Fakultät für Mathematik und Informatik

Lehrgebiet Theoretische Informatik

Masterthesis zur Erlangung des akademischen Grades Master of Science

über das Thema

Vergleich von Algorithmen zur Platzierung disjunkter Boxen

Vorgelegt von: Frank Pierstorf
Witthof 8, 22305 Hamburg, E-Mail: frank.pierstorf@mail.de
Matrikelnummer: 8616779, Masterstudiengang Praktische Informatik

Erstgutachter: Prof. Dr. André Schulz
Zweitgutachter: Dr. Philipp Kindermann
Abgabedatum: 02.05.2017

Hamburg, 30.04.2017

Zusammenfassung

Beim Layouten von Graphen, deren Knoten Grafiken oder Text enthalten, kann es wichtig sein, dass die Knoten bzw. deren Label-Boxen sich nicht überlappen. Neben Graphenlayout-Algorithmen, die dieses Kriterium bereits beim Layout-Schritt selbst berücksichtigen, gibt es Algorithmen, die diese Aufgabe erfüllen, nachdem ein Initial-Layout berechnet wurde. Hierzu zählen PRISM und GTree, die beide in der Praxis zum Einsatz kommen. Gemeinsame Zielsetzung beider Algorithmen ist es, dass die Struktur sowie die Form des Initial-Layouts bei der Berechnung eines neuen überlappungsfreien Layouts möglichst erhalten bleiben.

Ziel der vorliegenden Arbeit war es, diese beiden Algorithmen zu vergleichen, um eine Empfehlung geben zu können, in welchen Fällen PRISM bzw. GTree verwendet werden sollte. Hierzu wurde das Programm OverlapR entwickelt, das beide Algorithmen implementiert und deren Berechnungen visualiert. Es werden fünf Messwerte vorgestellt, die herangezogen werden können, um die Gleichheit bzw. Unterschiedlichkeit von zwei Graphenlayouts zu bewerten. Ein zentrales Ergebnis des Vergleiches ist, dass PRISM bessere Werte erzielt, wenn GTree und PRISM ungefähr die gleiche Platzausnutzung haben.

Inhaltsverzeichnis

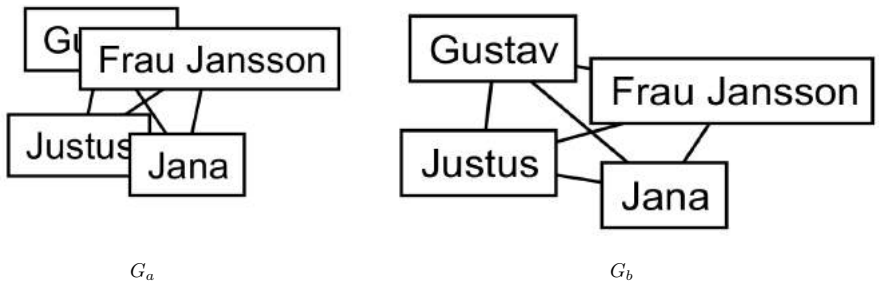
Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

1 Einleitung

Innerhalb der Graphentheorie ist das Graphenzeichnen eine eigene Disziplin. Sie das Platzieren von Knoten und Kanten im Raum oder in der Ebene zum Gegenstand. Durch das Graphenzeichnen entstehen Graphenlayouts, die im besten Fall verschiedenen ästhetischen Kriterien entsprechen. Solche Kriterien können beispielsweise sein, dass das Kreuzen von Kanten minimiert wird, dass durch Kanten verbundene Knoten auch nah aneinander platziert werden oder dass der benutzte Platz des Layouts möglichst klein ist [4]. Ein weiteres wichtiges Kriterium kann sein, dass sich die Label-Boxen (also die Beschriftung der Knoten) nicht überlappen sollten, damit alle Informationen lesbar sind [4]. Abbildung 1.1 verdeutlicht dies an einem Beispiel:



G_a G_b

Abbildung 1.1: Graph G_a zeigt ein Layout eines Graphen. G_b zeigt den gleichen Graphen mit einem überlappungsfreien Layout.

Es gibt in der Graphentheorie diverse Algorithmen, die dafür genutzt werden, eine Platzierung disjunkter Label-Boxen zu erreichen [22, 11]. In einigen Graphenlayout-Algorithmen wird die Platzierung überlappungsfreier Knoten mit berücksichtigt. Es gibt jedoch auch Algorithmen, die ein bereits berechnetes Layout (im Folgenden Text als Initial-Layout bezeichnet) auf Überlappungen untersuchen und diese dann entfernen [22]. Diese Algorithmen werden also in einem Nachbearbeitungsschritt an einen Layout-Algorithmus angewendet.
Die vorliegende Masterthesis mit dem Titel „Vergleich von Algorithmen zur Platzierung disjunkter Boxen" hat den Vergleich zweier solcher Algorithmen zum Gegenstand, die aus einem Graphenlayout, in dem sich Label-Boxen überlappen, ein neues Layout berechnen, das überlappungsfrei ist. Betrachtet werden hierbei die Algorithmen PRISM [11] und GTree [22]. Während PRISM im Jahr 2010 vorgestellt wurde und als Standard-Algorithmus zum Entfernen von überlappenden Knoten in der Software Graphviz[1] imple-

[1] Graphviz ist ein Programm zum Visualisieren von Graphen [13]. Es ist frei verfügbar und kann unter http://www.graphviz.org/ heruntergeladen werden. Alle in dieser Thesis genannten Links wurden zuletzt geprüft am 27.04.2017.

mentiert ist, wurde GTree 2016 entwickelt und wird in der Software MASGL[2] eingesetzt. Ein Vergleich dieser beiden Algorithmen scheint zum einem deshalb interessant, weil beide die Berechnung eines überlappungsfreien Layouts an der Delaunay-Triangulierung des Initial-Layouts durchführen und damit eine auffällige Gemeinsamkeit haben. Zum anderen wurde in der Arbeit zu GTree [22] ein Vergleich mit PRISM durchgeführt, jedoch können durch diesen nur bedingt Aussagen zur Qualität (womit die Gleichheit bzw. Unterschiedlichkeit zwischen Initial-Layout und überlappungsfreiem Layout gemeint ist) der beiden Algorithmen gezogen werden. Es ist also nicht möglich, anhand dieses Vergleichs eine Entscheidung treffen zu können, in welchen Fällen PRISM und in welchen Fällen GTree genutzt werden sollte. Darüber hinaus ist es nicht möglich zu sagen, ob der ein oder andere Algorithmus allgemein besser ist. Lediglich auf die Quantität und auf den Platzverbrauch der beiden Algorithmen lassen sich eindeutige Ausagen treffen: GTree ist signifikant schneller bei der Berechnung der Platzierung disjunkter Boxen als PRISM. Dabei hat GTree aber auch einen höheren Platzbedarf, d.h. dass die überlappungsfreien Layouts, die durch GTree berechnet wurden, mehr Fläche einnehmen als die, die durch PRISM berechnet wurden [22].
Ziel dieser Arbeit soll sein, durch einen Vergleich der Algorithmen PRISM und GTree, Empfehlungen zu formulieren, wann ersterer bzw. letzterer zum Einsatz kommen sollte, um ein qualitativ gutes Ergebnis zu erzielen. Hiermit ist ein überlappungsfreies Layout, das dem Initial-Layout stark ähnelt, gemeint.
Dafür müssen zunächst Messwerte entwickelt werden, anhand derer die Gleicheit bzw. Unterschiedlichkeit von Graphenlayouts abgelesen werden kann. In der vorliegenden Thesis werden fünf Messwerte definiert mittels derer der Vergleich durchgeführt wird. So werden die Veränderung der Abstände der Knoten und die Verschiebung von Knoten untersucht. Zudem wird die Veränderung der Nachbarknoten im Bezug einzelner Knoten gemessen und ausgewertet. Als weitere Kriterien innerhalb des Vergleichs von PRISM und GTree werden der genutzte Platz sowie das Seitenverhältnis eines Layouts betrachtet.
Um den Vergleich durchführen und von PRISM und GTree erzeugte überlappungsfreie Layouts betrachten zu können, wird das Programm OverlapR implementiert. In OverlapR werden die Algorithmen PRISM und GTree implementiert und Funktionen zum Laden, Erzeugen und Betrachten von Layouts bereit gestellt. Zudem lassen sich mit OverlapR die oben bereits kurz erwähnten Messwerte berechnen und darstellen.

Die vorliegende Masterthesis gliedert sich wie folgt: In Abschnitt 2 werden die beiden Algorithmen PRISM und GTree detailliert vorgestellt. Im darauffolgenden Abschnitt 3 werden die Messwerte zum Vergleichen von Graphenlayouts beschrieben. Abschnitt 4 hat die Implementierung von OverlapR zum Thema. Hierbei werden grundlegende Aspekte der Implementierung behandelt. Auf detaillierte Klassendiagramme, Sequenzdiagramme o.ä. wird bewusst verzichtet, da der Vergleich und dessen Auswertung im Vordergrund die-

[2]MASGL (Microsoft Automatic Graph Layout) ist wie Graphviz ein Werkzeug, mit dem Graphen gelayoutet und betrachtet werden können. Es ist frei verfügbar und kann unter https://www.microsoft.com/en-us/research/project/microsoft-automatic-graph-layout/ heruntergeladen werden.

ser Thesis stehen. Der kommentierte Quellcode liegt der Thesis in elektronischer Form bei. Ebenfalls in Abschnitt 4 werden die Benutzeroberfläche und ein typisches Nutzungsszenario von OverlapR vorgestellt. Der eigentliche Vergleich wird in Abschnitt 5 beschrieben, durchgeführt und ausgewertet. Er wird an insgesamt 800 Graphen aus vier Testgruppen durchgeführt. Die Masterthesis schließt in Abschnitt 6 mit Empfehlungen für die Verwendung von PRISM bzw. GTree. Darüber hinaus wird ein Ausblick gegeben, welche Aspekte bei zukünftigen Untersuchungen innerhalb der Themen Vergleichen von Graphenlayouts und Platzierung disjunkter Boxen berücksichtigt werden sollten. Abschließend wird auf Erweiterungs- und Optimierungsmöglichkeiten von OverlapR eingegangen.

2 Algorithmen zur Platzierung disjunkter Boxen

Im Verlauf der Arbeit werden folgende Notationen verwendet. Das Layout eines Graphen $G = (V, E)$ mit V = Menge der Knoten und E = Menge der Kanten, werden wir als x bezeichnen, wobei x^0 das Initial-Layout darstellt. Die durch PRISM bzw. GTree berechneten überlappungsfreien Layouts bezeichnen wir mit x^P bzw. x^G. Mit x_i werden die Koordinaten des Kotens $v_i \in V$ bezeichnet. Da nur Graphenlayouts in der Euklidischen Ebene betrachtet werden, ist $x_i \in \mathbb{R}^2$. Mit der Notation $\|\|$ ist die Euklidische Norm gemeint ist, die den Abstand von Punkten in der Ebene $\mathbb{R}^2$ beschreibt. Des Weiteren ist mit G_P die Delaunay-Triangulierung eines Graphen gemeint und mit E_P die Kanten von G_P.

