge werden die Matrixgraphen in der Form des Kreislayouts gezeichnet.

Unter dem Knopf „MATRIX“ liegt der Knopf „SPLATTER“. Er realisiert den „Splatting-Effekt“, wie die Abbildungen 5.10 und 5.11 zeigen. Die Abbildung 5.10 zeigt die Visualisierung in der Darstellungsform Rechteck nach Anwendung der Splatting-Operation. Die Abbildung 5.11 zeigt die Splatting-Wirkung beim Kreislayout. Zuerst beurteilt das Programm durch die Variable tags, ob die Matrizen schon vorhanden sind und berechnet wurden. Falls ja, wird die Klasse „DrawSplatter“ angerufen, um die Splatting-Operation durchzuführen. Die Matrizen werden dann als Eingabe aufgefasst. Die Variable tags wird danach ebenfalls neu konfiguriert. Falls nein, wird dem Benutzer ein Hinweis angezeigt.

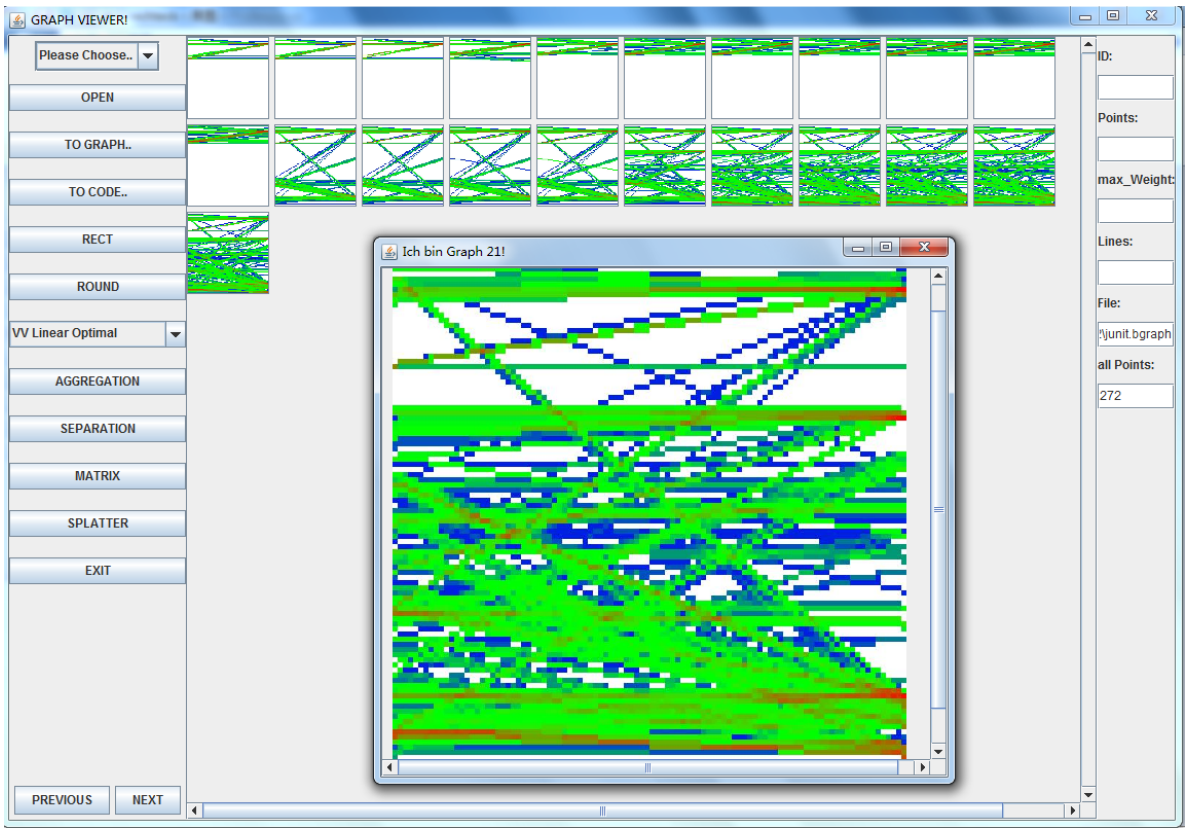


Abbildung 5.8: Matrixgraph der Darstellungsform „Rechteck“

Unter dem Knopf „SPLATTER“ befindet sich der Knopf „EXIT“. Wenn der Benutzer den Knopf anklickt, wird das Programm beendet und alle temporären Dateien entfernt. In seinem ActionListener wird eine selbst definierte Funktion „del()“ aufgerufen, um die temporären Dateien zu entfernen. Der Benutzer kann auswählen, wie viele Graphen in einer Seite einmal gezeigt werden. Deshalb gibt es vielleicht mehrere Seiten, um alle Graphen zu verteilen. Der Benutzer kann mit den Knöpfen „PREVIOUSE“ und „NEXT“ die Graphsequenz durchblättern.

Jetzt wird der Entwurf des zentralen Bereichs vorgestellt. Der zentrale Bereich soll die Inhalte der Eingabedatei zeigen. Darüber hinaus soll der Bereich auch die Graphen der Eingabedatei in einer oder mehreren Seiten zeigen. Deshalb wird das „Card-Layout“ für den zentralen Bereich verwendet. Zuerst wird eine „JTextArea“ benutzt, in der die Inhalte der Eingabedatei angezeigt werden. Die „JTextArea“ wird in einem „JPanel“ integriert. Das „JPanel“ wird im „Card-Layout“ eingebettet. Damit kann man ein oder mehrere „JPanel“s verwalten. Die aus „JCombobox“ zurückgegebene Zahl entscheidet, wie viele „JPanel“s im zentralen Bereich eingefügt werden sollen. Die Knöpfe „TO GRAPH..“, „TO CODE..“, „PREVIOUSE“ und „NEXT“ sind dafür verantwortlich, die oberste Seite mit den anderen Seiten im Card-Layout auszutauschen.

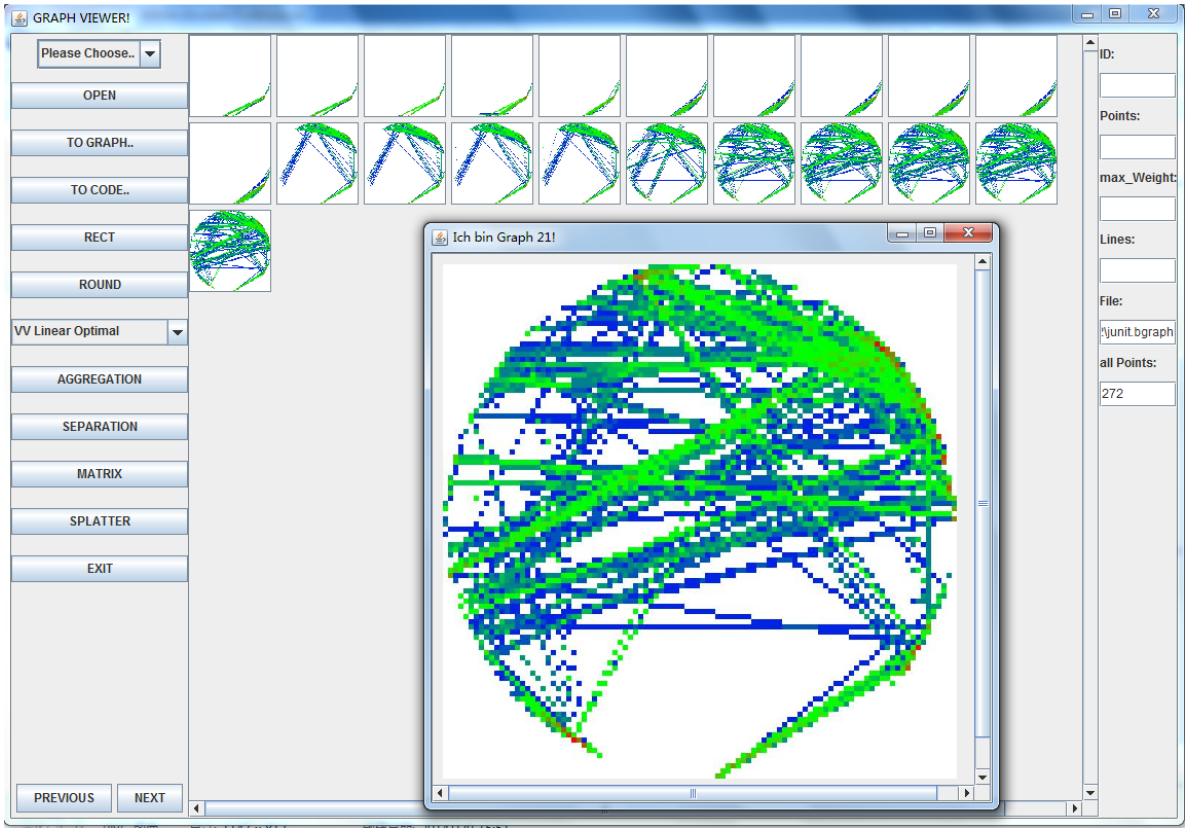


Abbildung 5.9: Matrixgraph der Darstellungsform „Kreis“

Der rechte Bereich zeigt die konkreten Informationen des ausgewählten Graphen. „ID“ zeigt den Index des ausgewählten Graphen. „Points“ und „Lines“ zeigen, wie viele Knoten und Kanten der Graph hat. „max_Weight“ ist das maximale Gewicht des Graphen. Diese Informationen können durch „JTextField“ realisiert werden.

Die Klasse „Colors“ definiert 19 unterschiedliche Farbenschemata. Die Farben werden in einem zweidimensionalen Array „Color[][] colors“ mit RGB-Werten angegeben. Durch die Formel „colors[jcb_F_Index][(colors[jcb_F_Index].length-1)*(Gewicht/max-Gewicht)]“ wird eine Farbe aus dem Intervall ausgewählt. Die erste Dimension „jcb_F_Index“ zeigt, welches Farbenschema ausgewählt wird. Die zweite Dimension ist das Produkt der Länge dieses Intervalls und dem Verhältnis von Gewicht und maximalem Gewicht. Die zweite Dimension bestimmt, welche Farbe in diesem Intervall ausgewählt wird.

Die Klasse „DataReader“ liest die Eingabedatei ein und analysiert diese Datei. In der Klasse werden „FileReader“ und „BufferedReader“ benutzt. DataReader liest zeilenweise ein. Die Inhalte der Eingabedatei werden dabei Zeile für Zeile in einer „LinkedList contents“ gespeichert. Die Indizes aller leeren Zeilen werden in einer „LinkedList num_EL“ gespeichert. Die Anzahl der Graphen, nämlich die Anzahl der leeren Zeilen, wird in der Variable „int count“ gespeichert.

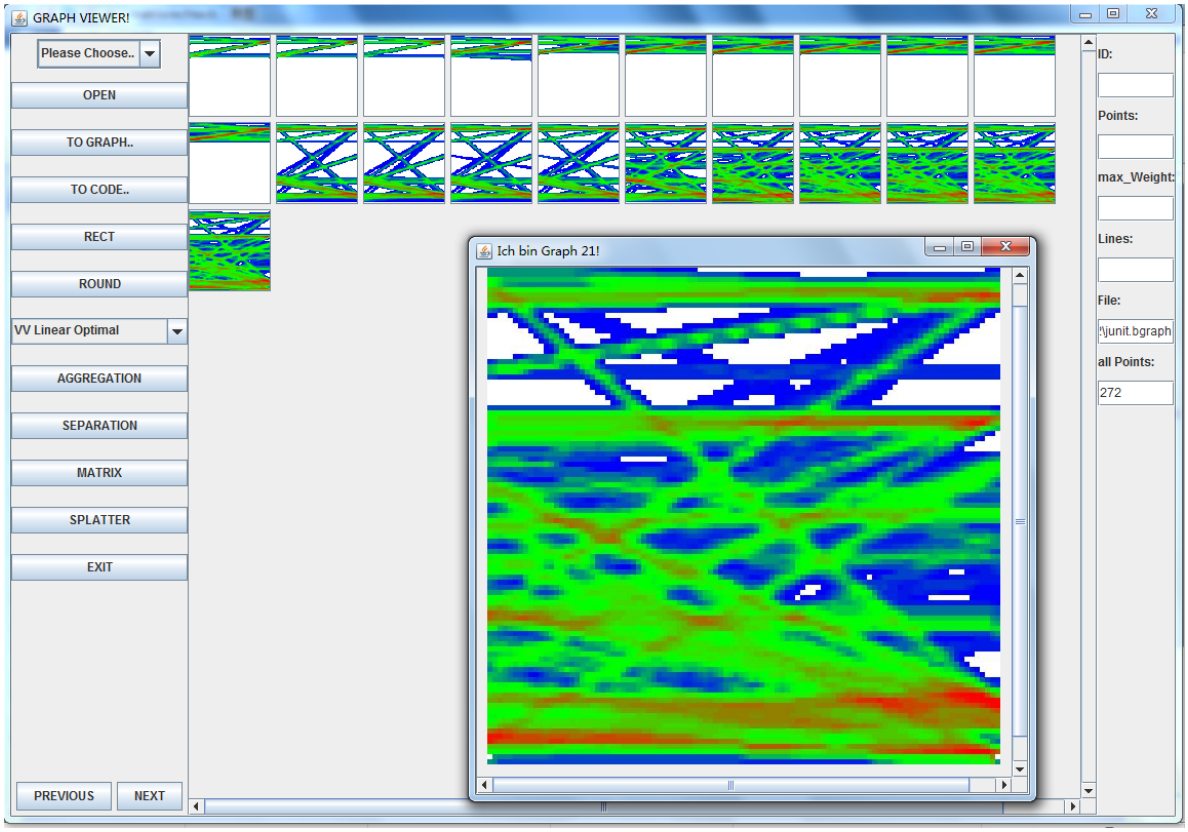


Abbildung 5.10: Splatting der Darstellungsform „Rechteck“

Die 3 oben erwähnten Eigenschaften werden durch die Methode „.getter“ zugegriffen und verändert.

Die Klasse „CreateGraph“ behandelt die in der Klasse DataReader schon analysierte Eingabedatei. Sie kann die nützlichen Inhalte extrahieren und speichert alle Informationen über Graphen der Eingabedatei in einer LinkedList. Im Prozess kann diese Klasse die Zeilen wie etwa „#cobertura-1.9.3“ und „root/net/sourceforge/cobertura“ filtern. Manchmal hat ein Graph nur eine Zeile, z.B. „#cobertura-1.9.3“. Solch einen Graphen bezeichnet man als leeren Graphen. Dieser leere Graph hat keine Kanten, keine Knoten, sondern nur einen Titel. Für einen leeren Graphen ist seine Knotenzahl 0, seine Kantenzahl 0 und seine Gewichte alle 0.0. Die Eingaben der Klasse CreateGraph sind die von der Klasse DataReader analysierten Informationen, nämlich die zeilenweise eingelesene Eigabedatei. Diese Datei wird so betrachtet, dass sie durch die leeren Zeilen in einigen Teilen unterteilt wird. Vor der ersten leeren Zeile befindet sich die Darstellung der Knoten, die wir nicht fokussieren. Zwischen der ersten und zweiten leeren Zeile ist der erste Graph der Eingabedatei zu finden. Tatsächlich ist jeder Teil zwischen zwei leeren Zeilen ein Graph. Deshalb können wir jeden Graph durch die leeren Zeilen relativ leicht bestimmen und extrahieren. Die LinkedList „num_EL“, die die Indizes aller leeren Zeilen speichert, wird von der Klasse DataReader zu der Klasse CreateGraph

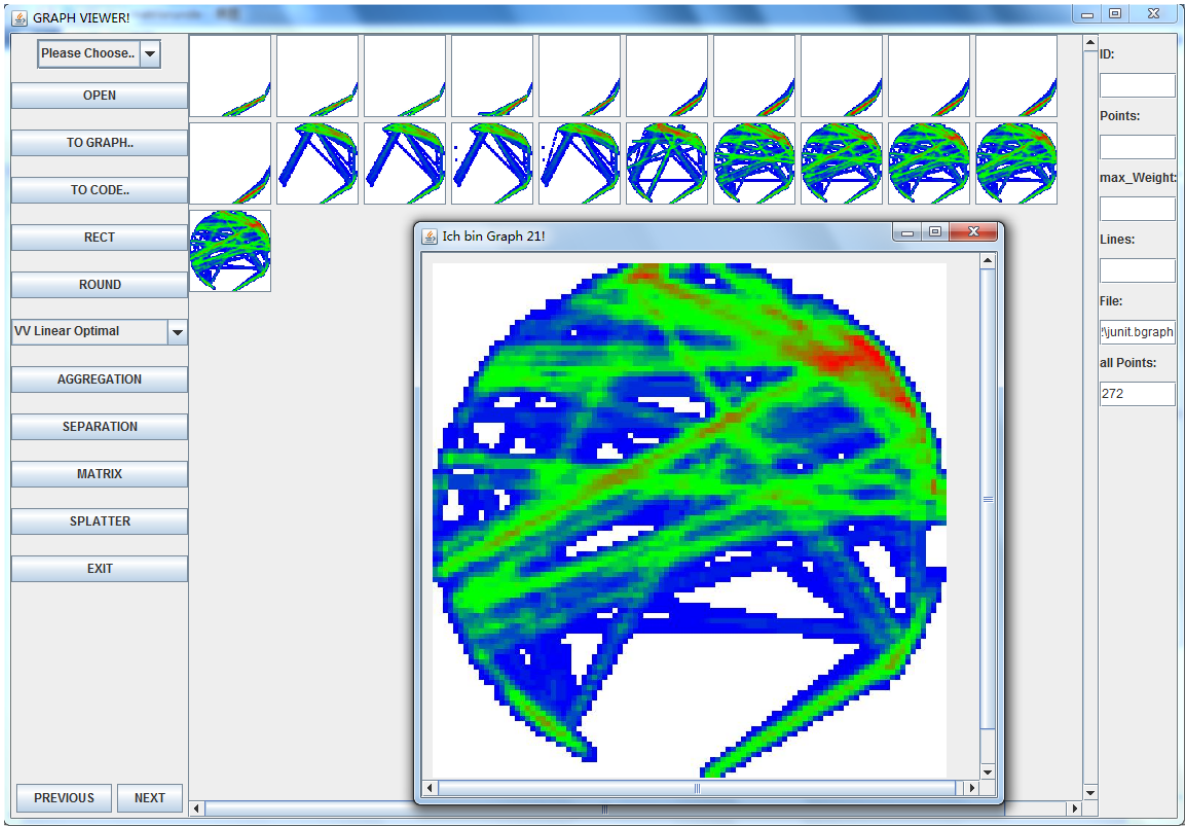


Abbildung 5.11: Splatting der Darstellungsform „Kreis“

übertragen. Jetzt kann das Programm die Inhalte jedes Graphen zwischen zwei leeren Zeilen in einer entsprechenden „for-Schleife“ einlesen. Die Grenzen der for-Schleife müssen hierbei beachtet werden. Weil der Index der Zeile bei 1 beginnt und der Index des Elements in der LinkedList bei 0 beginnt, sind die Grenzen der i-ten for-Schleife [num_EL[i], num_EL[i+1]-2]. Falls num_EL[i]==num_EL[i+1]-2, haben wir es mit einem leeren Graph zu tun. Ansonsten werden die Methoden „readStartPoint()“, „readEndPoint()“ und „readWeight()“ aufgerufen. Die drei Methoden sind selbst definierte Funktionen, die den Startpunkt, den Endpunkt und das Gewicht einer Zeile extrahieren können. Alle ausgelesenen Graphen werden in der LinkedList „graphs“ gespeichert und durch eine „getter“ Methode zurückgegeben.

Die Klasse „CreateMainPanel“ zeichnet das Layout der Graphen im zentralen Bereich, wie die Abbildung 5.5 illustriert. Der Benutzer kann eine Nummer in der Dropdownliste „JComboBox“ auswählen. Die Nummer drückt aus, wie viele Graphen in einem JPanel (einer Seite) gezeigt werden. Zum Beispiel gibt es zusammen 50 Graphen in der Eingabedatei. Die Zahl 10 wird in einer „JComboBox“ ausgewählt. Dann werden 5 JPanels erzeugt und in jedem JPanel werden 10 Graphen angezeigt. Falls 20 ausgewählt wird, werden drei JPanels erzeugt und im ersten und zweiten JPanel werden jeweils 20 Graphen angezeigt, während im dritten JPanel die übrigen 10 Graphen angezeigt werden. Mit der von „JComboBox“ zurückgegebenen Nummer kann das

Programm den obigen Prozess automatisch durchführen und abarbeiten. Der Algorithmus 5.2 auf der nächsten Seite zeigt das Verfahren, wie das Layout vom zentralen Bereich automatisch gezeichnet wird.

Wenn ein Bild eines Graphen auf einen JButton gezeichnet wird, soll das Bild den JButton vollständig „füllen“. Die Dimension des Bildes soll gleich groß sein wie die Dimension des JButtons. Eine einfache Methode, das Layout des zentralen Bereiches zu zeichnen, ist das sogenannte „GridLayout“, das von der Programmiersprache JAVA unterstützt wird. Aber beim „GridLayout“ wird das Layout vom System vorgegeben und gezeichnet. Die Dimensionen aller Komponenten werden deshalb vom System vorberechnet. Das bedeutet, dass wir die Dimension des JButtons nicht so einfach bestimmen können. Deshalb ist es unmöglich, den JButton mit einem Bild zu füllen. Das Bild ist entweder größer oder kleiner als der JButton. Um das Problem zu lösen, können wir zuerst „.setLayout(null)“ anwenden. Es fordert das System auf, das Layout selbst einzustellen. Dann bestimmen wir die Dimension jedes einzelnen JButtons mit „JButton.setBounds(int x,int y,int width, int height)“ und zeichnen alle JButtons selbst auf das JPanel im zentralen Bereich, wie der Algorithmus 5.2 auf der nächsten Seite illustriert. Die Dimension jedes JButtons ist somit algorithmisch bestimmt. Die Dimension wird auch ans Bild jedes Graphen übertragen, so dass die Dimensionen des Bildes und des JButtons gleich sind, so dass ein Bild auch jeweils einen JButton füllen kann.

In dieser Klasse werden die Bilder der Graphen auf die einzelnen JButtons gezeichnet. Die Gesamtzahl der Graphen ist anfänglich unbestimmt. Es ist möglich, dass eine Eingabedatei 100 Graphen umfasst, es könnten aber auch 1000 oder etwa 10000 Graphen sein. Falls die gezeichneten Graphen im Speicher bleiben, verursachen sie vielleicht „java.lang.OutOfMemoryError: Java heap space“, wenn zu viele davon im Speicher existieren. Deshalb gibt das Programm zuerst alle gezeichneten Graphen auf der Festplatte aus (Dies ist die Aufgabe der Klasse „DrawEachGraph“). Die Klasse „CreateMainPanel“ liest diese Bilder von der FestPlatte ein und zeichnet sie mit den Methoden „ImageIO.read()“ und „.setIcon()“ auf die zugehörigen JButtons. Darüber hinaus werden einige ActionListener von JButton benötigt. Wenn der Benutzer ein JButton linksklickt, wird ein Fenster geöffnet, das das konkrete Bild des Graphen anzeigt. Wenn ein JButton rechtsgeklickt wird, wird der JButton ausgewählt. Durch die Methode „mouseClicked(MouseEvent e)“ wird dies realisiert. Die Methoden „.getClickCount()“ und „.getButton()“ von MouseEvent können den Linksklick und den Rechtklick identifizieren und diese Ereignisse intern an entsprechende Klassen weiterleiten. Wenn die Maus sich auf einen JButton bewegt, erscheint ein Hinweis, z.B. der Index des Graphen oder etwa die Anzahl von Punkten und Linien. Diese Funktion wird durch „mouseEntered(MouseEvent e)“ realisiert.