2.1 PRISM

Der Algorithmus PRISM (PRoxImity Stress Model) [11] verwendet als Eingabe ein Layout eines Graphen $G = (V, E)$ mit einer Platzierung eines Knoten v_i in der Euklidischen Ebene $x_i \in \mathbb{R}^2$ sowie die Dimensionen der Label-Boxen des Knotens.
PRISM nutzt das Konzept eines Lage-Graphen (*proximity graph*): Der Lage-Graph wird aus der Lagebeziehung der Knoten V abgeleitet. Jeder Knoten v_i ist mit seinen nächsten Nachbarn mit einer Kante verbunden. PRISM berechnet hierfür zunächst die Delaunay-Triangulierung (DT) als einen solchen Lage-Graphen.
Die DT ist eine Triangulierung der Knoten v_i in der Ebene. Sie weist insbesondere das sogenannte Kreis Kriterium [18] auf:

Lemma. *Sei $V = \{v_1, ..., v_n\}$ eine Menge von Punkten. Das Dreieck $\Delta v_i, v_j, v_k$ ist ein Delaunay Dreieck der Triangulierung $DT(V)$, dann und nur dann, wenn sein Umkreis keine anderen Punkte aus V in seinem Inneren hat.*

Dies sorgt dafür, dass die Dreiecke größtmögliche Innenwinkel haben [18]. Das Dreiecksnetz hat zudem die Eigenschaft, dass es starr ist [11]. Das bedeutet, dass es nicht verformbar bzw. nicht flexibel ist, wenn Kräfte darauf einwirken [2]. Wir stellen uns dabei die Knoten als Gelenke vor und die Kanten als feste Stangen. Abbildung 2.1 soll dies verdeutlichen.

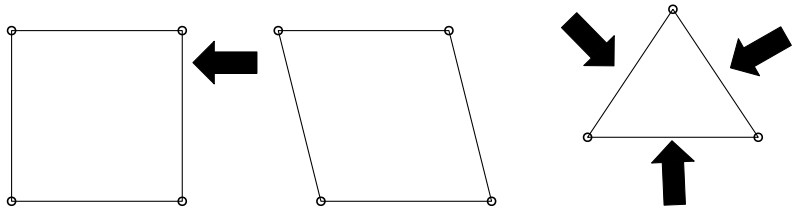
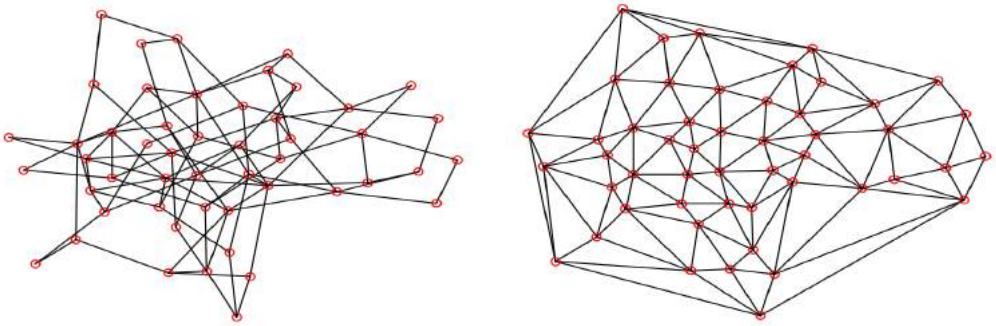


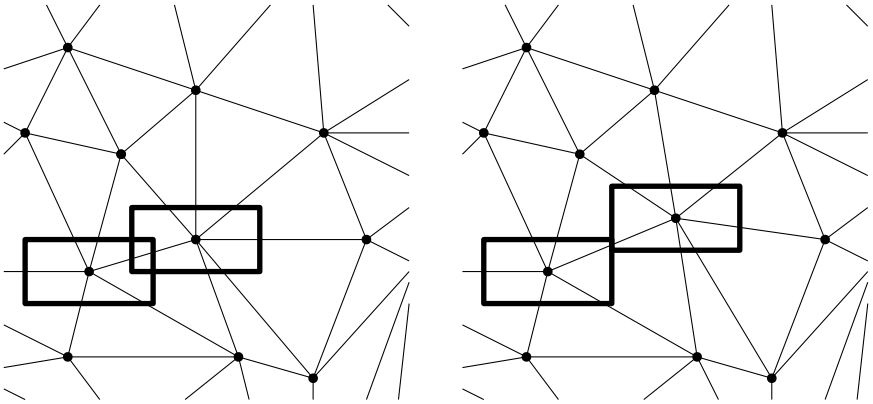
Abbildung 2.1: Der Quader ist eine flexible Struktur: Wirkt eine Kraft (schwarzer Pfeil) auf (a) so verändert sich die Form in (b). Das Dreieck ist dagegen starr: Egal, aus welcher Richtung eine Kraft wirkt, die Lagebeziehung der Ecken verändert sich nicht.

Die Delaunay-Triangulierung liefert somit ein gutes Gerüst [11], um die globale Struktur des Layouts zu erhalten: in diesem können sich die Knoten zwar bewegen, ihre relative Position zu den Nachbarknoten wird jedoch aufrecht erhalten, da in PRISM alle Verschiebungen von Knoten nur an den Kanten der Triangulierung durchgeführt werden (Näheres dazu s.u.). Alle Verschiebungen von einzelnen Knoten stehen somit lediglich im direkten Bezug zu ihren Nachbarn. Abbildung 2.2 zeigt die DT eines Graphen. Abbildung 2.3 soll illustrieren, dass Knoten sich nur in ihrer direkten Nachbarschaft bewegen.



G_a G_b

Abbildung 2.2: G_b zeigt die Delaunay-Triangulierung von G_a



(a) (b)

Abbildung 2.3: (a) zeigt einen Ausschnitt des zuvor gezeigten Graphen G_b aus Abbildung 2.2 mit zwei sich überlappenden Boxen. Diese Überlappung wird aufgelöst, indem die DT-Kante zwischen ihnen verlängert wird. Die Lagebeziehung zwischen den Knoten ändert sich dabei nicht.

Um ein neues Layout zu berechnen, das keine überlappenden Knoten mehr enthält, wird ein Spannungsmodell (stress model) zu Grunde gelegt. Dieses Modell nimmt an, dass zwischen den Knoten Federn gespannt sind, wobei diese in einem ausgeglichenem Kräftverhältnis mit einer idealen theoretischen Länge verharren [17]. Die Energie in diesem

Modell ist

$$\sum_{i \neq j} w_{ij} \left(\parallel x_i - x_j \parallel - d_{ij} \right)^2 \tag{2.1}$$

wobei d_{ij} der theoretische ideale Abstand zwischen den Knoten v_i und v_j und w_{ij} ein Gewichtungsfaktor ist, in der Regel $1/d_{ij}^2$. Andere Gewichtungen sind jedoch auch gebräuchlich [15]. Das Layout, das diese Funktion minimiert, wird dann als optimales Layout des Graphen angesehen. Durch die sogenannte Stress Majorization Technik [12], bei der eine Annäherung der Funktion durch quadratische Funktionen gebildet wird und die bei PRISM verwendet wird, kann (2.1) minimiert werden. Die dabei erzeugten quadratischen Funktionen können dann durch lineare Gleichungssysteme gelöst werden (s.u.).
Um nun den idealen Abstand zwischen zwei Knoten zu ermitteln, der schließlich so groß sein sollte, dass sich Knoten nicht (mehr) überlappen, berechnen die Autoren von PRISM den sogenannten Überlappungsfaktor t_{ij} an jeder Kante des triangulierten Graphen G_P:

$$t_{ij} = \max \left(\min \left(\frac{w_i + w_j}{\mid x_i^0 (1) - x_j^0 (1) \mid}, \frac{h_i + h_j}{\mid x_i^0 (2) - x_j^0 (2) \mid} \right), 1 \right) \tag{2.2}$$

mit

- w_i = Hälfte der Breite des Knotens v_i
- h_i = Hälfte der Höhe des Knotens v_i
- $x_i^0 (1)$ = x-Koordinate des Knotens v_i
- $x_i^0 (2)$ = y-koordniate des Knotens v_i

Für Knoten, die sich nicht überlappen ist $t_{ij} = 1$. Sollten sich nun zwei Knoten überlappen, kann diese Überlappung entfernt werden, indem die Kante um den errechneten Wert t_{ij} gestreckt wird. Das Layout soll also so entzerrt werden, dass die Überlappungen entfernt werden, die Längen der einzelnen Kanten jedoch dabei nahe dem Wert $t_{ij} \parallel x_i^0 - x_j^0 \parallel$ sind. Folgende Funktion soll schließlich, wie oben bereits eingeleitet, minimiert werden:

$$\sum_{(i,j) \in E_P} w_{ij} \left(\left|\left| x_i - x_j \right|\right| - d_{ij} \right)^2 \tag{2.3}$$

mit

- E_P = Kanten der Triangulierung G_P
- $d_{ij} = s_{ij} \parallel x_i^0 - x_j^0 \parallel$, mit s_{ij} als Skalierungsfaktor (s.u.)
- w_{ij} = Gewichtungsfaktor, in der Regel $\frac{1}{{d_{ij}}^2}$

(2.3) kann, wie bereits angedeutet, durch die Stress Majorization minimiert werden. Dabei handelt es sich um einen iterativen Prozess, der in jedem Schritt lineare Gleichungssysteme löst, bis eine bestimmte Toleranzgrenze erreicht ist. Dabei sei (2.3) mit stress(x)

bezeichnet. Nun wird zu einem Layout $X(t)$ (X ist eine $n \times 2$ -Matrix, wobei n Anzahl der Knoten ist und 2 die Anzahl der Dimensionen, t ist ein Schritt im Algorithmus) ein neues Layout $X(t+1)$ gesucht, so dass stress$(X(t+1)) <$ stress$(X(t))$. Mit jedem Schritt t wird sich folglich dem Minimum von (2.3) angenähert.
Um dies zu erreichen, wird die folgende lineare Gleichung gelöst:[3]

$$L^w X(t+1)^{(a)} = L^{X(t)} X(t)^{(a)},\ a = 1, 2 \tag{2.4}$$

mit

- $a =$ Spalte der Matrix X
- $L^w_{i,j} = \begin{cases} -w_{ij} & i \neq j \\ \sum_{k \neq i} w_{ik} & i = j \end{cases}$
- $L^Z_{i,j} = \begin{cases} -(w_{ij} d_{ij}) \text{inv}(\|Z_i - Z_j\|) & i \neq j \\ -\sum_{j \neq i} L^Z_{i,j} & i = j \end{cases}$
  wobei $\text{inv}(x) = \frac{1}{x}$, wenn $x = 0$, sonst 0.