Die Klasse „DrawEachGraph“ zeichnet die konkreten Bilder aller Graphen in den Darstellungsformen „Rechteck“ und „Kreis“. Sie zeichnet auch die Matrixgraphen und die Splatted-Graphen der beiden Darstellungsformen. Seine wichtigste Eingabe ist die „LinkedList graphs“, die alle Graphen abspeichert. Für jeden daraus extrahierten Graph wird die Klasse „OverlappedEdge“ zuerst aufgerufen. Die Klasse kann alle duplizierten Items entfernen, die die gleichen Startknoten und Endknoten haben. Nur ein Item mit den Start- und Endknoten und dem maximalen Gewicht bleibt in der Liste erhalten. Darüber hinaus werden „int tags“, „int jcb_F_Index“ und „double max_Weight“ zusätzlich auch noch eingeführt. „int tags“ identifiziert, ob Rechteckbilder, Kreisbilder oder ihre Matrixgraphen gezeichnet werden. „int jcb_F_Index“ ist der Index

Algorithmus 5.2 Algorithmus für die Graphen im zentralen Bereich

```
procedure MainPanelMalen(int max,int jcb_Index)
                                                    // max: Gesamtzahl von Graphen
                                          // jcb_Index: Rückgabewert von JComboBox
    int num_JP = ((max − 1)/jcb_Index) + 1;
                                                    // num_JP: JPanelszahl
    for all int i ∈ num_JP do
        ein neues JPanel erzeugen;
        seines Layout mit null konfigurieren;
        int a = 1 + (jcb_Index ∗ i);
        int b = jcb_Index + (jcb_Index ∗ i);
                                          // a: der Index vom ersten Graph eines JPanel
                                         // b: der Index vom letzten Graph eines JPanel
        int ten_Buttons = 10;
                                                    // Graphenzahl in jeder Zeile
        int k = 0;
        int l = 0;
                                                    // k, l: Koordinaten zu berechnen
        int n_Line = (jcb_Index − 1)/ten_Buttons + 1;
                                                    // n_Line: Zeilenzahl in einem JPanel
        for all int j ∈ [a, b] do
            if j < max then
                int b_X = (w_B + distance) ∗ k;
                int b_Y = (h_B + distance) ∗ l;
                                          // b_X, b_Y: die Koordinaten eines JButtons
                                                    // w_B: die Breite eines JButtons
                                                     // h_B: die Höhe eines JButtons
                                         // distance: der Abstand zwischen 2 JButtons
                k + +;
                if k == ten_Buttons then
                    k = 0;
                    l + +;
                    JButton mit den Eigenschaften b_X, b_Y,w_B und h_B in JPanel zeichnen;
                end if
            end if
        end for
    end for
end procedure
```

des Farbmodells, der von der Dropdownlist für die Farbschemata zurückgegeben wird. „double max_Weight“ ist das maximale Kantengewicht. Wir benutzen dies, um die Farbe zu berechnen. In dieser Klasse gibt es drei selbst definierte Methoden: „drawSelected()“, „drawRechtangleImage()“ und „drawRoundImage()“. Zuerst wird „drawSelected()“ aufgerufen, die das Zeichen für den ausgewählten Graph erzeugt. Dann wird durch „int tags“ beurteilt, in welcher Darstellungsform die Graphen gezeichnet werden. Falls tags=1, wird „drawRechtangleImage()“ aufgerufen. Falls tags=2, wird „drawRoundImage()“ aufgerufen. Sonst bedeutet „tags“, dass das Programm die Matrixgraphen zeichnen soll. Dann wird die Klasse „DrawMatrix“ aufgerufen, die die Matrixgraphen der Darstellungsformen „Rechteck“ und „Kreis“ generiert. Wenn diese zwei Methoden aufgerufen werden, werden die Farbschemata mit „int jcb_F_Index“ durch die Klasse „Colors“ zuerst berechnet. Dann werden alle Graphen mit „Graphics“ gezeichnet. Wenn es zu viele Graphen gibt, wird es vielleicht das Problem „java.lang.OutOfMemoryError: Java heap space“ verursachen, das oben schon erwähnt wurde. Alle schon gezeichneten Bilder der Graphen werden durch „ImageIO.write()“ auf die Festplatte gespeichert.

Die Klasse „DrawMatrix“ erzeugt die Matrix eines jeden Graphen und zeichnet die entsprechenden Bilder gemäß dieser Matrizen. Diese Klasse wird in der Klasse „DrawEachGraph“ aufgerufen. Die Variable „int tags“ wird zuerst eingeführt. Dadurch kann das Programm entscheiden, welcher Typ von Matrixgraphen gezeichnet werden soll. Dann werden der Graph und die Nummer des Farbschemas (int jcb_F_Index) an die entsprechenden Methoden übergeben. Das Ziel dieser Umwandlung ist es, den eingeführten Graph in eine 100*100 Matrix zu komprimieren. (Normalerweise hat die Eingabedatei viele Punkte, vielleicht einige Tausende. Aber wenn die Eingabedatei weniger als 100 Punkte hat, ist es kein „Komprimieren“, sondern ein „Vergrößern“.)

Zuerst wird ein 2 dimensionales 100*100 Array definiert und initialisiert. Jede Einheit der Matrix beschreibt die Akkumulation der Gewichte der einzelnen Pixel im Graph. Weil alle Gewichte vom Typ „double“ sind, wird jede Einheit der Matrix auch als Typ „double“ definiert und anfänglich mit 0.0 initialisiert. Danach wird eine selbstdefinierte Methode „createMatrix“ aufgerufen. Diese Methode zeichnet die konkreten Matrixgraphen gemäß der Variable tags. Für jede Kante in einem ursprünglichen Graph sind die Koordinaten von Startpunkt und Endpunkt vorher bekannt. Der ursprüngliche Graph besteht aus n*n Pixeln und die Matrix hat insgesamt die Größe 100*100. Durch den Maßstab n/100 können die neuen Koordinaten des Startpunktes und des Endpunktes in der Matrix bestimmt werden. Wenn zwei Punkte in einer Linie bekannt sind, kann diese Linie trivialerweise bestimmt werden. Dadurch kann das Programm ausrechnen, welche Einheiten der Matrix die Kante überqurt. Die Werte (Gewichte) dieser Einheiten werden auch bestimmt. Der Unterschied beider Typen von Matrixgraph ist nur, dass die Koordinaten der Knoten unterschiedlich sind. Die Knoten eines Graphen bilden ein „Rechteck“, während die Knoten des anderen Graphen sich auf einem „Kreisring“ verteilen. Wenn mehrere Kanten eine Einheit überqueren, werden ihre Gewichte akkumuliert. Nachdem alle Kanten behandelt worden sind, wird eine Matrix zurückgegeben. Weil der Graph in der Matrix „komprimiert“ ist, überschneiden sich viele Kanten in dieser Matrix, die sich im originalen Graph nicht überschneiden würden. Deshalb werden die Werte in einigen Einheiten vielleicht plötzlich zu hoch. Wir hoffen, dass die Farben der Kanten relativ kontinuierlich sind. Die Lösung hierfür ist eine „log()“ Operation. Für den Wert jeder Einheit n berechnen wir log(n). Dann ist die Veränderung der Werte nicht so groß. Aber es gibt noch ein Problem.

Falls eine Einheit „1.0“ ist, ist log(1.0)=0.0. Der Wert 0.0 beschreibt die „Backgroundcolor“, nämlich die Farbe „Weiß“. Alle Linien, deren Gewichte 0.0 sind, „verschwinden“ in der Matrix. Darüber hinaus ist es auch möglich, dass das Gewicht einer Einheit kleiner als 1.0 ist, z.B. 0.5. $\log(0.5) < 0$ und es kann nicht als Gewicht zugewiesen werden, weil negative Gewichte bei der Aggregation problematisch sind. Um das Problem zu behandeln, benutzen wir „log(n+1)“ statt „log(n)“. Das kann sicherstellen, dass alle Werte nach der log-Operation nicht kleiner als 0.0 sind. Für alle Einheiten der Matrix berechnen wir „log(n+1)“ und eine neue Matrix wird erhalten. Mit Hilfe dieser Matrix zeichnet das Programm einen neuen Matrixgraph für jeden Graph. Zuerst beurteilt das Programm, ob der Wert einer Einheit der Matrix 0.0 ist. Falls ja, wird die Farbe Weiß angerufen. Falls nein, wird die Klasse „Colors“ aufgerufen.

Die Klasse „DrawSplatter“ gibt eine neue Matrix aus, die auf der in der Klasse „DrawMatrix“ erzeugte Matrix basiert. Das Ziel ist die „Splatting“-Wirkung zu realisieren. Zuerst wird die alte Matrix eingelesen. Der Wert jeder Einheit wird mit den Werten aller ihrer umgebenden Einheiten akkumuliert. Dann wird ihr Durchschnittswert berechnet und zu der entsprechenden gleichen Einheit in der neuen gesplatteten Matrix zugewiesen. Es gibt zwei Methoden in der Klasse: „createSplMatrix()“ und „drawSplGraph()“. Die erste Methode erzeugt die neue Matrix und die zweite Methode zeichnet den „Splatting-Graph“ gemäß der neuen Matrix. Für eine 100*100 Matrix hat jede der vier Einheiten in den vier „Ecken“, nämlich (0,0), (0,99), (99,0) und (99,99), nur drei umgebende Einheiten. (0,0) hat zum Beispiel drei umgebende Einheiten (0,1), (1,0), (1,1), inklusive sich selbst, also vier Einheiten insgesamt. Der Durchschnittswert ist ((0,0)+(0,1)+(1,0)+(1,1))/4. Für die Einheiten an den vier Seiten der Matrix hat jede fünf umgebende Einheiten, inklusive sich selbst, also sechs Einheiten insgesamt. Die Summe aller sechs Werte muss man durch 6 teilen, um den Durchschnittswert zu erhalten. Für alle anderen Einheiten hat jede acht umgebende Einheiten und ihre Summe muss dementsprechend durch 9 geteilt werden. Hat man die neue gesplattete Matrix berechnet, so zeichnet das Programm den „Splatting-Graph“ und speichert ihn auf der Festplatte.

Die Klasse „Graph“ ist eine selbst definierte Datenstruktur. Jeder in einer Eingabedatei angegebene Graph ist ein Objekt dieser Klasse. Sie beinhaltet einige Variablen, um den Graph zu verwalten und zu beschreiben. Die „LinkedList items“ beinhaltet alle Items des Graphen. Die Variable „int flag“ wird mit 0 initialisiert. Nach „AGGREGATION“-Operation ist die Variable flag=1. Dies bedeutet, dass der Graph ein Aggregations-Graph ist. Die „LinkedList merged_Gs“ speichert alle Graphen, die diesen Graph kombinieren. Darüber hinaus gibt es auch die entsprechenden „getter“- und „setter“-Methoden.

Die Klasse „Item“ ist auch eine selbst definierte Datenstruktur. In der Eingabedatei besteht jeder Graph aus „Items“. Zum Beispiel ist „166 162 1.0“ ein einzelnes Item, in diesem Fall eine gewichtete Kante des Graphen. Jedes Item beinhaltet einen Startpunkt, einen Endpunkt und ein Gewicht. Diese Klasse umfasst auch diese drei Eigenschaften und ihre entsprechenden „getter“- und „setter“-Methoden.

Die Klasse „MyPanel“ realisiert das Popup-Fenster, wenn ein Graph im zentralen Bereich angeklickt wird. Diese Klasse expandiert das „JPanel“, liest ein Image von der Festplatte ein und zeichnet es mit Hilfe der „paintComponent()“ Methode.

Die Klasse „DrawSplatter“ realisiert die Splatting-Operation. Die Eingabe ist die in der Matrix-Operation erzeugte Matrix. Dann wird ein Box-Filter für jede Einheit der Matrix angewandt. Es gibt drei unterschiedlichen Fälle: Behandlung der Einheit auf den Kanten, Behandlung der Einheit in der Ecke und Behandlung aller anderen Einheiten (siehe Abschnitt 4.3.4 auf Seite 27). Durch die „splatted“ Matrix wird ein neuer Splatting-Graph gezeichnet und zurückgegeben.

Die Klasse „OverlappedEdge“ behandelt die duplizierten Items, die die gleichen Startknoten und Endknoten haben. Die Eingabe ist stets ein Graph. Für jeden eingegebenen Graph wird jedes Item mit allen anderen Items verglichen. Falls zwei Items die gleichen Start- und Endknoten haben, werden ihre Gewichte verglichen. Das Item mit dem größeren Gewicht wird erhalten bleiben, während das andere gelöscht wird. Zum Schluss wird nur ein Item mit dem maximalen Gewicht erhalten bleiben. Nachdem alle Items behandelt worden sind, wird der neue Graph zurückgegeben.

6 Fallstudien

In diesem Abschnitt werden die 9 gegebenen Eingabedateien mit dem Programm eingelesen und visualisiert. Das Programm kann alle Graphen, die in den Eingabedateien enthalten sind, in beiden Darstellungsformen Rechteckslayout und Kreislayout anzeigen. Die Interaktion zwischen Benutzer und Graphen ist ebenfalls realisiert. Die Änderungen in der Evolution der dynamischen Graphen können auch leicht durch die statische Visualisierung durch einfaches Betrachten ohne Interaktion schon voranalysiert werden.

Zuerst betrachten wir die Eingabedatei „wicket_method.bgraph“. Die Abbildung 6.1a) zeigt alle Graphen der Datei in der Darstellungsform des Rechtecklayouts. Durch Betrachten der Abbildung 6.1a) können wir sehen, dass diese Datei 14 Graphen und 8799 Knoten beinhaltet. Der erste Graph hat 2388 Knoten, 3195 Kanten und sein maximales Gewicht ist 1.0. Der zweite Graph hat 2398 Knoten, 3208 Kanten und das maximale Gewicht ist auch 1.0. Der dritte Graph hat 2326 Knoten, 3139 Kanten und sein maximales Gewicht ist 1.0. Der vierte Graph hat 2401 Knoten, 3211 Kanten und sein maximales Gewicht verändert sich nicht. Die Änderungen zwischen diesen 4 Graphen sind nicht groß. Ihre Knotenzahlen schwanken um eine Anzahl von 2390, ihre Kantenzahlen liegen bei etwa 3200. Ab dem fünften Graph nehmen die Knoten und Kanten zu. Die Knotenzahlen der Graphen 5 und 6 sind 3141 und 3147. Ihre Kantenzahlen sind 4334 und 4351. Im Vergleich zu Graph 1, Graph 2, Graph 3 und Graph 4 ist die Zunahme noch nicht so offensichtlich. Aber ab Graph 7 bis zum Graph 14 ist die Zunahme deutlich zu erkennen. Die Anzahlen der Knoten von Graph 7 bis zum Graph 14 sind jeweils 4693, 5004, 5108, 5207, 5216, 5217, 5220 und 5222. Ihre Knotenzahlen werden von 4693 auf 5222 ständig erhöht. Die Anzahlen der Kanten sind jeweils 7981, 8692, 8958, 9126, 9146, 9150, 9156 und 9163. Ihre Kantenzahlen erhöhen sich auch von 7981 auf 9163. Mit Hilfe der Abbildung 6.1a) können wir sagen, dass die Knoten und Kanten des dynamischen Graphen in der Datei „wicket_method.bgraph“ ständig zunehmen und ihre Gewichte sich nicht verändern. Die Abbildung 6.2a) zeigt die Matrixgraphen in der Darstellungsform des Rechtecklayouts. Durch die Matrixgraphen können wir diese Veränderungen noch etwas klarer sehen. Wegen der Akkumulation der Gewichte (siehe Abschnitt 4.3.3 auf Seite 25) stellen die verschiedenen Farben die unterschiedlichen Gewichte dar. Von Graph 1 bis 4 gibt es noch viele blaue Kanten. Aber in Graph 5 und 6 werden die blauen Kanten immer weniger. In Graph 7 bis zu 14 können wir keine blauen Kanten mehr sehen. Nur die Farben Grün und Rot bleiben übrig. Der Grund liegt darin, dass die Knoten und Kanten immer dichter und dichter werden. Die Abbildung 6.2b) zeigt alle „splatted“ Graphen. Nach der „splatting“-Operation ist die Veränderung deutlicher, so dass wir das gleiche Ergebnis wie oben erhalten werden. Die Abbildung 6.1b) zeigt die Graphen in der Darstellungsform Kreislayout. Die Knotenzahlen und Kantenzahlen sind gleich der oben erwähnten Zahlen. Die ständige Zunahme der Knoten und Kanten in den Abbildungen ist auch hier deutlich zu sehen. Die Abbildung 6.3a) zeigt die entsprechenden Matrixgraphen.

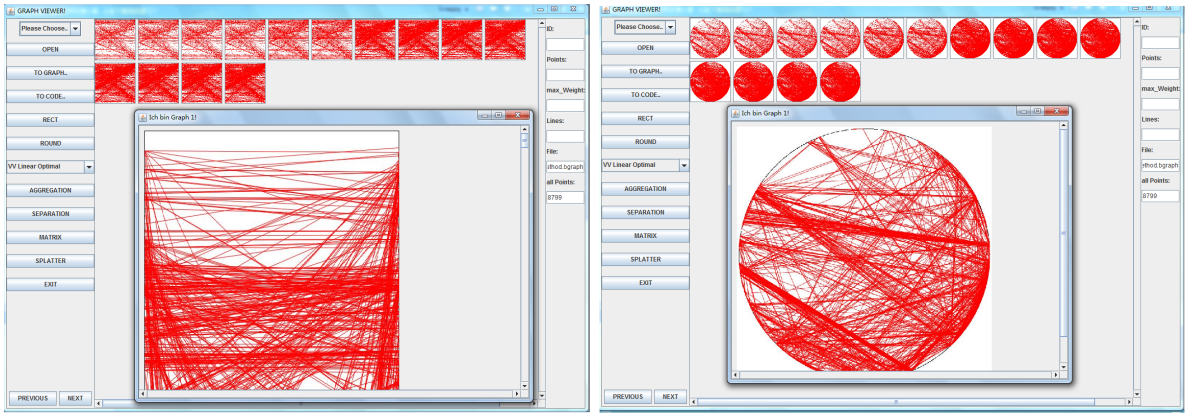


Abbildung 6.1: a) Die Darstellungsform Rechteckslayout der Graphen im Datensatz „wicket_method.bgraph“. b) Die Darstellungsform Kreislayout der Graphen vom Datensatz der Datei „wicket_method.bgraph“.

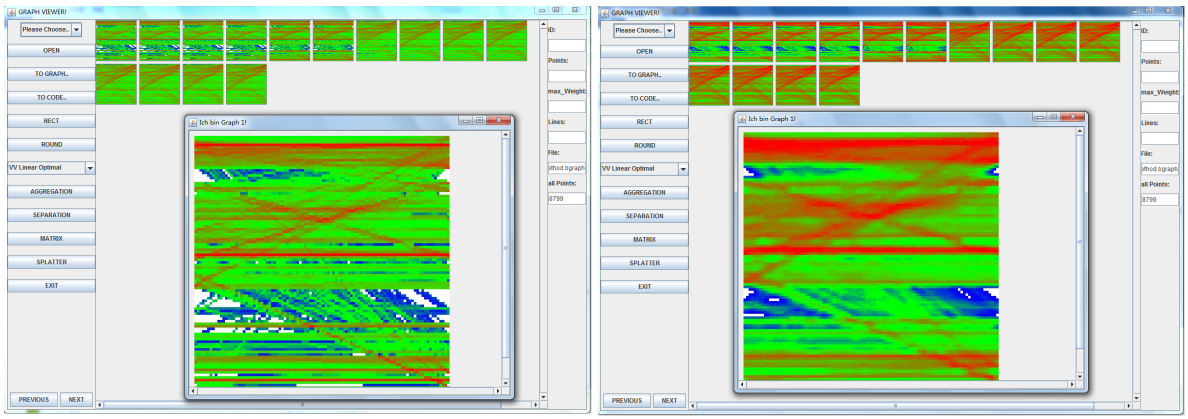


Abbildung 6.2: a) Die Matrixgraphen in der Darstellungsform Rechteckslayout vom Datensatz „wicket_method.bgraph“. b) Die splatted Graphen in der Darstellungsform Rechteckslayout von „wicket_method.bgraph“.

Die grüne Farbe wird deutlich stärker in dieser Abbildung. Das bedeutet, dass immer mehr Kanten eine Einheit der Matrix überqueren. Ihre Gewichte werden akkumuliert. Deshalb tritt die blaue Farbe immer weiter in den Hintergrund. Die Abbildung 6.3b) zeigt die entsprechenden „splatted“ Graphen. Die Veränderungen der dynamischen Graphen sind in den „splatted“ Graphen deutlicher zu erkennen.

Jetzt sehen wir die Eingabedatei „junit.bgraph“. Die Abbildung 6.4a) zeigt die Rechteck Darstellungsform ihrer Graphen. Die Datei hat 21 Graphen und nur 272 Knoten insgesamt,

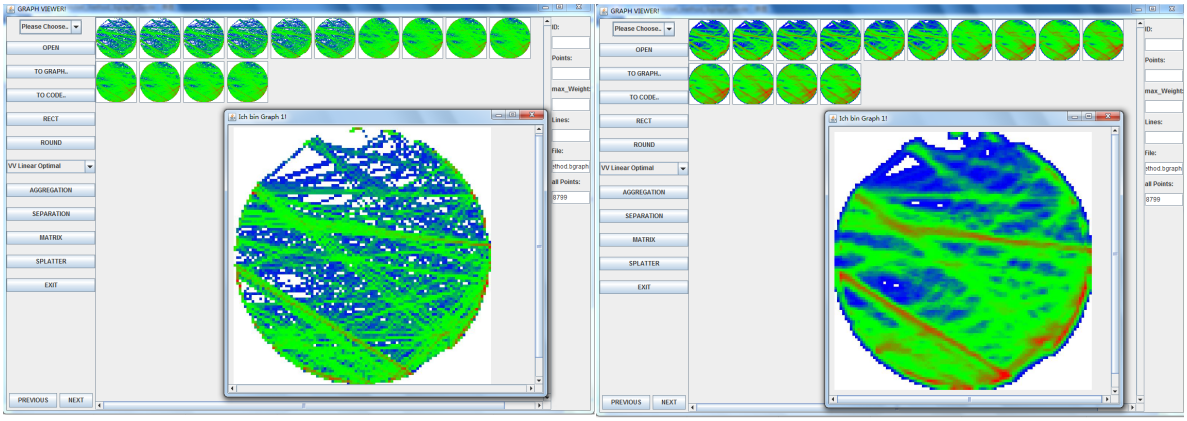


Abbildung 6.3: a) Die Matrixgraphen der Darstellungsform Kreislayout des Datensatzes „wicket_method.bgraph“. b) Die splatted Graphen der Darstellungsform Kreislayout von „wicket_method.bgraph“.

die viel weniger als die Knotenzahl bei „wicket_method.bgraph“ ist. Die Graphen 1, 2 und 3 haben jeweils 17 Knoten und 29 Kanten. Aber ihre maximalen Gewichte sind unterschiedlich. Das maximale Gewicht von Graph 1 ist 8.0. Das maximale Gewicht von Graph 2 ist 9.0, während das maximale Gewicht von Graph 3 11.0 ist. Die Knoten, Kanten und das maximale Gewicht von Graph 4 nehmen ein wenig zu. Die 3 Zahlen sind jeweils 25, 48, 18.0. Die Anzahl der Knoten von Graph 5 sinkt auf 22 und seine Kantenanzahl sinkt um 1 auf 47. Aber das maximale Gewicht verändert sich nicht. Beim Graph 6 nimmt die Anzahl an Knoten zu, von 22 auf 35. Die Anzahl an Kanten erhöht sich von 47 auf 74. Das maximale Gewicht steigert sich auf 19.0. Die Graphen 7 und 8 sind fast identisch. Nur die Anzahl der Kanten nimmt auf 76 zu. Die anderen verändern sich nicht. Die Graphen 9, 10 und 11 sind auch identisch. Die Knotenzahl, die Kantenzahl und das maximale Gewicht sind jeweils 37, 81, 15.0. In der Periode geschehen keine Änderungen über die Evolution der Graphen. Beim Graph 12 verändert sich der dynamische Graph sehr. Vom Graph 1 bis zum Graph 11 konzentrieren sich alle Kanten auf das vordere Drittel der Knoten. Aber nach Graph 12 treten auch Kanten in den anderen zwei Drittel der Knoten auf. Der Graph 12 hat 58 Knoten, 161 Kanten und sein maximales Gewicht ist 11.0. Beim Graph 13 nehmen die Knotenzahl auf 59 und die Kantenzahl auf 163 zu. Beim Graph 14 erhöhen sich die Anzahlen von Knoten und Kanten jeweils auf 61 und 166. Beim Graph 15 sind die Knotenzahl und Kantenzahl jeweils 62 und 170. Vom Graph 12 bis zum Graph 15 verändert sich das maximale Gewicht nicht, welches immer noch bei 11.0 liegt. Beim Graph 16 ist die Änderung wieder groß. Offensichtlich erhöht sich hier die Anzahl der Kanten immens. Der Graph 16 hat 103 Knoten und 253 Kanten. Das maximale Gewicht erhöht sich auch, dies ist nun 20.0. Vom Graph 17 bis zum Graph 20 steigern sich die Knotenzahlen und Kantenzahlen stetig. Ihre Knotenzahlen sind 127, 133, 145 und 146, während ihre Kantenzahlen 307, 333, 355 und 360 sind. Diese Änderung existiert zwar, ist aber nicht so deutlich zu sehen in der Visualisierung. Ihre maximalen Gewichte bleiben gleich und liegen immer bei 20.0. Der Graph 20 und der Graph 21 sind identisch. Die Abbildung 6.4b)