(2.4) lässt sich näherungsweise mit dem iterativen Verfahren der konjugierten Gradienten (CG-Verfahren) lösen [11, 12, 16].
Weil für beide Spalten $a = 1, 2$ der Matrix $X(t)$ die Gleichung gelöst wird, wird ein neues Layout $X(t+1)$ berechnet. Die Stress Majorization löst (2.4) iterativ für jeden Schritt t bis ein Toleranzwert ϵ erreicht ist:

$$\frac{\text{stress}(X(t)) - \text{stress}(X(t+1))}{\text{stress}(X(t))} < \epsilon.$$

Hierbei ist ϵ überlicherweise 10^{-4} [12]. Für die Implementierung ist es wichtig zu beachten, dass die Matrix L^w nur einmal berechnet werden muss, während die Berechnung von $L^{X(t)}$ in jedem Schritt neu durchgeführt werden muss.
Nachdem (2.3) minimiert wurde, können immer noch Überlappungen auftauchen, weswegen PRISM iterativ arbeitet: in jeder Iteration wird die Delaunay-Triangulierung des Graphen berechnet, die Überlappungsfaktoren werden ermittelt und (2.3) durch die Stress Majorization - Technik gelöst. Der Algorithmus terminiert, wenn keine Überlappungen an den Kanten der Triangulierung mehr vorhanden sind.
Wie oben bereits ersichtlich, wird die ideale Distanz d_{ij} in den einzelnen Iterationen durch einen Skalierungsfaktor s_{ij} „gedämpft":

$$s_{ij} = \min(t_{ij}, s_{max}) \tag{2.5}$$

Dies stellt sicher, dass Überlappungen nicht auf einmal entfernt werden, sondern eher in kleinen Schritten: Sollte es zum Beispiel nur einen Knoten im Layout geben, der eine sehr

[3] Um den Rahmen der Arbeit nicht zu sprengen, wird auf ein genaues Herleiten sowie auf einen Beweis verzichtet. Diese finden sich ausführlich beschrieben in [12, 15].

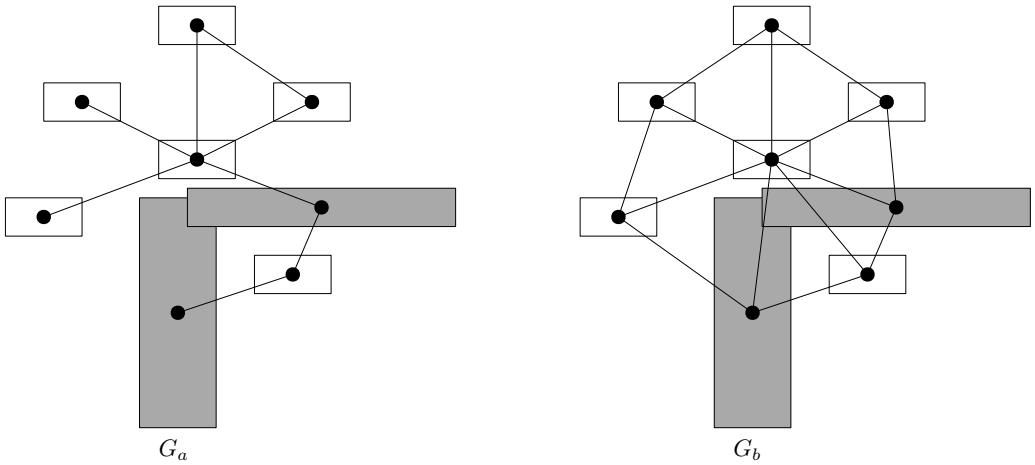


Abbildung 2.4: In Graph G_a überlappen sich die grauen Knoten. Graph G_b zeigt, dass die Delaunay-Triangulierung die beiden Knoten nicht verbindet. Deshalb werden im zweiten Teil von PRISM solche Knoten mit einem ScanLine-Algorithmus gefunden und der Graph mit einer entsprechenden Kante erweitert.

lange Label-Box hat, wird durch den Faktor garantiert, dass dieser nicht auf einmal außerhalb der anderen Knoten liegt. Die Autoren von PRISM schlagen einen Skalierungsfaktor von $s_{max} = 1.5$ vor [11]. Auch wir verwenden diesen Faktor bei den Vergleichen.
Es kann nun allerdings vorkommen, dass sich immer noch Knoten überlappen. Diese Situation kann auftreten, wenn sehr lange Knoten-Boxen im Layout vorhanden sind: Knoten, die über eine Kante in der Triangulierung verbunden sind, können sich nicht mehr überschneiden; Knoten, die allerdings nicht über solche Kanten verbunden sind, können sich sehr wohl noch überlappen. Um auch diese Knoten disjunkt zu platzieren, wird ein Scanline-Algorithmus [8] verwendet, der sich überschneidende Knoten erkennt. Die Delaunay-Triangulierung wird dann um jene Kanten erweitert, an denen sich Knoten überlappen. Die Situation wird in Abbildung 2.4 dargestellt. Algorithmus 1 stellt den gesamten Ablauf von PRISM dar.

Algorithm 1 PRISM

Eingabe: Koordinaten jedes Knotens x_i^0, sowie die Breite und Höhe der Label-Boxen $\{w_i, h_i\}, i = 1, 2, ..., |V|$.
Ausgabe: Überlappungsfreies Layout.
repeat
    Berechne Delaunay-Triangulierung G_P von x^0.
    Berechne die Überlappungsfaktoren (2.2) an allen Kanten von G_P.
    Löse das Spannungsmodell (2.3) und berechne somit neues Layout x.
    Setze $x^0 = x$.
until (keine Überlappungen mehr an den Kanten in G_P vorhanden)
repeat
    Berechne Delaunay-Triangulierung G_P von x^0.
    Finde alle Überlappungen mit einem ScanLine-Algorithmus.
    Füge Kanten zwischen Knoten, die sich überlappen, zu G_P hinzu.
    Berechne die Überlappungsfaktoren (2.2) an allen Kanten von G_P.
    Löse das Spannungsmodell (2.3) und berechne somit neues Layout x.
    Setze $x^0 = x$.
until (keine Überlappungen mehr vorhanden)

Abschließend soll noch kurz auf die Laufzeiten von PRISM eingegangen werden. Hierbei sei $n = |V|$. Die Delaunay-Triangulierung kann in $O(n \log n)$ berechnet werden [20, 18]. Das CG-Verfahren, mit dem die Gleichungssysteme der Stress Majorization gelöst werden, hat eine Laufzeit von $O(n)$ in einer Iteration [11]. Der Scanline-Algorithmus im zweiten Schritt von PRISM hat eine Laufzeit von $O(n \log n)$ [8, 11]. Insgesamt ergibt sich damit eine Laufzeit von
$O(t\,(m\,k\,n + n \log n))$, wobei t die Anzahl der Iterationen in PRISM ist, m die Anzahl der Iterationen der Stress Majorization und k die Anzahl der Iterationen des CG-Verfahrens [11]. PRISM hat damit eine asymptotische Laufzeit von $O(n \log n)$.

2.2 GTree

Bei dem zweiten Algorithmus handelt es sich um GTree (Growing Tree). Dieser wurde 2016 von Nachmanson et. al. [22] vorgestellt. Die Hauptidee hierbei besteht darin, aus dem Initial-Layout einen minimal aufspannenden Baum zu berechnen und diesen „wachsen" zu lassen, so dass Überlappungen von Knoten-Labeln entfernt werden. Wie auch bei PRISM ist das Erzeugen eines überlappungsfreien Layouts ein Schritt, der erst nach der Berechnung eines Layouts eines Graphen ausgeführt wird. Eine weitere Gemeinsamkeit ist, dass auch bei GTree zunächst die Delaunay-Triangulierung des Graphen als „Gerüst" berechnet wird.
Die Eingabe bei GTree ist eine Menge von Knoten V, in der jeder Knoten $v_i \in V$ durch ein Rechteck B_i und einen Mittelpunkt in diesem Rechteck x_i repräsentiert wird. Es wird vorrausgesetzt, dass sich für jede $v_i, v_j \in V$ auch ihre zugehörigen Punkte x_i, x_j unterscheiden. Im Folgenden wird die Delaunay-Triangulierung analog zu Abschnitt 2.1 als G_P bezeichnet und E_P als die Menge der Kanten in G_P.
Um einen minimal aufspannenden Baum zu berechnen, wird eine Kanten-Gewichtung

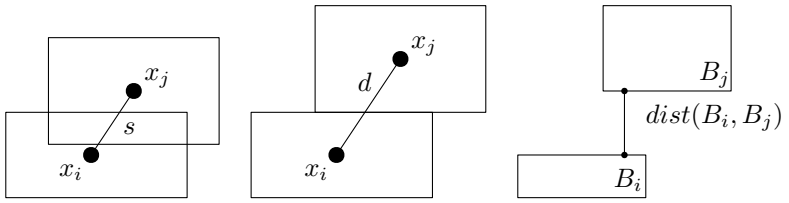
benötigt [24]. GTree verwendet hier eine Kostenfunktion c, welche die Kanten folgendermaßen gewichtet: Je größer die Überlappung, die wie der Überlappungsfaktor t_{ij} bei PRISM berechnet wird, desto niedriger die Gewichtung der Kanten. Überlappen sich Knoten nicht, dann werden sie mit dem Abstand ihrer Boxen gewichtet.
Sei $(v_i, v_j) \in E_P$. Wenn die Boxen B_i und B_j sich nicht überlappen ist die Gewichtung der Kante

$$c_{ij} = dist(B_i, B_j). \tag{2.6}$$

Dies ist der Abstand zwischen dem Rand der Box B_i und dem Rand der Box B_j (s. Abbildung 2.5). Wenn sich die Boxen allerdings überlappen, berechnet sich $c(v_i, v_j)$ folgendermaßen: wir berechnen den Wert t_{ij} analog zu PRISM, also den Wert, um den die Kante zwischen v_i und v_j verlängert werden muss, damit sich die Label-Boxen nicht mehr überschneiden. Die Gewichtung ist dann

$$c_{ij} = s - d \tag{2.7}$$

mit $s = \|x_j - x_i\|$ und $d = t_{ij}s$. Abbildung 2.5 illustriert zusammenfassend die Kostenfunktion c.



(a): $c_{ij} = s - d$ (b): $c_{ij} = dist(B_i, B_j)$

Abbildung 2.5: Kostenfunktion c_{ij} für die Kanten der Delaunay-Triangulierung. (a) zeigt den Fall, dass sich die Label-Boxen überlappen. (b) dagegen zeigt die Berechnung von c_{ij}, wenn sich die Boxen nicht überlappen.

Wenn die Gewichtung für jede Kante $(v_i, v_j) \in E_P$ berechnet wurde, kann nun ein minimal aufspannender Baum berechnet werden. Hierbei verwendet GTree Prim's Algorithmus. Der resultierende Baum T mit der Knoten-Menge V minimiert die Summe der einzelnen Gewichtungen $\sum_{(i,j)\in E'} c_{ij}$, wobei E' die Kantenmenge des Baumes ist [24, 22].
T ist nun der Input einer Funktion $GrowingT$, in der T schrittweise wächst: Zunächst wird für jedes Kind der Wurzel, das sich mit dieser überlappt, die Kante zwischen Wurzel und Kind so verlängert, dass keine Überlappung mehr stattfindet. Dabei bleibt die Wurzel an ihrer Position. Lediglich die Kinder sowie deren Unterbäume (das sind wiederum deren Kinder und Kindeskinder) werden entsprechend verschoben. Überlappen sich Kind und Wurzel nicht, so bleibt die Position des Kindes unverändert. Abbildung 2.6 visualisiert die Funktion $GrowingT$.

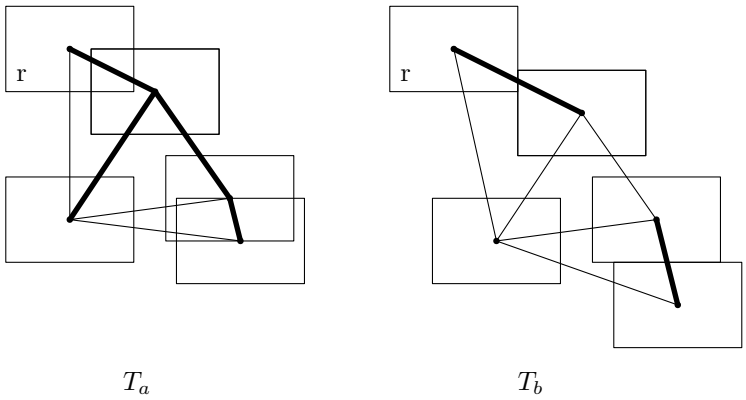


Abbildung 2.6: Funktionsweise *GrowingT*: Die fett gezeichneten Kanten in T_a zeigen einen minimal aufspannenden Baum in der Delaunay-Triangulierung des Graphen. Die Wurzel ist Knoten r. T_b zeigt den Graphen, nachdem die Funktion *GrowingT* einmal ausgeführt wurde. Dabei sind hier die gestreckten Kanten fett dargestellt.

Die Positionen der Kinder der Unterbäume werden nicht sofort geändert - lediglich die Position des Kindes wird aktualisiert. Dadurch, dass die initialen Positionen eines Elternknotens und eines Kindknotens, sowie die neue Position des Elternknotens genutzt werden, erhält der Algorithmus die neue Position des Kindes. Das heißt, dass der „alte" Abstand der Knotenpositionen x_i^0 und x_j^0 bei der Berechnung der neuen Position x_i verwendet wird, siehe Zeile 6 in Algorithmus 2. Dies sichert, dass bei einer Verschiebung eines Knotens auch sein Unterbaum mit verschoben wird.

Algorithm 2 GrowingT

```
1: Eingabe: Knoten-Positionen x^0 und Wurzel x_r
2: Ausgabe: Neue Knoten-Positionen (neues Layout) x
3: x_r = x_r^0
4: function GrowAtNode(i)
5:     for all j ∈ Kinder(i) do
6:         x_j = x_i + t_ij(‖x_j^0 − x_i^0‖)
7:         GrowAtNode(j)
8:     end for
9: end function
```

Die Wahl der Wurzel ist dabei unerheblich. Unterschiedliche Wurzeln liefern dasselbe Layout - lediglich eine Translation des gesamten Layouts in der Ebene kann vorkommen [22].
Analog zu PRISM (2.5) kann der Wert t_{ij} im Algorithmus *GrowingT* mit einem Skalierungsfaktor sij „gedämpft" werden. Die Autoren von GTree verwenden in ihren Vergleichen keinen Skalierungsfaktor und einen von $s_{ij} = 1.5$. Letzterer erzeugt Layouts mit einem geringeren Platzverbrauch, GTree hat dann jedoch eine etwas längere Laufzeit [22].
Nach dem Durchführen von Algorithmus 2 wird wiederum die Delaunay-Triangulierung G'_P berechnet. Sollten sich nun noch Überlappungen an den Kanten $(v_i, v_j) \in E'_P$ der Triangulierung G'_P befinden, wird erneut die Kostenfunktion c berechnet, der minimal

aufspannende Baum T' gebildet und Algorithmus 2 auf diesen angewandt. Diese Prozedur wird so oft wiederholt, wie Überlappungen in der Triangulierung vorhanden sind.
Wie bereits in Abschnitt 2.1 zu PRISM beschrieben, kann es trotzdem noch vorkommen, dass sich Knoten-Label überlappen. Das Vorgehen ist dabei dasselbe wie bei PRISM: Mittels eines Scanline-Algorithmus [8] werden Knoten, die sich weiterhin überschneiden, ermittelt und die Delaunay-Triangulierung um die Kante zwischen ihnen erweitert. Dann wird wieder oben beschriebene Prozedur wiederholt. Algorithmus 3 zeigt den gesamten Ablauf von GTree.

Algorithm 3 GTree

```
1: Eingabe: Koordinaten jedes Knotens x_i, sowie die Label-Boxen B_i.
2: Ausgabe: Überlappungsfreies Layout.
3: repeat
4:     Berechne Delaunay-Triangulierung G_P von allen x_i.
5:     Berechne Kostenfunktion c (1.5 bzw. 1.6).
6:     Berechne minimal aufspannenden Baum T anhand von c.
7:     Berechne neue x_i mit der Funktion GrowAtNode(Wurzel r von T).
8: until (keine Überlappungen mehr an den Kanten in G_P vorhanden)
9: repeat
10:    Berechne Delaunay-Triangulierung G_P von allen x_i.
11:    Finde alle Überlappungen mit einem ScanLine-Algorithmus.
12:    Füge Kanten zwischen Knoten, die sich überlappen, zu G_P hinzu.
13:    Berechne Kostenfunktion c (1.5 bzw. 1.6).
14:    Berechne minimal aufspannenden Baum T anhand von c.
15:    Berechne neue x_i mit der Funktion GrowAtNode(Wurzel r von T).
16: until (keine Überlappungen mehr vorhanden)
```

Die Autoren von GTree geben in ihrer Arbeit an, dass es möglich ist Initial-Layouts so zu konstruieren, dass GTree neue Layouts erzeugt, in denen noch Überlappungen vorhanden sind [22]. In unseren Experimenten mit OverlapR und innerhalb unseres Vergleichs ist dieser Fall kein einziges Mal aufgetreten.
Die Laufzeit von GTree berechnet sich ähnlich zu PRISM: Die Berechnung der Delaunay-Triangulierung sowie der Scanline-Algorithmus brauchen jeweils $O(n \log n)$ Zeit [20, 18, 8, 11]. Die Berechnung der Kostenfunktion c hat eine lineare Laufzeit von $O(n)$. Der minimal aufspannende Baum kann mit Prim's Algorithmus in $O(n \log n)$ aufgebaut werden [22, 24]. Der Algorithmus *GrowingT* weist eine lineare Laufzeit auf. Analog zu PRISM ergibt sich schließlich eine Gesamtlaufzeit von $O(t(n \log n + n \log n))$. GTree hat damit eine asymptotische Laufzeit von $O(n \log n)$.

2.3 Gegenüberstellung PRISM - GTree

Stellen wir PRISM und GTree gegenüber und vergleichen deren Arbeitsweise, können wir zusammenfassen, dass GTree Überlappungungen entfernt, indem der Graph bzw. der durch ihn vorgegebene minimal aufspannende Baum, wächst. Dabei werden auch ganze Teilbäume mit verschoben, was dafür sorgt, dass Teile der Struktur (zumindest

in einer Iteration) zunächst gut erhalten bleiben. Durch das Berechnen der Delaunay-Triangulierung wird dabei dafür gesorgt, dass benachbarte Knoten im Initial-Layout auch im berechneten Layout benachbart sind. PRISM nutzt den gleichen Effekt der Triangulierung, wie oben gezeigt. Durch die Stress Majorization, die dafür sorgt, dass ideale Abstände minimiert werden, werden Überlappungen in PRISM dadurch entfernt, dass in einer Iteration alle Knoten in Bezug zu ihrer Nachbarschaft verschoben werden. Bei PRISM werden beispielsweise bei zwei Knoten, die sich überlappen, beide Positionen verändert, während bei GTree nur einer von beiden Knoten in eine bestimmte Richtung bewegt wird. Abbildung 2.7 soll dies verdeutlichen. Um zu zeigen, dass GTree den Graphen „wachsen" lässt, und dass bei PRISM alle Knoten in Bezug zueinander verschoben werden, haben wir den Ursprung eingezeichnet.

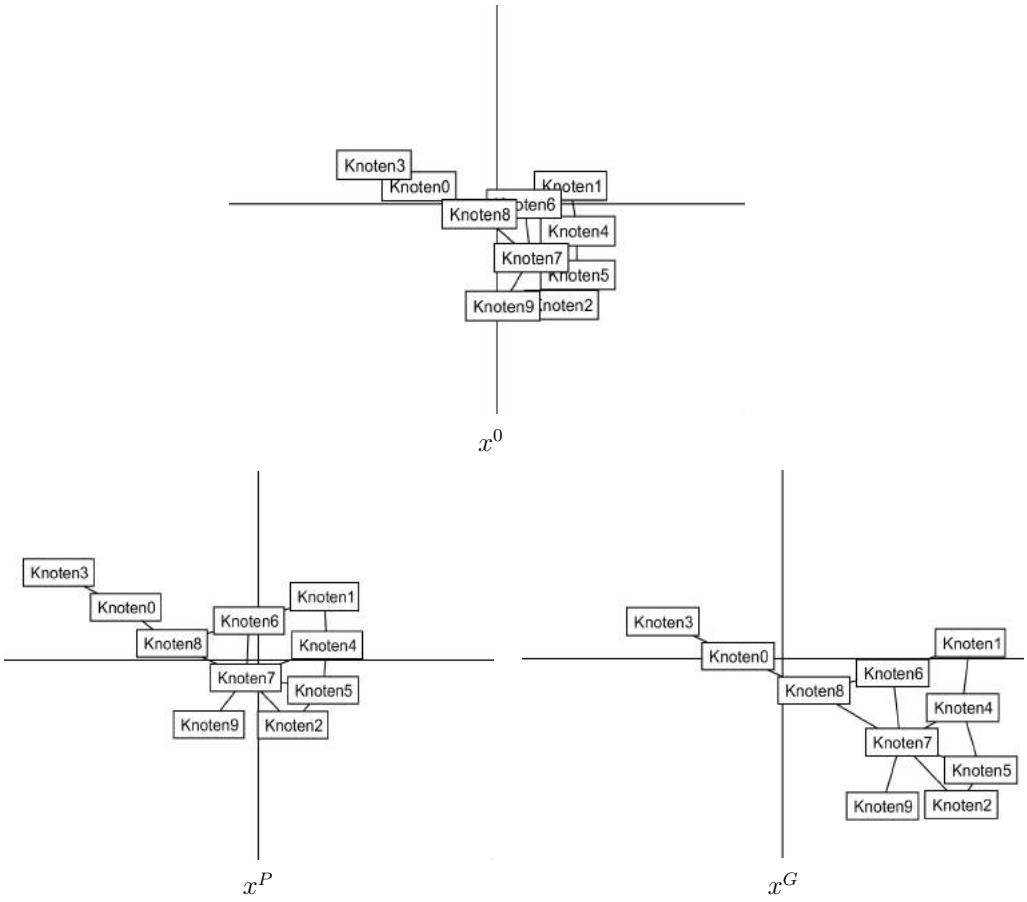


Abbildung 2.7: Arbeitsweisen PRISM - GTree

3 Vergleichen von Graphenlayouts

Das Definieren von Messwerten, die zum Vergleichen von Graphenlayouts herangezogen werden können, ist eine nicht triviale Aufgabe [22]. Während es viele Arbeiten zum Vergleichen von Graphen gibt, die die Struktur dieser untersuchen [6, 9, 26], gibt es bisher kaum Arbeiten zum Messen von Ähnlichkeiten bzw. Unterschieden zwischen Graphenlayouts [11]. In der vorliegenden Masterthesis werden in Anlehnung an [11, 22] insgesamt fünf Messwerte definiert und zum Vergleich zwischen PRISM und GTree herangezogen. Sie sind Thema dieses Abschnittes.