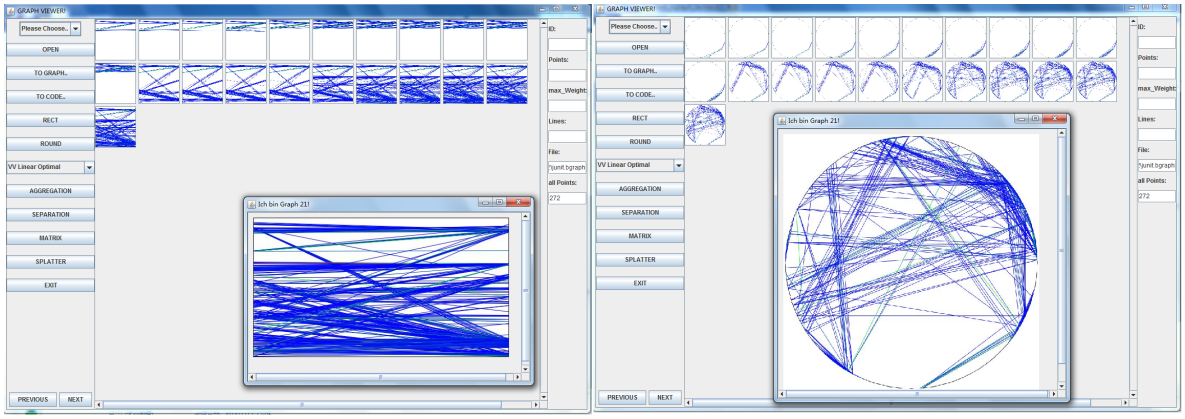


Abbildung 6.4: a) Die Graphen der Darstellungsform Rechteckslayout von „junit.bgraph“.
b) Die Graphen der Darstellungsform Kreislayout von „junit.bgraph“.

zeigt deren Darstellungsform im Kreislayout. Die Analyse geschieht hier sehr ähnlich. Die Knotenzahlen und Kantenzahlen in der Sequenz der Graphen nehmen ständig zu. Trotzdem sind die Veränderungen nicht so deutlich zu sehen. Dennoch verändert sich etwas beim Graph 12, denn dort wird der Graph der Sequenz plötzlich deutlich größer. In den anderen Perioden verändert sich der Graph „sukzessive“.

Die dritte Eingabedatei ist „JHotDraw.bgraph“. Die Abbildung 6.5a) zeigt alle Graphen in der Darstellungsform Rechteckslayout. Die Datei hat 1496 Knoten, aber nur 6 Graphen. Der Graph 1 hat 279 Knoten und 5214 Kanten. Das maximale Gewicht ist 4.0. Beim Graph 2 verringern sich die beiden Anzahlen von Knoten und Kanten. Die Knotenzahl sinkt auf 165, während die Kantenzahl auf 2560 sinkt. Die Veränderung kann in der Abbildung 6.5a) leicht beobachtet werden. Das maximale Gewicht vermindert sich auf 3.0. Beim Graph 3 werden die Knotenanzahl und Kantenanzahl wieder auf 269 und 4872 erhöht. Das maximale Gewicht nimmt auf 6.0 zu. Wir können die roten Kanten deutlich sehen. Der Graph 4 hat 282 Knoten und 5078 Kanten. Sein maximales Gewicht bleibt gleich. Obwohl die Knotenzahl und die Kantenzahl ein wenig zunehmen, können wir durch das Programm beobachten, dass einige Kanten gelöscht werden und andere neue Kanten darin hinzugefügt werden. Die Änderungen von Graph 5 sind auch offensichtlich. Sowohl die Knotenzahl als auch die Kantenzahl vermindern sich heftig. Die Anzahl der Knoten sinkt von 282 auf 46. Die Anzahl der Kanten verringert sich von 5078 auf 716. Das maximale Gewicht sinkt von 6.0 auf 1.0. Der Graph 6 ist ein „leerer“ Graph, der keine Knoten und Kanten enthält. Die Abbildung 6.5b) zeigt die entsprechende Darstellungsform Kreislayout. Das Ergebnis der Analyse ist ähnlich. Die grünen Kanten in den Graphen 3 und 4 sind jedoch offensichtlicher als sie in der Darstellungsform des Rechtecklayouts zu sehen waren. Die Änderungen der dynamischen Graphen in der Datei „JHotDraw.bgraph“ sind offensichtlich und können leicht beobachtet werden.

Die vierte Eingabedatei ist „iText.bgraph“. Diese Datei beinhaltet 5 Graphen und 972 Knoten. Die Abbildung 6.6a) zeigt die Darstellungsform des Rechtecklayouts aller Graphen. Der Graph

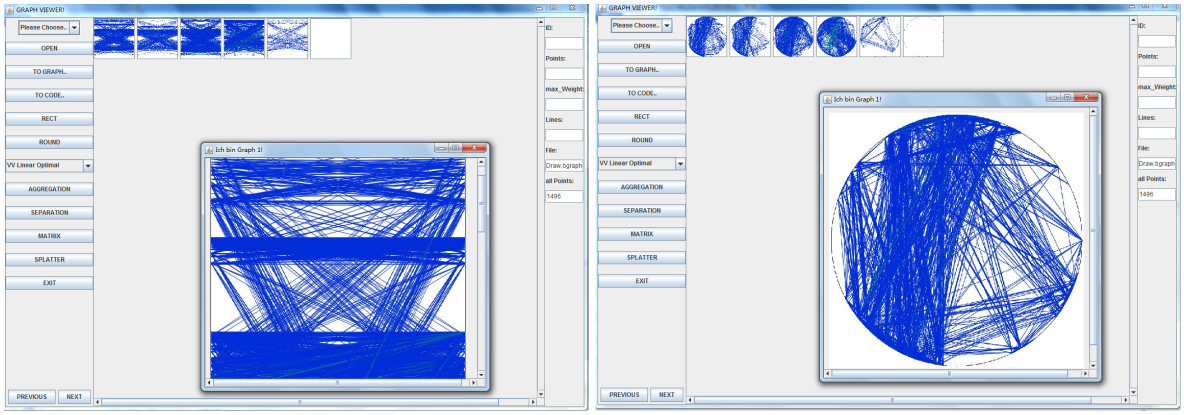


Abbildung 6.5: a) Die Graphen der Darstellungsform des Rechtecklayouts von „JHotDraw.bgraph“. b) Die Graphen der Darstellungsform im Kreislayout des Datensatzes „JHotDraw.bgraph“.

1 hat 115 Knoten und 1114 Kanten. Das maximale Gewicht ist 5.0. Der Graph 2 hat 159 Knoten und 2286 Kanten. Die Anzahl der Kanten wird fast verdoppelt. Das maximale Gewicht steigert sich auch auf 10.0. Mit Hilfe der Abbildung können wir beobachten, dass alle Kanten von Graph 1 und Graph 2 nur die hintere Hälfte der Knoten betreffen. Beim Graph 3 nimmt die Knotenzahl auf 200 und die Kantenzahl auf 2318 zu. Die Zunahme der Kanten ist nicht so extrem. Aber in der Abbildung 6.6a) können wir sehen, dass viele Kanten gelöscht werden und viele neue Kanten hinzugefügt werden. Die neu hinzugekommenen Kanten betreffen die vordere Hälfte der Knoten. Das maximale Gewicht sinkt auf 8.0. Beim Graph 4 ist die Änderung auch sehr offensichtlich. Der Graph 4 hat nur 24 Knoten und 132 Kanten. Die meisten Kanten in Graph 3 werden entfernt. Sein maximales Gewicht sinkt auf 3.0. Der letze Graph ist ein „leerer“ Graph, der weder Kanten noch Knoten hat. Die Abbildung 6.6b) zeigt alle Graphen in der Drastellungsform des Kreislayouts. Die dynamischen Graphen verändern sich im Laufe der Zeit eindeutig. Die Abbildung zeigt die Änderungen des dynamischen Graphen sehr klar.

Die fünfte Eingabedatei ist „cobertura_method_deleted.bgraph“. Die Abbildung 6.7a) zeigt die Darstellungsform des Rechtecklayouts aller Graphen. In der Abbildung können wir sehen, dass alle Gewichte 1.0 sind. Das maximale Gewicht bleibt immer gleich. Deshalb wird es hier nicht diskutiert. Die Eingabedatei „cobertura_method_deleted.bgraph“ hat 14 Graphen und 5000 Knoten. Aber für jeden einzelnen Graphen gibt es nicht viele Kanten. Im Vergleich mit den oberen Dateien ist es offensichtlich, dass die Kanten jedes Graphen sehr „spärlich“ sind. Der Graph 1 hat 280 Knoten und 343 Kanten. Der Graph 2 hat 213 Knoten und 266 Kanten. Die Veränderung ist nicht groß. Der Graph 3 hat nur 62 Knoten und 66 Kanten. In dieser Abbildung wird die Veränderung klar repräsentiert. Die Knotenzahl und die Kantenzahl von Graph 4 nehmen zu, die jeweils 156 und 226 sind. Der Graph 4 ist ähnlich wie Graph 1 und Graph 2. Im Vergleich mit Graph 3 ist die Veränderung offensichtlich zu erkennen. Der Graph 5 hat 42 Knoten und 51 Kanten. Beim Graph 6 verringern sich die Knotenzahl und Kantenzahl weiter. Die Anzahl der Knoten sinkt auf 19 und die Anzahl der Kanten vermindert sich auf

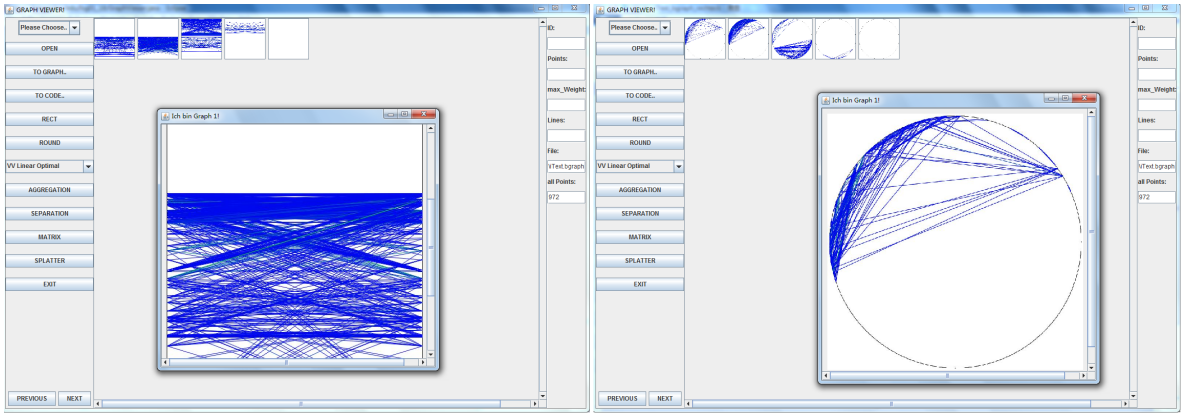


Abbildung 6.6: a) Die Graphen der Darstellungsform des Rechtecklayouts von „iText.bgraph“. b) Die Graphen der Darstellungsform des Kreislayouts von „iText.bgraph“.

16. Es ist klar, dass der dynamische Graph sich vom Graph 4 bis zum Graph 6 immer weiter „verkleinert“. Der Graph 7 hat 55 Knoten und 52 Kanten. Aber beim Graph 8 sinken die Knotenzahl und die Kantenzahl wieder auf 6 und 4. Beim Graph 9 nehmen die Knotenzahl auf 59 und die Kantenzahl auf 63 zu. Der Graph 10 hat 29 Knoten und 22 Kanten. Vom Graph 7 bis zum Graph 10 außer dem Graph 8 verändern sich die Graphen nicht so stark. Beim Graph 11 geschieht eine plötzliche Zunahme der Knotenzahl und der Kantenzahl. Die beiden Anzahlen sind jeweils 578 und 783. Dann tritt mit Graph 12 ein „leerer“ Graph auf. Der Graph 13 hat 81 Knoten und 89 Kanten. Beim Graph 14 sinken die beiden Anzahlen auf 23 und 21. Die Veränderungen vom Graph 11 bis Graph 14 sind immer groß. Die Abbildung 6.7b) zeigt alle Graphen in der Darstellungsform des Kreislayouts. Im Vergleich zur Abbildung 6.7a) repräsentiert die Abbildung in der Darstellungsform des Kreislayouts die Veränderungen der dynamischen Graphen nicht so gut. Der Grund liegt darin, dass die Kanten in einem kleinen Bereich verteilt werden. Zum Beispiel vom Graph 1 bis zum Graph 7 konzentrieren sich alle Kanten im rechten kleinen Bereich. Darüber hinaus gibt es nur wenige Kanten. Der Graph 8 hat 6 Knoten und 4 Kanten. Seine Knoten und Kanten sind extrem wenig, sodass er sehr ähnlich zum Graph 12 ist, der ein leerer Graph ist.

Die sechste Eingabedatei ist „cobertura_method.bgraph“. Die Abbildung 6.8a) zeigt alle Graphen in der Darstellungsform des Rechtecklayouts. Diese Datei enthält auch 14 Graphen und 5000 Knoten insgesamt, wie bei der Eingabedatei „cobertura_method_deleted.bgraph“. Die Gewichte aller Graphen bleiben gleich, die immer 1.0 sind. Deshalb werden die maximalen Gewichte jedes Graphen nicht diskutiert. Die Veränderungen von Graph 1, Graph 2 und Graph 3 sind nicht groß. Ihre Anzahlen der Knoten sind jeweils 280, 208 und 201. Ihre Anzahlen der Kanten sind 343, 272 und 275. Der Graph 4 hat 188 Knoten und 267 Kanten. In der Abbildung können wir sehen, dass einige Kanten zwischen den „hinteren“ Knoten gelöscht werden. Dann werden einige neue Kanten zwischen den „vorderen“ Knoten hinzugefügt. Im Vergleich zum

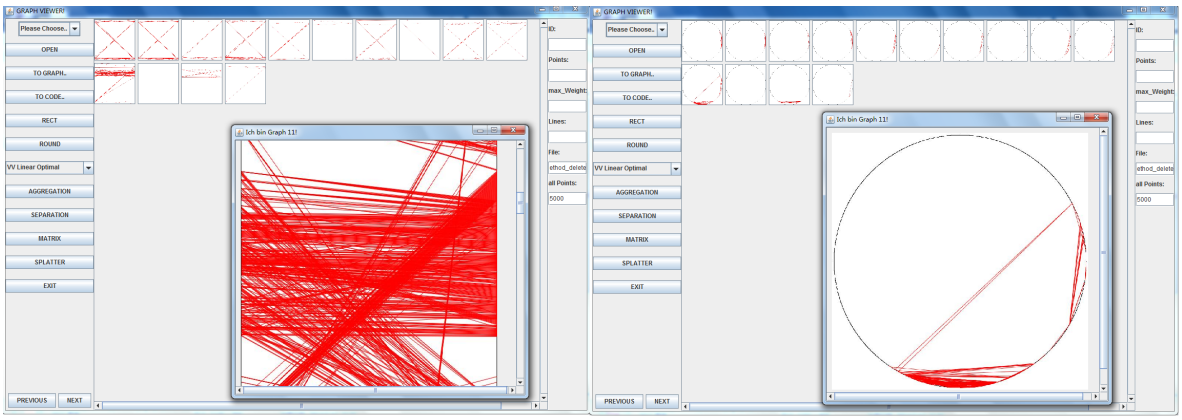


Abbildung 6.7: a) Die Graphen in der Darstellungsform des Rechtecklayouts von „cobertura_method_deleted.bgraph“. b) Die Graphen in der Darstellungsform des Kreislayouts von „cobertura_method_deleted.bgraph“.

Graph 4 sind die Veränderungen von Graph 5, Graph 6, Graph 7 und Graph 8 auch nicht besonders groß. Ihre Knotenzahlen sind jeweils 205, 225, 252 und 254. Ihre Kantenzahlen sind 277, 312, 399 und 403. Die Zunahmen von Kanten und Knoten geschehen zwar ständig aber sind nicht groß. Aber beim Graph 9 vermehren sich die Knotenzahl auf 926 und die Kantenzahl auf 1918. Im weiteren Verlauf der Evolution verändern sich Graph 10, 11 und 12 auch ein wenig. Ihre Anzahl an Knoten sind 1021, 1274 und 1289. Ihre Kantenzahlen sind jeweils 2053, 2714 und 2730. Die nächste große Änderung geschieht beim Graph 13. Die Anzahl der Knoten nimmt auf 3068 zu. Die Kantenzahl erhöht sich auf 7612. Der Graph 14 hat 3086 Knoten und 7633 Kanten. Im Vergleich zum Graph 13 ist die Zunahme nicht groß. Die Abbildung 6.8b) zeigt alle Graphen in der Darstellungsform des Kreislayouts. Die Änderungen von dynamischen Graphen bei Graph 4, Graph 9 und Graph 13 sind sehr offensichtlich und können durch die Visualisierung der Graphen leicht beobachtet werden.

Die siebte Eingabedatei ist „cobertura_method_added.bgraph“. Die Abbildung 6.9a) zeigt alle Graphen in der Darstellungsform des Rechtecklayouts. Diese Datei hat auch 5000 Knoten und 14 Graphen. Die Gewichte aller Kanten verändern sich auch nicht, sie bleiben immer bei 1.0. Durch den Überblick in der Abbildung 6.9a) können wir erkennen, dass der dynamische Graph sich im Laufe der Zeit stark verändert. Der Graph 1 hat 280 Knoten und 343 Kanten. Der Graph 2 hat 142 Knoten und 195 Kanten. Mit Hilfe des Visualisierungsprogramms können wir sehen, dass viele Kanten gelöscht werden. Beim Graph 3 verringern sich die Kanten und Knoten weiter. Die Knotenzahl und Kantenzahl sind 61 und 69. Aber beim Graph 4 nehmen die Knotenzahl und Kantenzahl wieder zu. Die beiden Zahlen sind jeweils 145 und 218. Beim Graph 5 sinken die Anzahlen von Knoten und Kanten wieder, die sich auf 57 und 61 vermindern. Der Graph 6 hat 56 Knoten und 51 Kanten. Die Schwankung ist nicht groß. Beim Graph 7 steigert sich die Knotenzahl auf 99. Die Anzahl der Kanten erhöht sich auch auf 139. Der Graph 8 hat nur 14 Knoten und 8 Kanten. Aber beim Graph 9 nehmen die Knotenzahl auf

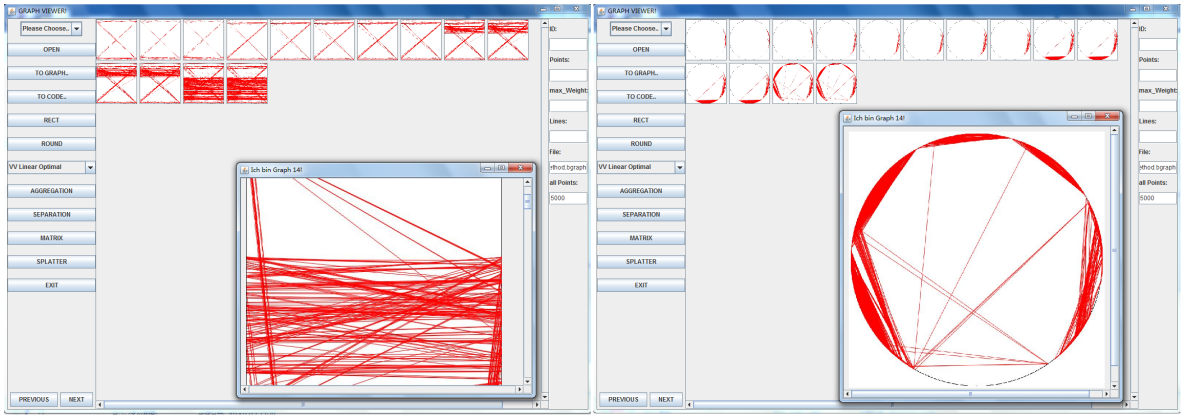


Abbildung 6.8: a) Die Graphen in der Darstellungsform des Rechtecklayouts vom Datensatz „cobertura_method.bgraph“. b) Die Graphen in der Darstellungsform des Kreislayouts vom Datensatz „cobertura_method.bgraph“.

743 und die Kantenzahl auf 1578 zu. Die Zunahmen von Kanten und Knoten sind hier sehr offensichtlich. Der Graph 10 „verkleinert“ sich offenbar. Die Anzahlen der Knoten und Kanten sinken auf 137 und 157. Beim Graph 11 erhöhen sich diese beiden Zahlen wieder. Der Graph 11 hat 841 Knoten und 1444 Kanten. Der Graph 12 hat nur 24 Knoten und 12 Kanten. Der Graph 12 „verkleinert“ sich wieder. Beim Graph 13 werden die Knotenzahl auf 2657 und die Kantenzahl auf 7001 erhöht. Eine große Zunahme findet hier statt. Der Graph 14 hat 41 Knoten und 42 Kanten. Die Reduzierung der Kanten und Knoten ist sehr deutlich zu sehen. Die Abbildung 6.9b) zeigt die Darstellungsform im Kreislayout für alle Graphen. Die Analyse geschieht hier ähnlich und wir erhalten das gleiche Ergebnis wie oben. Der dynamische Graph in der Datei „cobertura_method_added.bgraph“ verändert sich heftig im Laufe der Zeit. Die Abbildung 6.9a) repräsentiert diese Änderung deutlicher als die Abbildung 6.9b).

Die nächste Eingabedatei ist „clustered_dsn_hourly.bgraph“. Die Abbildung 6.10a) zeigt alle Graphen in der Darstellungsform des Rechtecklayouts. Diese Eingabedatei hat 59 Graphen. Diese Anzahl ist nicht so groß, aber größer als die in allen oben erwähnten Dateien. Die maximalen Gewichte sind immer 1.0. In der Abbildung 6.10a) können wir sehen, dass die ganze Abfolge der Graphen in drei Gruppen geteilt wird: vom Graph 2 bis zum Graph 12, vom Graph 25 bis zum Graph 37 und vom Graph 49 bis zum Graph 59. Jeder Graph in den drei Gruppen hat einige Hunderte Kanten. Zwischen den 3 Gruppen sind leere Graphen oder die Graphen, die nur 2 oder 3 Kanten besitzen. Wenn die ganze Reihenfolge der Graphen in einem Panel gezeigt wird, sind die Änderungen der Graphen sehr offensichtlich. Die Abbildung 6.10b) zeigt das entsprechende Kreislayout. Die Anzahlen der Kanten in der ersten Gruppe verteilen sich meistens zwischen 522 und 946. Die Kantenzahlen in der zweiten Gruppe sind zwischen 300 und 661. Diese Zahlen schwanken in der dritten Gruppe zwischen 463 und 901. Manchmal nimmt die Kantenzahl eines Graphen auf über 1000 zu. Aber sein entsprechendes Bild verändert sich nicht sonderlich stark. Der Grund liegt darin, dass solche Graphen viele mehrfache, dublizierte

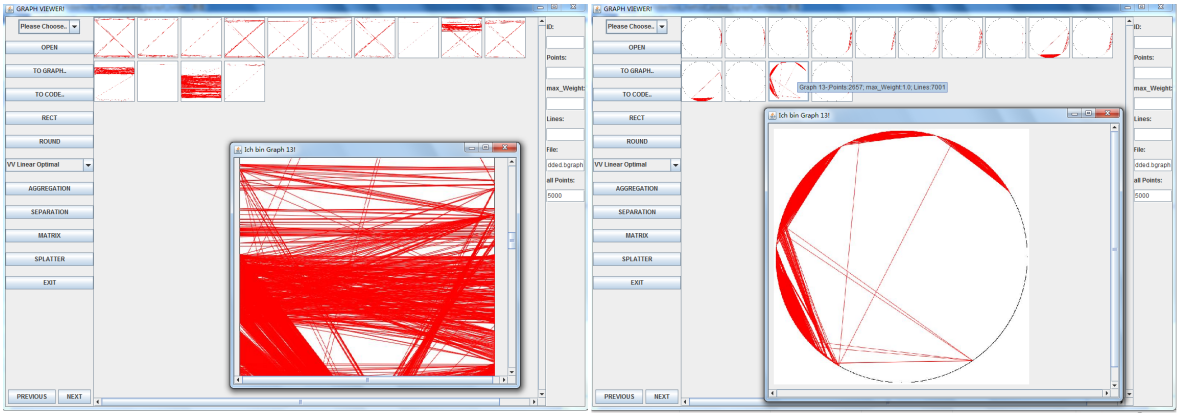


Abbildung 6.9: a) Die Graphen in der Darstellungsform des Rechtecklayouts von „cobertura_method_added.bgraph“. b) Die Graphen in der Darstellungsform des Kreislayouts von „cobertura_method_added.bgraph“.