3.1 Distanzen zwischen Knoten

Um verschiedene Layouts miteinander zu vergleichen, bietet es sich an, die Abstände zwischen allen Knoten des Initial-Layouts x^0 zu messen und diese mit den Abständen aller Knoten in dem neuen Layout x zu vergleichen. So kann der sogenannte Frobenius-Fehler berechnet werden, der als Maß für die Gleichheit zweier Layouts verwendet werden kann [7]. Da hierbei der Abstand eines jeden Knotens zu jedem anderen Knoten im Layout berechnet werden muss, ist diese Berechnung sehr teuer.
Die Autoren von PRISM schlagen deshalb als Maß für die Gleichheit von Graphenlayouts unter anderem σ_{dist} vor [11]. Dieser Wert berechnet sich aus den Unterschieden der Kantenlängen der zu vergleichenden Graphen. Dabei werden nur die Kanten der Delaunay-Triangulierung des Initial-Layouts x^0 untersucht.
Hierbei gibt r_{ij} das Verhältnis zwischen der Kantenlänge einer Kante in der Triangulierung $\{v_i, v_j\} \in E_P$ des Initial-Layouts x^0 und dem Abstand der Knoten v_i und v_j im neuen Layout x an:

$$r_{ij} = \frac{\parallel x_i - x_j \parallel}{\parallel x_i^0 - x_j^0 \parallel}, \{v_i, v_j\} \in E_P \tag{3.1}$$

Das Maß der Ungleichheit ergibt sich dann aus der Standardabweichung

$$\sigma_{\text{dist}}\left(x^0, x\right) = \frac{\sqrt{\frac{\sum_{\{v_i,v_j\}\in E_P}(r_{ij}-\bar{r})^2}{|E_P|}}}{\bar{r}} \tag{3.2}$$

mit

$$\bar{r} = \frac{1}{\mid E_P \mid} \sum_{\{v_i,v_j\}\in E_P} r_{ij}. \tag{3.3}$$

Je kleiner dieser Wert (≥ 0), desto ähnlicher sind sich die Graphenlayouts. Der Wert 0 ist dabei das beste Ergebnis.
Das Vorgehen zum Messen der Gleichheit mit σ_{dist} als Maß ist zusammenfassend also folgendes: Zunächst berechnen wir die Delaunay-Triangulierung des Originalen Layouts x^0. Nun wird an jeder Kante $\{v_i, v_j\} \in E_P$ dieser Triangulierung der Abstand zwischen den zwei Knoten v_i und v_j, also die Länge der Kante, gemessen. Diesen Abstand setzen wir zum Abstand der Knoten v_i und v_j im neuen Layout x ins Verhältnis und berechnet mit (3.2) die Standardabeichung.

Sollen beispielsweise Layout x^P und Layout x^G mit dem Initial-Layout x^0 verglichen werden, um eine Aussage darüber zu treffen, ob Layout x^P oder Layout x^G ähnlicher zu x^0 ist, können wir die Werte $\sigma_{\text{dist}}\left(x^0, x^P\right)$ und $\sigma_{\text{dist}}\left(x^0, x^G\right)$ berechnen und vergleichen. Sollte zum Beispiel $\sigma_{\text{dist}}\left(x^0, x^P\right)$ kleiner sein, so würde das Layout x^P als besser gewertet.

3.2 Prokrustes Analyse

Als ein weiteres Maß zum Messen von Ungleichheit zwischen zwei Graphenlayouts kann die Prokrustes Statistik herangezogen werden [5, 11]. Hierbei wird die Verschiebung einzelner Knoten zwischen zwei Layouts gemessen. Ein neues Layout, das aus einem Ursprung-Layout abgeleitet wird, wird als gleich angesehen, wenn es sich durch Translation, Skalierung und Rotation auf dieses zurückführen lässt [11]. Abbildung 3.1 verdeutlicht dies.

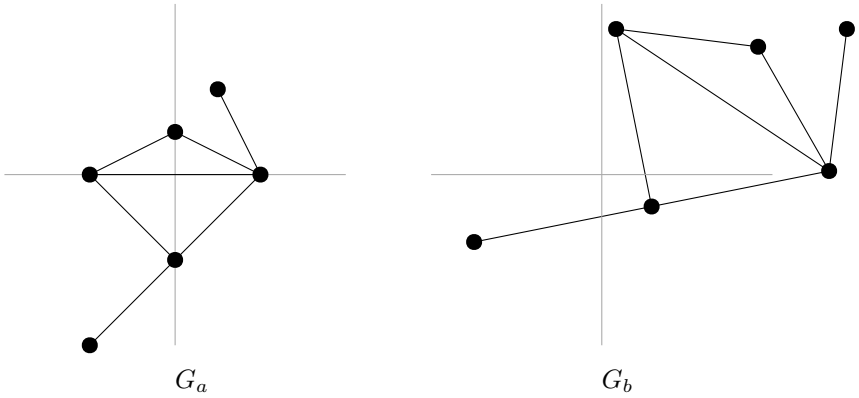


Abbildung 3.1: In der Procrustes analysis werden beide Layouts als gleich angesehen: G_b kann durch Translation, Rotation und Skalierung in Layout G_a überführt werden und umgekehrt.

Um nun den Unterschied zwischen zwei Layouts zu bestimmen, wird ein Layout in das andere näherungsweise durch Translation, Rotation und Skalierung „überführt". Am Ende dieser Schritte kann dann die Prokrustes Statistik berechnet werden [11]. In Anlehnung an Cox und Cox [5] ist das Vorgehen wie folgt. Zunächst werden die Layouts x^0 und x in die Matrizen X und Y überführt. Im Rahmen dieser Arbeit sind das $n \times d$ - Matrizen, wobei $n = |V|$ und $d = 2$ für die Anzahl der Dimensionen. Als Beispiel: Für die drei Knoten v_1, v_2, v_3 mit den Koordinaten $(1.5, 2), (3.54, -3.4), (2, 0.23)$ sieht die dazugehörige Matrix folgendermaßen aus:

$$\begin{bmatrix} 1.5 & 2 \\ 3.54 & -3.4 \\ 2 & 0.23 \end{bmatrix}.$$

Die Knoten in Matrix X werden nun in der Ebene skaliert, transliert und rotiert, sodass sie möglichst nahe an die Knoten-Positionen aus Matrix Y „herankommen". Folgende Berechnungen werden dafür durchgeführt:

1. Translation: Die Zentren beider Layouts X und Y werden auf den Ursprung (hier

also auf $(0,0)$) verschoben.

2. Rotation: Es wird eine Matrix $A = \left(X^T Y Y^T X\right)^{1/2} \left(Y^T X\right)^{-1}$ berechnet und X zu XA rotiert.

3. Skalierung: Layout X wird skaliert, indem jede Koordinate mit $p = {}^{tr(X^T Y Y^T X)}/_{tr(X^T X)}$ multipliziert wird[4].

4. Berechnen der Prokrustes Statistik:

$$R^2 = 1 - {}^{tr\left(\left(X^T Y Y^T X\right)^{1/2}\right)^2}/_{tr(X^T X) tr(Y^T Y)} \tag{3.4}$$

Der Wert R^2 (3.4), der ab jetzt mit $\sigma_{\text{disp}}\left(x^0, x\right)$ bezeichnet wird, kann zwischen 0 und 1 liegen, wobei 0 ausdrückt, dass die Layouts x^0 und x gleich sind.
Analog zu Kapitel 3.1 soll nun zusammenfassend das Vorgehen zum Vergleich der Layouts x^P und x^G anhand der Prokrustes Statistik erläutert werden. Zunächst bilden wir die benötigten Matrizen, vollziehen die Operationen 1. bis 3. und berechnen jeweils die Werte $\sigma_{\text{disp}}\left(x^0, x^P\right)$ und $\sigma_{\text{disp}}\left(x^0, x^G\right)$ und vergleichen sie. Sollte zum Beispiel $\sigma_{\text{disp}}\left(x^0, x^G\right)$ kleiner sein, so würde das Layout x^G als besser gewertet.

3.3 k-näheste Nachbarn

Bei ihren Vergleichen zwischen PRISM und GTree verwenden die Autoren von GTree neben σ_{dist} und σ_{disp} den Wert k-näheste Nachbarn [22]. Hierbei werden im Initial-Layout x^0 zu jedem Knoten v_i die k-nähesten Knoten ermittelt. Auch die k-nähesten Knoten von Knoten v_i im neuen Layout x werden entsprechend ermittelt. Danach wird der Durchschnitt m beider Mengen gebildet, um eine Verzerrung $(k-m)^2$ berechnen zu können, die angibt, wie stark sich die Menge der k-nähesten Nachbarn verändert hat. Sollten sich die k-nähesten Knoten nicht geändert haben, so ist die Verzerrung bzw. der Fehler gleich 0.
Seien also $x_1^0, ..., x_n^0$ die Koordinaten der Knoten $v_1, ..., v_n$ im Initial-Layout, k eine natürliche Zahl mit $0 < k \leq n$. Die Menge $I = \{1, ..., n\}$ enthält die Knoten-Indizes. Für jedes $i \in I$ wird die Menge $N_k(x^0, i) \subset I \setminus \{i\}$ gebildet, so dass $|N_k(x^0, i)| = k$ und dass für jedes $j \in I \setminus N_k(x^0, i)$ und jedes $j' \in N_k(x^0, i)$ folgende Bedingung gilt: $\|x_j^0 - x_i^0\| > \|x_{j'}^0 - x_i^0\|$.
Seien $x_1, ..., x_n$ die Koordinaten der Knoten $v_1, ..., v_n$ im neuen Layout x. Wir können nun die Anzahl der Elemente im Durchschnitt der jeweiligen Mengen berechnen.

$$m_i = |N_k(x^0, i) \cap N_k(x, i)| \tag{3.5}$$

und damit die Verzerrung $(k - m_i)^2$. Summieren wir diesen Wert und teilen ihn durch

[4]Mit tr(A) ist Spur(A) gemeint. Hierbei wird aus den Hauptdiagonalelementen in einer $n \times n$ - Matrix die Summe gebildet [14].

die Anzahl der Knoten erhalten wir

$$\mathrm{cn_k}(x^0, x) = \frac{\sum_i^n (k - m_i)^2}{n} \tag{3.6}$$

Je niedriger cn_k, desto geringer die Verzerrung und damit ein größeres Übereinstimmen von x^0 und x.
Das Berechnen der Menge m_i (2.5) soll an einem Beispiel mit $k = 8$ erläutert werden: Zunächst sei die Menge an Knoten-Indizes $I = \{1, 2, ..., 20\}$. Die Menge der nähesten 8 Nachbarn von Knoten v_3 im Layout x^0 sei $N_8(x^0, 3) = \{6, 11, 2, 1, 19, 18, 5, 4\}$, die Menge der nähesten 8 Nachbarn im neuen Layout x sei $N_8(x, 3) = \{7, 11, 2, 1, 20, 19, 18, 9\}$. Die Schnittmenge dieser beiden Mengen ist $\{11, 2, 1, 19, 18\}$, m_3 ist also 5 und somit ist $(k - m_3)^2 = (8 - 5)^2 = 9$.
Um nun Layout x^P und Layout x^G mit dem Initial-Layout x^0 zu vergleichen, um eine Aussage darüber zu treffen, ob Layout x^P oder Layout x^G ähnlicher zu x^0 ist, berechnen und vergleichen wir $\mathrm{cn_k}(x^0, x^P)$ und $\mathrm{cn_k}(x^0, x^G)$ für ein festes k. Sollte zum Beispiel $cn_k\left(x^0, x^P\right)$ kleiner sein, so würde das Layout x^P als besser gewertet.