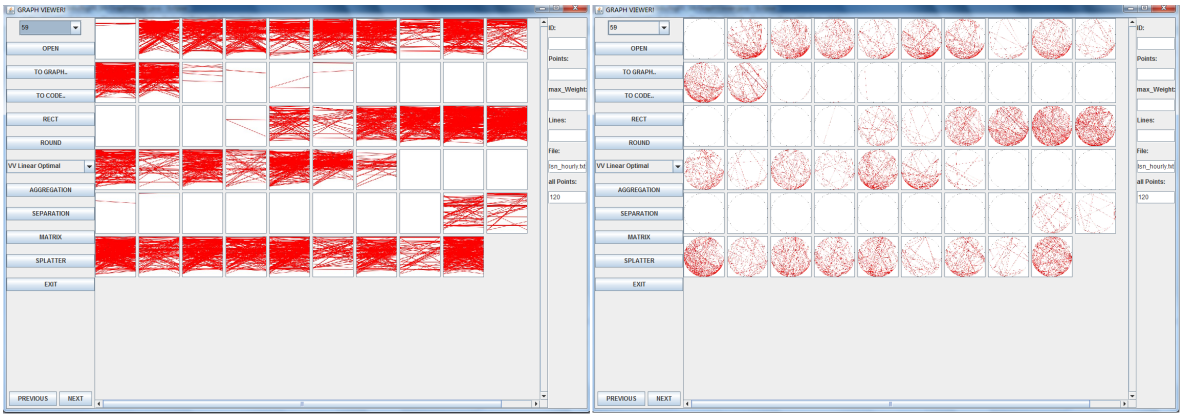


Abbildung 6.10: a) Die Graphen der Darstellungsform im Rechtecklayout von „clustered_dsn_hourly.bgraph“. b) Die Graphen der Darstellungsform im Kreislayout von „clustered_dsn_hourly.bgraph“.

Kanten haben. Zum Beispiel besitzt Graph 3 genau 49 Kanten in der Form „21 16 1.0“. Diese Kanten nennt man auch Multikanten, wie sie bei Multigraphen auftreten.

Die letzte Eingabedatei ist „clustered_dynamic_social_network.bgraph“. Die Gewichte sind immer 1.0. Deshalb wird das maximale Gewicht nicht diskutiert. Die Abbildung 6.11a) zeigt das Rechteck-Layout aller Graphen. Diese Eingabedatei enthält 1180 Graphen. Diese Anzahl ist viel größer als die Anzahl in den oben erwähnten Eingabedateien. In der Abbildung 6.11a) werden 250 Graphen in jeder Seite gezeigt. Trotzdem braucht man 5 Seiten, um alle Graphen einigermassen Clutter-frei und deutlich zu repräsentieren. Hier wird nur die erste Seite gezeigt.

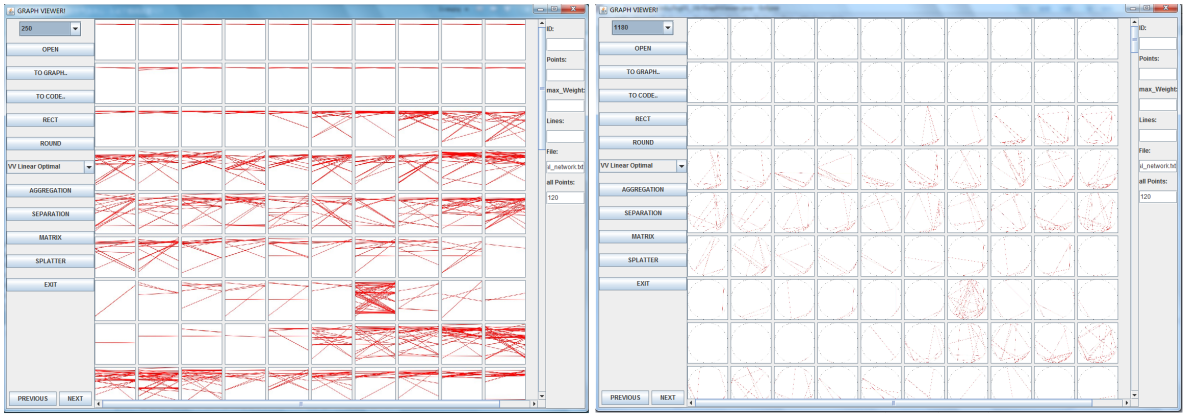


Abbildung 6.11: a) Die Graphen in der Darstellungsform Rechtecklayout von „clustered_dynamic_social_network.bgraph“. b) Die Graphen in der Darstellungsform Kreislayout von „clustered_dynamic_social_network.bgraph“.

Vom Graph 1 bis 24 sind die Veränderungen nur gering. Die Knotenzahl erhöht sich von 2 auf 4. Die Kantenzahl ist immer unter 18. Vielleicht haben Sie schon beobachtet, dass die Knotenzahl viel niedriger als die Kantenzahl ist. Zum Beispiel hat der Graph 1 nur 2 Knoten aber 8 Kanten. Wie können 2 Knoten 8 Kanten haben? Der Grund liegt darin, dass Graph 1 exakt 8 gleiche Kanten „13 15 1.0“ besitzt wie dies bei einem Multigraph der Fall ist. In Abbildung 6.11a) können wir auch sehen, dass es nur eine Kante in Graph 1 gibt. Vom Graph 25 bis zum Graph 66 sind die Änderungen nicht groß aber beobachtbar. Graph 67 verändert sich plötzlich stark. Die Anzahl der Knoten erhöht sich von 4 auf 31 und die Kantenzahl erhöht sich von 4 auf 64. Beim Graph 68 verkleinert sich dann der dynamische Graph wieder. Die Knoten und Kanten vermindern sich auf 7 und 5. Bis zum Graph 76 vergrößert sich der dynamische Graph wieder. Solche Schwankungen dauern bis zum Graph 242. Keine heftigen Änderungen geschehen dann mehr. Aber alle Änderungen sind durch die Visualisierungtechnik in dieser Diplomarbeit klar und deutlich zu erkennen. Von Graph 243 bis 479 sind fast alle Graphen „leere“ Graphen. Von Graph 480 bis 726 verändern sich die Graphen weiter. Ab Graph 727 sind die Graphen wieder leere Graphen. Ab Graph 964 hören die leeren Graphen zunächst auf. Es gibt drei Schwankungen, wobei die Knoten und Kanten stark zunehmen: von Graph 1040 bis 1047, von Graph 1079 bis 1102 und von Graph 1167 bis 1177. Viele Veränderungen der Graphen sind schwer formulierbar. Aber durch die Visualisierung können wir die Veränderungen klar beobachten. Die Abbildung 6.11b) zeigt die Graphen im Kreislayout. Um einen guten Überblick zu bekommen, werden alle Graphen in dieser Abbildung in einem Panel gezeigt. In dieser Abbidlung werden nur die vorderen Graphen repräsentiert. Der Benutzer kann den Scrollbar benutzen, um die anderen Graphen zu sehen. Alle 1180 Graphen werden auf einmal gezeigt. Wir können die Änderungen des dynamischen Graphen im Laufe der Zeit klar beobachten. Durch diese Visualisierung kann der Benutzer leicht erkennen, wie die Knoten und Kanten sich verändern und wann die heftigen Änderungen sich ereignen.

Normalerweise hat die Darstellungsform des Kreislayouts den besseren Überblick als die Darstellungsform des Rechtecklayouts. Aber wenn der Startknoten und der Endknoten einer Kante sehr nah beieinander sind, wird die Kante für das Kreislayout schwieriger zu beobachten sein. Dies kann man in den Abbildungen 6.9b), 6.7b) und 6.11b) leicht erkennen. Zum Beispiel bedeutet die Zeile „1 2 1.0“ eine Kante vom ersten Knoten zum zweiten Knoten, deren Gewicht 1.0 ist. Solche Kanten sind im Kreislayout fast nur als ein Punkt sichtbar, weil der erste Knoten und der zweite Knoten zu nah beieinander liegen. In diesem Fall ist das Rechteck-Layout besser geeignet als das Kreislayout. Aber wenn es viele Kanten gibt und diese Kanten sich relativ gleichmäßig verteilen, wie z.B. in der Abbildung 6.5 gezeigt, dann funktioniert das Kreislayout deutlich besser.

7 Diskussion

Die meisten „for-Schleifen“ sind für die Extraktion eines Graphen aus der LinkedList verantwortlich oder etwa für das Einlesen der Inhlate der Eingabedatei „line-by-line“. Zum Beispiel ist die Zeitkomplexität der Klasse DataReader O(n), wobei n die Zeilenanzahl der Eingabedatei darstellt. Die Komplexität der Klasse DrawEachGraph ist O(m*n), wobei m die Graphenanzahl und n die Itemanzahl eines Graphen ist.

Normalerweise sind die Komplexitäten noch nicht zu hoch, wenn diese Zahlen nicht zu groß sind. Dieser Aufwand ist auch nötig, aber die Klasse OverlappedEdge behandelt die Kanten, die gleichen Startknoten und Endknoten haben (siehe die Abschnitte 5.3.2 auf Seite 34). Sie muss hierzu jedes Item mit allen anderen Items vergleichen. Die Zeitkomplexität ist $O(n^2)$, wobei n die Itemanzahl eines Graphen beschreibt. Wenn ein Graph viele Items enthält, ist n sehr groß. Die Komplexität $O(n^2)$ wird dann sehr hoch sein. Zum Beispiel enthält die Datei „wicket_method.bgraph“ mehrere Graphen, deren Itemanzahl über 9000 liegt. Für solch einen Graph ist seine Komplexität O(9000*9000), die offensichtlich sehr hoch ist. Um das Problem zu lösen, können wir vielleicht einen anderen Algorithmus implementieren, aber man wird nicht um das Einlesen aller Daten herumkommen. Aus diesem Grund bleibt diese Komplexität in der Regel immer bestehen. An der Datenspeicherung kann man allerdings in der Zukunft noch arbeiten. Zur Zeit werden hier viele LinkedLists verwendet, die eine hohe Zugriffszeit auf die einzelnen Elemente verursachen.

Alle Graphen in der Arbeit sind gerichtete Graphen. Die Darstellungsform des Rechtecklayouts kann die Richtung einer Kante gut anzeigen. Alle Kanten sind von der linken Seite zur rechten Seite gerichtet. Aber die Darstellungsform des Kreislayouts kann die Richtung nicht so gut darstellen. Wir können nicht unterscheiden, ob eine Kante (u,v) in der Darstellungsform des Kreislayouts von u zu v oder von v zu u verläuft. In der Zukunft soll die Darstellungsform des Kreislayouts verbessert werden und eventuell mit einer weiteren Darstellungsform erweitert werden.

Das Programm kann die Graphen in den Formen Rechteck und Kreis in eine „Matrix“ umwandeln (siehe den Abschnitt 4.3.3 auf Seite 25). Je feiner die Matrix verteilt wird, umso genauer ist eine solche Transformation. Aber der Aufwand ist auch höher und die Laufgeschwindigkeit ist deutlich langsamer. Die Balance zwischen Genauigkeit und Laufgeschwindigkeit muss man sorgfältig abwiegen.