3.4 Fläche

Neben den zuvor vorgestellten Maßen, die sich auf die Entfernung von Knoten bzw. auf die Verschiebung von Knoten beziehen, gilt der verbrauchte Platz des Layouts auch als Qualitätsmerkmal [11, 22]. Insbesondere wenn es um die Platzierung disjunkter Label-Boxen geht, ist dieses Merkmal wichtig: schließlich kann das Layout einfach skaliert werden bis keine Überlappungen der Boxen mehr auftauchen. Die Layouts würden sich dann nicht unterscheiden bzw. die Werte $\sigma_{\mathrm{dist}}\left(x^0, x\right)$ und $\sigma_{\mathrm{disp}}\left(x^0, x^G\right)$ wären beide 0. Der Platzbedarf des überlappungsfreien Layouts wäre dann jedoch unter Umständen groß.
Eine andere Vorgehensweise kann sein, dass die Label-Boxen in noch freien Raum verschoben werden und die Boxen somit nah beieinander sind. Dies würde wenig Platz verbrauchen. Der Unterschied zwischen dem Initinal-Layout und dem überlappungsfreien Layout wäre jedoch unter Umständen groß [11]. Abbildung 3.2 verdeutlicht beide Extreme.
Um also auch die Platzausnutzung bei dem Vergleich von PRISM und GTree zu berücksichtigen, wird der genutzte Platz berechnet. Hierbei wird die Distanz zwischen dem äußerst linken Knoten und dem äußerst rechten Knoten des Layouts x als Breite b_x berechnet und die Distanz zwischen dem obersten Knoten und dem untersten Knoten als Höhe h_x. Der genutzte Platz wird im Vergleich als

$$area\,(x) = \frac{b_x \cdot h_x}{10^6} \tag{3.7}$$

berechnet, wobei die Standardisierung mit 10^6 dazu dient, den Platzbedarf schneller lesen bzw. vergleichen zu können. In dem Vergleich zwischen PRISM und GTree wird die geringere Platzausnutzung als besser bewertet.

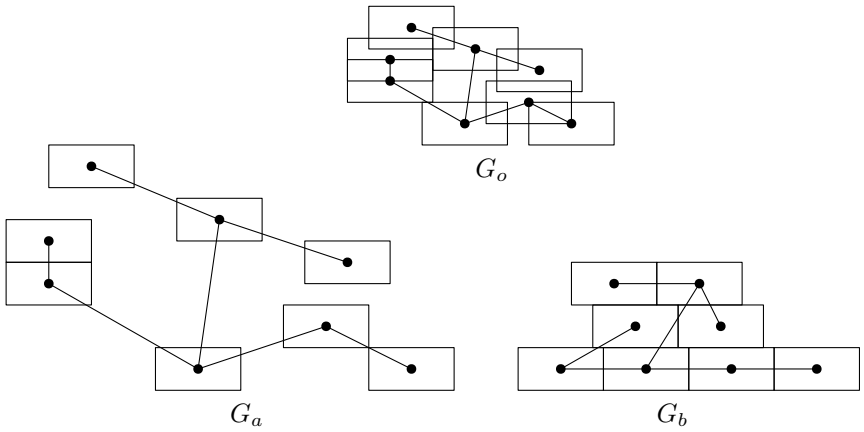


Abbildung 3.2: Zwei Extreme: G_a zeigt das Entfernen der Überlappungen in G_o durch Skalierung, bis keine Überlappungen mehr vorhanden sind. Die Platzausnutzung ist hierbei am größten. G_b hingegen zeigt das „Packen" von Knoten so eng wie möglich aneinander. Die Platzausnutzung ist hier sehr niedrig.

3.5 Seitenverhältnis

Neben der Platzausnutzung spielt unter Umständen auch das Seitenverhältnis eine Rolle. So kann es wichtig sein (etwa beim Ausdruck eines Graphen), dass das Seitenverhältnis nicht zu stark von dem des Initial-Layouts abweicht, wenn eine disjunkte Platzierung von Label-Boxen errechnet wird.
Das Seitenverhältnis wird mit

$$s(x) = b_x \cdot h_x \tag{3.8}$$

berechnet und somit zu 1 normiert. Es ergibt sich dann das Seitenverhältnis $1 : s(x)$. Um dieses Verhältnis im folgenden Vergleich zwischen PRISM und GTree zu nutzen, werden die Seitenverhältnisse des Initial-Layouts $s\left(x^0\right)$ und der neuen Layouts von PRISM $s\left(x^P\right)$ und GTree $s\left(x^G\right)$ berechnet. Im nächsten Schritt werden die Beträge der Differenzen zwischen $|s\left(x^0\right) - s\left(x^P\right)|$ und $|s\left(x^0\right) - s\left(x^G\right)|$ gebildet. Der Geringere von beiden wird dann als besser angesehen, weil hier entsprechend das Seitenverhältnis weniger vom Seitenverhältnis des Initial-Layouts abweicht.

4 Implementierung

Im folgenden Abschnitt wird das Programm OverlapR vorgestellt, mit dem wir die Vergleiche zwischen PRISM und GTree anhand verschiedener Graphen durchgeführt haben. Dabei gehen wir vor allem auf die Implementierung der Algorithmen ein und stellen darüber hinaus die Benutzeroberfläche (GUI) vor.

4.1 Überblick

Für den Vergleich der beiden Algotithmen PRISM und GTree wurde das Programm OverlapR entwickelt. Es wurde mit dem Java SE Development Kit 8[5] und der Entwicklungsumgebung Eclipse Neon[6] geschrieben.
Obwohl es auch zum Betrachten von Graphen verwendet werden kann, dient es hauptsächlich dazu, die von PRISM und GTree erzeugten überlappungsfreien Layouts zu vergleichen. Zudem wurde größerer Wert auf die Lesbarkeit des Quellcodes und auf die Genauigkeit der Algorithmen gelegt als auf die Performance. So sind die in den Arbeiten zu PRISM und GTree [22, 11] angegebenen Laufzeiten wesentlich kürzer als die Zeiten, die OverlapR benötigt.
Folgende Anforderungen soll OverlapR im Hinblick auf den Vergleich erfüllen:

- Implementierung von PRISM
- Implementierung von GTree
- Implementierung der Kriterien und Messwerte im Bezug auf das Vergleichen von Graphen (Platzausnutzung, Beibehaltung der Struktur)

Anforderungen an die Bedienbarkeit sind unter anderem:

- Einlesen von `graphML`-Dateien (s.u.)
- Darstellung von Graphen mit Knoten, Label-Boxen und Kanten
- Darstellung der Messwerte

Im Folgenden gehen wir genauer auf die Implementierung ein, wobei wir jedoch aus Platzgründen auf zu detaillierte Beschreibungen verzichten. Zunächst stellen wir für jedes Paket die Klassendiagramme dar. Dabei beschränken wir uns auf die Angabe der öffentlichen Funktionen. Danach werden wir kurz auf einige wichtige Aspekte und Funktionen der Klassen eingehen. Der interessierte Leser sei auf den Quellcode verwiesen, der der Arbeit in elektronischer Form beiliegt und kommentiert ist.

[5] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[6] https://www.eclipse.org/downloads/eclipse-packages/

Die folgende Abbildung zeigt die Pakete des Programms OverlapR mit den dazugehörigen Klassen.

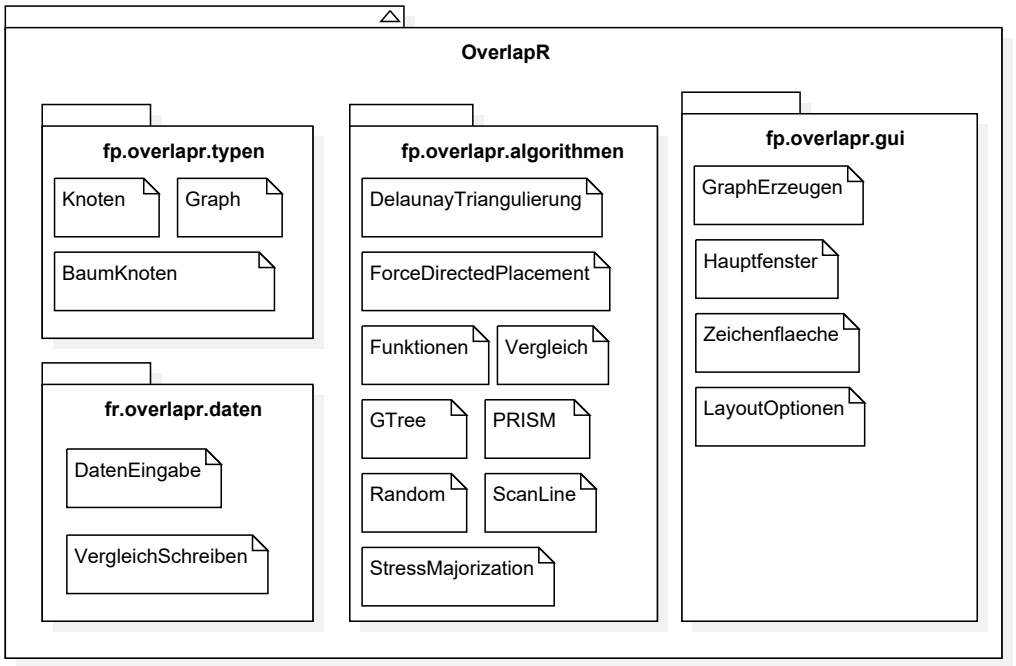


Abbildung 4.1: Paketstruktur OverlapR

4.2 Datentypen

Die Klassen der Datentypen befinden sich im Quellcode im Paket `fp.overlapr.typen`.

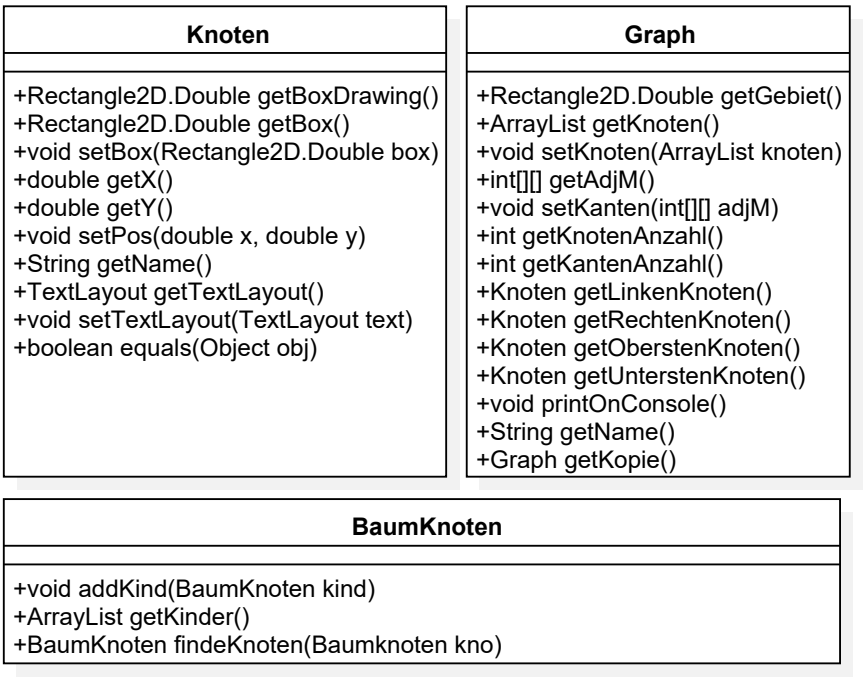


Abbildung 4.2: Klassendiagramme des Pakets `fp.overlapr.typen`

Klasse `Graph`

Die Klasse `Graph` implementiert einen Graphen. Ein Graph hat einen Namen, Knoten in Form einer `ArrayList<Knoten>`[7] sowie eine Adjazenzmatrix in Form eines zweidimensionalen Arrays, die die Kanten im Graphen angibt. Neben den üblichen Settern und Gettern bietet die Klasse folgende Methoden an:

- `Rectangle2D.Double getGebiet()`

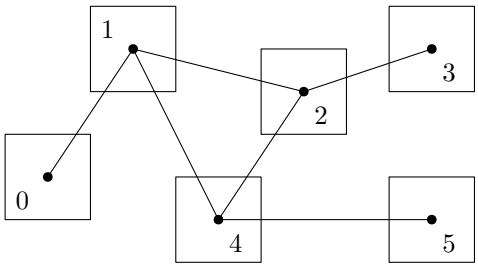
  Berechnet das rechteckige Gebiet, das der Graph einnimmt und gibt es als Rechteck[8] zurück, welches dann unter anderem im Programm gezeichnet werden kann.

- `Graph getKopie()`

  Erstellt eine vollständige Kopie des Graphen.

Um die Algorithmen in einfacher Weise berechnen zu können, sind die Knoten indirekt in der Adjazenzmatrix implementiert. Jeder Knoten, der ein Objekt der Klasse `Knoten` (s.u.)

[7] `Objekt der Java-Klasse java.util.ArrayList`
[8] Objekt der Java-Klasse `java.awt.geom.Rectangle2D`



G_a

Abbildung 4.3: Graph G_a mit Knoten-IDs

ist, bekommt beim Erstellen des Graphen eine eindeutige $ID \in \{0, 1, ..., n\}$ zugewiesen, wobei n die Anzahl der Knoten ist. Diese ID dient dazu, die Adjazenzmatrix aufzubauen. Aus dieser ist abzulesen, welche Knoten miteinander über eine Kante verbunden sind. Für den Graphen G_a aus Abbildung 4.3, bei dem die Knoten mit ihrer ID bezeichnet sind, verwenden wir folgende Matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Aus der Spalten bzw. Zeilen-Nummer kann nun die Knoten-ID abgelesen werden und so ermittelt werden, zwischen welchen Knoten Kanten vorhanden sind.

Klasse `Knoten`

In OverlapR repräsentiert ein Objekt der Klasse `Knoten` einen Knoten im Graphen. Neben seinen Koordinaten x, y in der Ebene, die vom Java-Primitiv `double` sind, hat der Knoten ein Text-Layout[9], welches den Text, Schriftgrad und die Label-Box des Knotens bestimmt. Folgende Methoden wollen wir kurz erklären:

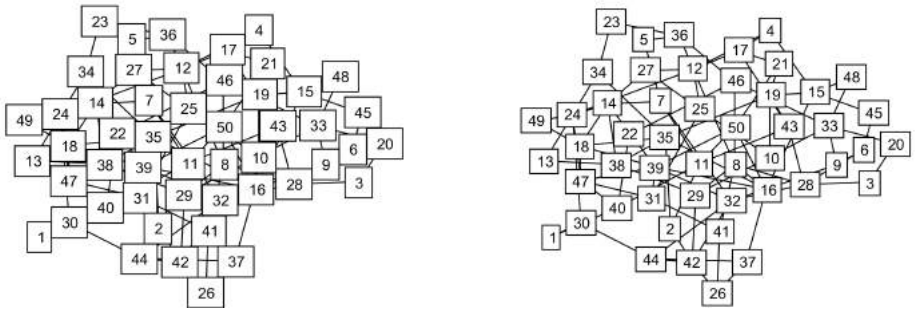
- `Rectangle2D.Double getBox()`

  Diese Methode liefert die Box zurück, anhand derer in den Algorithmen PRISM und GTree Überlappungen ermittelt und schließlich entfernt werden. Um die Lesbarkeit des gezeichneten Graphen zu erhöhen, bietet die Klasse zudem folgende Funktion an:

[9] Objekt der Java-Klasse `java.awt.font.TextLayout`

- `Rectangle2D.Double getBoxDrawing()`

  Da PRISM und GTree sich überlappende Boxen so nah wie möglich aneinander platzieren, kann das die Lesbarkeit des Layouts erschweren, weil Kanten eventuell nicht mehr sichtbar sind.
  Diese Funktion liefert ein Rechteck, das um einen bestimmten Wert (in OverlapR: `ABSTAND_BOX_UMGEBUNG` = 5) kleiner ist als die Box, mit der Überlappungen entfernt werden, sodass zwischen disjunkten Boxen ein kleiner Abstand ist. Abbildung 4.4 zeigt den Unterschied.



G_a G_b

Abbildung 4.4: G_a wurde mit der Methode getBox() gezeichnet, während G_b mit der Methode getBoxDrawing() gezeichnet wurde. Durch den Abstand der Boxen lassen sich hier die Kanten besser erkennen. Da getBoxDrawing() nur in der GUI genutzt wird, liefern beide Graphen bei unseren Berechnungen dieselben Ergebnisse.

Klasse `BaumKnoten<T>`

Da GTree einen minimal aufspannenden Baum berechnet, benötigen wir eine Klasse, die einen Baum repräsentieren kann. Dafür haben wir die generische Datenstruktur `BaumKnoten<T>` implementiert. Ein BaumKnoten enthält ein Datum vom Typ `T` und gegebenenfalls weitere `BaumKnoten` als Kinder - in Form einer `ArrayList`. Neben Funktionen, mit denen Kinder hinzugefügt werden können und mit denen wir beispielweise die Wurzel setzen können, ist vor allem folgende Methode wichtig:

- `BaumKnoten<T> findeKnoten(T data)`

  Findet einen BaumKnoten anhand des übergebenen Datums `data` und liefert diesen zurück, wenn vorhanden. Da es sich um eine rekursive Struktur handelt, wird der gesamte Baum durchsucht.

4.3 Algorithmen

Der Quellcode, der die Algorithmen (unter anderem PRISM, GTree, Stress Majorization) des Programms implementiert, befindet sich im Paket
`fp.overlapr.algorithmen`.

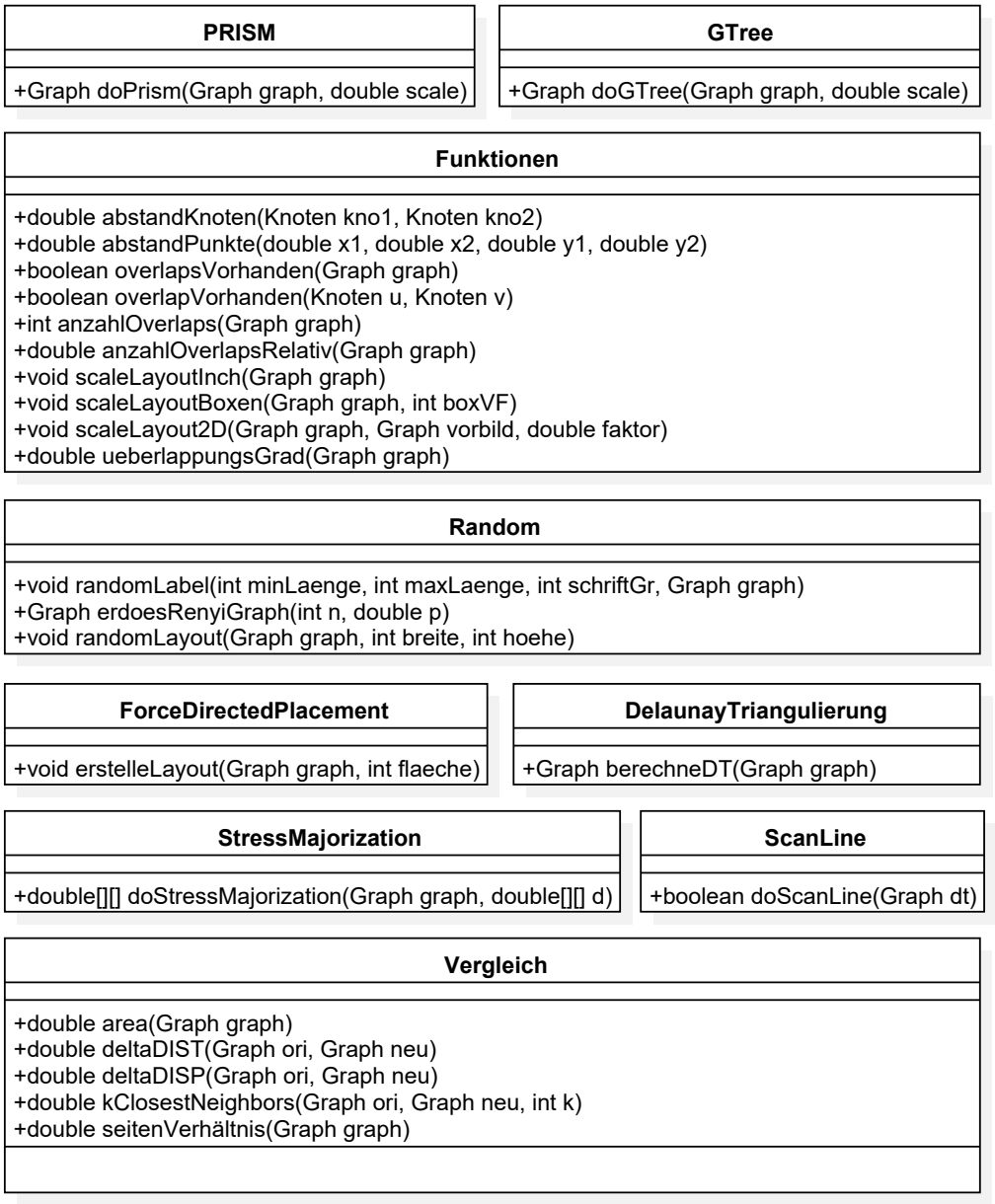


Abbildung 4.5: Klassendiagramme des Pakets `fp.overlapr.algorithmen`

Klasse `DelaunayTriangulierung`

Diese Klasse stellt eine Methode bereit, die die Delaunay-Triangulierung eines übergebenen Graphen zurückgibt. Hierfür nutzen wir die Java-Implementierung von Johannes Diemke[10]. Hauptsächlich entschieden wir uns für diese Bibliothek, weil sie eine einfache Schnittstelle anbietet und der Entwickler sie für wissenschaftliche Zwecke frei zur Verfügung stellt.