JAVA hat eine Besonderheit, dass der Bunutzer die „Java Garbage Collection(GC)“ normalerweise nicht kontrollieren kann. Er kann nur JAVA GC beim System „angeben“. Die temporären Dateien werden schon auf der Festplatte gespeichert, um den Speicheraufwand zu sparen. Aber wenn die Eingabedatei zu viele Graphen besitzt, z.B. einige Tausend Graphen, ist es möglich, dass der Fehler „java.lang.OutOfMemoryError: Java heap space“ auftritt. Um das

Problem zu vermeiden, können die Argumente beim Ablauf des JAVA Programms angewandt werden, die die von JAVA erzeugte Größe des Speichers konfigurieren, wie z.B. „-Xmx2048m -Xms2048m -Xmn256m -Xss16m“. Dies ist tatsächlich eine Lösung, mehre Speicherplätze an JAVA Programme zu verteilen.

8 Zusammenfassung und Ausblick

Dynamische Graphen treten in vielen Anwednungsgebieten auf. Aufrufgraphen im Bereich des Software Engineerings verändern sich über die Zeit. Kontakte in sozialen Netzwerken zeichnen sich dadurch aus, dass sie einem ständigen Wechsel unterworfen sind - Personen kommen neu hinzu und verlassen ein Teilnetzwerk auch wieder. Um einen guten Überblick über die dynamischen Verhaltensweisen in solchen zeitbasierten Daten zu bekommen, kann man sich Visualisierungstechniken zu Nutze machen. Insbesondere für dynamische Graphen bieten sich sogenannte Small Multiples Darstellungen an, die möglichst viele Graphen nebeneinander auf einen Blick zeigen und somit einen guten Vergleich der einzelnen Zeitschritte bieten. In dieser Diplomarbeit soll eine Visualisierungstechnik entwickelt werden, mit der man in längeren Graphsequenzen browsen kann und in der immer ein bestimmter benutzer-definierter Zeitbereich angezeigt wird. Die einzelnen Graphen werden hierbei mit existierenden Knoten-Kanten- oder Matrix-Darstellungen angezeigt. Interaktionstechniken erlauben es dem Benutzer, die Daten in verschiedenen Zeitgranularitäten zu betrachten, indem Aggregationstechniken auf den Graphen angewendet werden. In einer Fallstudie wird die Benutzbarkeit der neuen Visualisierung verdeutlicht.

Durch Visualisierung werden die abstrakten Daten und ihre Zusammenhänge in einer graphischen erfassbaren Form gezeigt. Der Zweck ist, die akstrakten, komplexen und eintönigen Daten für die Menschen leicht verständlich zu machen. Um dieses Ziel zu realisieren, gibt es verschiedene Methoden. In dieser Arbeit werden alle Graphen in den Darstellungsformen des Rechtecklayouts und des Kreislayouts repräsentiert.

Aber Yee und Fisher's Arbeit fokussiert ein anderes Layout, z.B. das radiale Layout in der Darstellungsform des „Ausstrahlens“ [YFDH01] vorliegt, das im Unterschied zum Kreislayout in dieser Arbeit ist. Einige Arbeiten [CG95, RMC91, LR96] diskutieren, wie die Bäume und ihre „parent-children“ Beziehungen durch die Visualisierung repräsentiert werden. Fekete's Arbeit [FWD$^+$03] zerlegt einen Graph in einen Baum und andere Kanten. Dann wird die Datenstruktur der„Treemap“ angewandt, um einen Graph zu visualisieren. Es gibt auch eine Arbeit [SČG05], die die Interaktionen zwischen Systemen und Menschen in der Visualisierung besonders diskutiert. Alle Graphen in der Arbeit sind 2D, während Bruß und Frick's Forschung [BF96] sich auf 3D-Graphenvisualisierung konzentriert. Diese Forschungen über die Visualisierungstechniken in unterschiedlichen Richtungen haben sich schon bewährt, so dass diese für die Informationsvisualisierung in der heutigen Zeit unentbehrlich sind. Tatsächlich kann eine Redensart die Wichtigkeit der Visualisierung schön beschreiben: „*One Graph Is Worth A thousand Words.*“ [LS87]

Literaturverzeichnis

[AL09] P. O. Ayoo, J. T. Lubega. A requirements analysis framework for Human Activity Systems (HAS): the case of online learning. *Strengthening the Role of ICT in Development*, S. 14, 2009. (Zitiert auf Seite 31)

[BD08] M. Burch, S. Diehl. Timeradartrees: Visualizing dynamic compound digraphs. In *Computer Graphics Forum*, Band 27, S. 823–830. Wiley Online Library, 2008. (Zitiert auf den Seiten 12 und 14)

[BEW95] R. A. Becker, S. G. Eick, A. R. Wilks. Visualizing Network Data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):16–28, 1995. (Zitiert auf Seite 12)

[BF96] I. Bruß, A. Frick. Fast interactive 3-D graph visualization. In *Graph Drawing*, S. 99–110. Springer, 1996. (Zitiert auf Seite 65)

[BFBD10] M. Burch, M. Fritz, F. Beck, S. Diehl. TimeSpiderTrees: A Novel Visual Metaphor for Dynamic Compound Graphs. In *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, S. 168–175. 2010. (Zitiert auf Seite 12)

[BN11] U. Brandes, B. Nick. Asymmetric Relations in Longitudinal Social Networks. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2283–2290, 2011. (Zitiert auf Seite 15)

[BVB+11] M. Burch, C. Vehlow, F. Beck, S. Diehl, D. Weiskopf. Parallel edge splatting for scalable dynamic graph visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2344–2353, 2011. (Zitiert auf den Seiten 12 und 14)

[CG95] I. F. Cruz, A. Garg. Drawing graphs by example efficiently: Trees and planar acyclic digraphs. In *Graph Drawing*, S. 404–415. Springer, 1995. (Zitiert auf Seite 65)

[DBETT99] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999. (Zitiert auf den Seiten 12 und 14)

[DG02] S. Diehl, C. Görg. Graphs, They Are Changing. In *Proceedings of International Symposium on Graph Drawing*, S. 23–30. 2002. (Zitiert auf Seite 15)

[Ead84] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984. (Zitiert auf Seite 12)

[Ead92] P. Eades. Drawing Free Trees. *Bulletin of the Institute for Combinatorics and its Applications*, 5(2):10–36, 1992. (Zitiert auf Seite 14)

[FR91] T. M. J. Fruchterman, E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. (Zitiert auf Seite 12)

[FT08] Y. Frishman, A. Tal. Online Dynamic Graph Drawing. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):727–740, 2008. (Zitiert auf Seite 15)

[FWD+03] J.-D. Fekete, D. Wang, N. Dang, A. Aris, C. Plaisant. Overlaying graph links on treemaps. In *IEEE Symposium on Information Visualization Conference Compendium (demonstration)*, Band 5. 2003. (Zitiert auf Seite 65)

[G+] O. M. GROUP, et al. OMG Unified Modeling Language TM (OMG UML), Superstructure v2. 3 [documento en línea]. 2010. (Zitiert auf Seite 33)

[GBD09] M. Greilich, M. Burch, S. Diehl. Visualizing the evolution of compound digraphs with TimeArcTrees. In *Computer Graphics Forum*, Band 28, S. 975–982. Wiley Online Library, 2009. (Zitiert auf Seite 14)

[Ged13] B. Gedat. *Java - Sprache und Programming.* Open Source Press, 2013. (Zitiert auf Seite 21)

[GFC04] M. Ghoniem, J.-D. Fekete, P. Castagliola. A Comparison of the Readability of Graphs Using Node-Link and Matrix-Based Representations. In *Proc. IEEE Symposium on Information Visualization*, S. 17–24. 2004. (Zitiert auf Seite 13)

[GGK+11] R. Gove, N. Gramsky, R. Kirby, E. Sefer, A. Sopan, C. Dunne, B. Shneiderman, M. Taieb-Maimon. NetVisia: Heat Map and Matrix Visualization of Dynamic Social Network Statistics and Content. In *Proceedings of IEEE Conference on Social Computing*, S. 19–26. 2011. (Zitiert auf Seite 15)

[HE05] W. Huang, P. Eades. How people read graphs. In *Proc. Asia-Pacific Symposium on Information Visualisation*, S. 51–58. 2005. (Zitiert auf Seite 13)

[HHE05] W. Huang, S.-H. Hong, P. Eades. Layout Effects on Sociogram Perception. In *Proc. Symposium on Graph Drawing*, S. 262–273. 2005. (Zitiert auf Seite 13)

[HIWF11] D. Holten, P. Isenberg, J. J. van Wijk, J.-D. Fekete. An Extended Evaluation of the Readability of Tapered, Animated, and Textured Directed-Edge Representations in Node-Link Graphs. In *Proc. IEEE Pacific Visualization Symposium*, S. 195–202. 2011. (Zitiert auf den Seiten 11 und 13)

[HW09] D. Holten, J. J. van Wijk. A user study on visualizing directed edges in graphs. In *Proc. SIGCHI Conference on Human Factors in Computing Systems*, S. 2299–2308. 2009. (Zitiert auf den Seiten 11 und 13)

[KK89] T. Kamada, S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989. (Zitiert auf Seite 12)

[LR96] J. Lamping, R. Rao. The hyperbolic browser: A focus+ context technique for visualizing large hierarchies. *Journal of Visual Languages & Computing*, 7(1):33–55, 1996. (Zitiert auf Seite 65)

[LS87] J. H. Larkin, H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–100, 1987. (Zitiert auf Seite 65)

[Mem02] A. M. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88, 2002. (Zitiert auf Seite 22)

[Ora13] Oracle. Java™ Platform, Standard Edition 7 API Specification, 2013. URL http://docs.oracle.com/javase/7/docs/api/index.html. (Zitiert auf den Seiten 21 und 22)

[PCA02] H. C. Purchase, D. Carrington, J.-A. Allder. Empirical Evaluation of Aesthetics-based Graph Layout. *Empirical Software Engineering*, 7(3):233–255, 2002. (Zitiert auf Seite 13)

[PCJ96] H. C. Purchase, R. F. Cohen, M. James. Validating Graph Drawing Aesthetics. In *Proc. Symposium on Graph Drawing*, S. 435–446. 1996. (Zitiert auf Seite 13)

[Pur97] H. C. Purchase. Which Aesthetic has the Greatest Effect on Human Understanding? In *Proc. Symposium on Graph Drawing*, S. 248–261. 1997. (Zitiert auf Seite 13)

[RLMJ05] R. Rosenholtz, Y. Li, J. Mansfield, Z. Jin. Feature Congestion: A Measure of Display Clutter. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, S. 761–770. 2005. (Zitiert auf den Seiten 12 und 14)

[RMC91] G. G. Robertson, J. D. Mackinlay, S. K. Card. Cone trees: animated 3D visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, S. 189–194. ACM, 1991. (Zitiert auf Seite 65)

[Roy70] W. W. Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, Band 26. Los Angeles, 1970. (Zitiert auf Seite 31)

[RS07] D. Rosenberg, M. Stephens. *Use Case Driven Object Modeling with UMLTheory and Practice*. Apress, 2007. (Zitiert auf Seite 33)

[SČG05] M.-A. D. Storey, D. Čubranić, D. M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proceedings of the 2005 ACM symposium on Software visualization*, S. 193–202. ACM, 2005. (Zitiert auf Seite 65)

[SWS10] K. Stein, R. Wegener, C. Schlieder. Pixel-Oriented Visualization of Change in Social Networks. In *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining*, S. 233–240. 2010. (Zitiert auf Seite 15)

[TA07] Tech-Algorithm.com. Box Filtering, 2007. URL http://tech-algorithm.com/articles/boxfiltering/. (Zitiert auf Seite 27)

[TMB02] B. Tversky, J. B. Morrison, M. Bétrancourt. Animation: Can it Facilitate? *International Journal on Human-Computer Studies*, 57(4):247–262, 2002. (Zitiert auf Seite 15)

[WPCM02] C. Ware, H. Purchase, L. Colpoys, M. McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002. (Zitiert auf Seite 13)

[YEL10] J. S. Yi, N. Elmqvist, S. Lee. TimeMatrix: Analyzing Temporal Social Networks Using Interactive Matrix-Based Visualizations. *International Journal on Human Computer Interaction*, 26(11):1031–1051, 2010. (Zitiert auf Seite 15)

[YFDH01] K.-P. Yee, D. Fisher, R. Dhamija, M. Hearst. Animated exploration of dynamic graphs with radial layout. In *Presented at IEEE Symposium on Information Visualization*. 2001. (Zitiert auf Seite 65)

Alle URLs wurden zuletzt am 19. 01. 2014 geprüft.

Erklärung

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

Ort, Datum, Unterschrift