[10] http://www.informatik.uni-oldenburg.de/~trigger/page4.html

Klasse `ForceDirectedPlacement`

In dieser Klasse ist ein Graphenlayout Algorithmus implementiert. Hierfür verwenden wir ein Kräfte-basiertes Layout der Knoten, das zwei Anforderungen genügt [10]:

1. Knoten, die über eine Kante miteinander verbunden sind, sollten nah zusammen platziert werden.
2. Knoten sollten nicht zu nah aneinander platziert werden.

Dazu nutzen wir den Algorithmus von Fruchterman und Reingold [10], wobei wir darauf verzichten, das Gebiet in seiner Breite und Höhe einzuschränken:

- `void erstelleLayout(Graph graph, int flaeche)`

  Die Methode berechnet für den übergebenen Graphen `graph` ein neues Layout, basierend auf der Ganzzahl `flaeche`. Diese codiert die optimale Distanz, die Knoten zueinander haben sollten und bestimmt die maximale Verschiebung der Knoten in einer Iteration des Algorithmus. Folgende Beziehung gilt somit: Je kleiner der Wert `flaeche`, desto kleiner auch das Gebiet, das das Layout einnimmt.

Klasse `Funktionen`

Die Klasse `Funktionen` stellt diverse Methoden bereit, die wiederkehrend an verschiedenen Stellen von OverlapR verwendet werden. Ein wichtiges Attribut dieser Klasse ist der Wert `THRESHOLD` vom Typ `double`. Dieser codiert die Genauigkeit des Auffindens einer Überlappung. Wir verwenden hierfür den Wert 0.01, welcher ausdrückt, dass der berechnete Überlappungsfaktor (2.2) größer als 1.01 sein muss, damit zwischen den berechneten Knoten eine Überlappung festgestellt wird. Dieser Wert liefert zufriedenstellende Ergebnisse und verbessert die Laufzeit des Programms.
Die wichtigsten Methoden, die in dieser Klasse implementiert sind, sind folgende:

- `double abstandKnoten(Knoten u, Knoten v)`

  Berechnet den Abstand zwischen den übergebenen Knoten `u` und `v`.

- `boolean overlapVorhanden(Knoten u, Knoten v)`

  Gibt `true` zurück, wenn die Knoten `u` und `v` sich überlappen und `false` wenn nicht.

- `boolean overlapsVorhanden(Graph graph)`

  Sucht an den Kanten eines Graphen nach Überlappungen der Label-Boxen. Sobald eine gefunden wurde, wird `true` zurückgeliefert. Wird keine Überlappung gefunden, so wird `false` zurückgegeben.

- `void scaleLayoutInch(Graph graph)`

  Skaliert das Layout eines Graphen, so dass die durchschnittliche Kantenlänge des Graphen 1 inch (in Java 72 units[11]) beträgt.

- `void scaleLayoutBoxen(Graph graph, int boxVF)`

  Skaliert das Layout eines Graphen, so dass die durchschnittliche Kantenlänge des Graphen das `boxVF`-fache der durchschnittlichen Label-Boxengröße (siehe Abschnitt 5.2) beträgt.

- `void scaleLayout2D(Graph graph, Graph vorbild, double faktor)`

  Skaliert das Layout des Graphen `graph` auf das angegebene `faktor`-fache eines anderen Layouts `vorbild`. Hierbei wird nicht bezüglich der Kantenlänge skaliert, sondern bezüglich der x- und y-Dimensionen. Wir nutzen diese Methode in unseren Vergleichen, um das Initial-Layout auf das 0.85-fache des Platzbedarfes, den GTree braucht, zu skalieren, um dann PRISM auszuführen (siehe Abschnitt 5.2).

- `double ueberlappungsGrad(Graph graph)`

  Berechnet den Überlappungsgrad des Graphen (siehe Abschnitt 5.2).

Klasse `GTree`

Diese Klasse implementiert den GTree-Algorithmus, wie er in [22] beschrieben wird:

- `Graph doGTree(Graph graph, double scale)`

  Diese Methode berechnet einen überlappungsfreien Graphen aus einem übergebenen Graphen und gibt diesen zurück. Der Wert scale gibt hierbei einen Skalierungsfakor s_{ij} an, wie er analog in PRISM verwendet wird (2.5).

- `GrowAtNode(Graph graph, BaumKnoten<Knoten> i, double scale)`

  In dieser Funktion ist Algorithmus 2 implementiert. Es handelt sich um eine rekursive Funktion.

- `berechneMinimalAufspannendenBaum(Graph graph, double[][] costFunc)`

  Diese Methode berechnet aus einem Graphen einen minimal aufspannenden Baum anhand der Kostenfunktion `costFunc`. Da in OverlapR alle Kanten in Adjazenzmatrizen kodiert sind, ist auch die Kostenfunktion c (2.6 bzw. 2.7) in einer $n \times n$ -

[11] http://www.oracle.com/technetwork/articles/java/java2dpart1-137217.html

Matrix gespeichert. Ist beispielsweise die Adjazenzmatrix folgende:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix},$$

so könnte die Kostenfunktion an den Kanten (Matrix-Eintrag ist 1) des Graphen folgende Gewichtungen berechnet haben:

$$\begin{bmatrix} 0 & 0 & 1,5 & 0 & 2,7 \\ 0 & 0 & 1,8 & 22 & 28 \\ 1,5 & 1,8 & 0 & 0 & 5 \\ 0 & 22 & 0 & 0 & 1,7 \\ 2,7 & 28 & 5 & 1,7 & 0 \end{bmatrix}.$$

Anhand dieser Matrix wird dann der minimal aufspannende Baum berechnet. Um den Baum zu berechnen, verwenden wir Prim's Algorithmus wie er in [24] beschrieben wird und Objekte der oben vorgestellten Klasse `BaumKnoten<T>`.

- `double[][] berechneC(Graph graph)`

  Berechnet die Kostenfunktion c (2.6 bzw. 2.7) für den Graphen und speichert das Ergebnis in einer $n \times n$ - Matrix (s.o.).

- `double berechneDist(Rectangle2D box1, Rectangle2D box2)`

  Diese Methode berechnet die Distanz zwischen zwei Label-Boxen (2.6) anhand ihrer Lagebeziehungen.

Klasse `PRISM`

Die Klasse `PRISM` implementiert den Algotithmus PRISM, wie er in [11] beschrieben wird:

- `Graph doPRISM(Graph graph, double scale)`

  Diese Methode berechnet einen überlappungsfreien Graphen aus einem übergebenen Graphen und gibt diesen zurück. Der Wert `scale` gibt hierbei einen Skalierungsfakor s_{ij} an wie er in PRISM verwendet wird (2.5).

- `double[][] berechneDistanzMatrix(Graph graph, double scale)`

  Berechnet die idealen Distanzen zwischen Label-Boxen, sodass diese disjunkt platziert werden und speichert diese Werte in einer $n \times n$ - Matrix. Diese Matrix wird dann für die Stress Majorization genutzt.

Klasse `StressMajorization`

Hier wird die Stress Majorization implementiert, wie sie in [12] beschrieben wird.

- `double[][] doStressMajorization(Graph graph, double[][] d)`

  Diese Methode gibt neue Knoten-Koordinaten in Form einer $n \times 2$ - Matrix zurück. Diese wurden anhand von d_{ij}-Werten (2.3), die in der Matrix (zweidimensionales Array von Typ `double`) `d` gespeichert sind, für den übergebenen Graphen berechnet. Um die Berechnungen durchzuführen, verwenden wir die Bibliothek Apache Commons Math[12]. Für das Lösen der linearen Gleichungssysteme nutzen wir insbesondere das CG-Verfahren, das in der Klasse `org.apache.commons.math3.linear.ConjugateGradient` implementiert ist. Hierbei verwenden wir als Stop-Kriterium $\delta = 0.0001$. Das CG-Verfahren stoppt, wenn $\|r\| \leq \delta\|b\|$, wobei der b Lösungsvektor im Verfahren ist, r der Residualvektor und δ eine benutzerdefinierte Toleranz [13]. Wir folgen den Autoren von PRISM [11] und setzen die Anzahl der Iterationen innerhalb der Stress Majorization auf 1 (in OverlapR `int ITER = 1`), da dies die Laufzeit verbessert und eine höhere Anzahl an Iterationen nicht zwangsläufig in einem besseren Layout münden [11].

Klasse `Random`

Die Klasse `Random` bietet verschiedene Methoden für das Erstellen von zufälligen Daten an:

- `void randomLabel(int minLaenge, int maxLaenge, int schriftGr, Graph graph)`

  Erzeugt für die Knoten des Graphen neue Label und damit auch Label-Boxen (s.o.). Dabei hat der Text für jeden Knoten eine Länge zwischen `minLaenge` Zeichen und `maxLaenge` Zeichen und einen Schriftgrad der Größe `schriftGr`.

- `Graph erdoesRenyiGraph(int n, double p)`

  Erzeugt aus `n` Knoten und der Wahrscheinlichkeit `p` einen Erdös-Rényi-Graphen. `p` gibt dabei an, wie hoch die Wahrscheinlichkeit ist, dass ein Knoten eine Kante zu einem beliebig anderen Knoten hat. Wir verwenden dafür den Algorithmus 1 aus [23].

- `void randomLayout(Graph graph, int breite, int hoehe)`

  Berechnet für den Graphen ein neues Layout, wobei die Knoten auf einer Fläche der Größe `breite` $\times$ `hoehe` zufällig platziert werden.

[12] http://commons.apache.org/proper/commons-math/
[13] http://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/linear/ConjugateGradient.html

Klasse `Scanline`

Diese Klasse implentiert den Scanline-Algorithmus wie er in [8] beschrieben ist.

- `boolean doScanLine(Graph graph)`

  Die Funktion findet Überlappungen in dem Graphen `graph` und fügt Kanten zwischen Knoten hinzu, deren Label-Boxen sich überschneiden. Die Funktion liefert zudem `true` zurück, wenn Überlappungen gefunden wurden und `false`, wenn nicht. Dieser Wert gilt als Abbruchkriterium in der zweiten Schleife von PRISM bzw. von GTree.

Klasse `Vergleich`

Die Klasse `Vergleich` implementiert die Berechnung der Werte für das Vergleichen von Graphenlayouts aus Kapitel 3 dieser Arbeit:

- `double area(Graph graph)`

  Diese Funktion berechnet die Fläche, die der Graph `graph` einnimmt. Siehe Kapitel 3.4.

- `double deltaDIST(Graph ori, Graph neu)`

  In dieser Funktion wird der Wert σ_{dist} zwischen den Graphen `ori` und `neu` berechnet. Siehe Kapitel 3.1.

- `double deltaDISP(Graph ori, Graph neu)`

  `deltaDISP` berechnet den Wert σ_{disp} zwischen den Graphen `ori` und `neu`, wobei `ori` als Initial-Layout betrachtet wird. Siehe Kapitel 3.2.

- `double kClosestNeighbors(Graph ori, Graph neu, int k)`

  Diese Methode berechnet den Wert cn_k zwischen den Graphen `ori` und `neu`, wobei `k` die Anzahl der zu betrachtenden Nachbarknoten ist. Siehe Kapitel 3.3.

- `double seitenVerhaeltnis(Graph graph)`

  In dieser Funktion wird das Seitenverhältnis berechnet und zurückgegeben. Siehe Kapitel 3.5.

4.4 Datenverarbeitung

Die Klassen für die Datenverarbeitung finden sich im Paket `fp.overlapr.daten`.

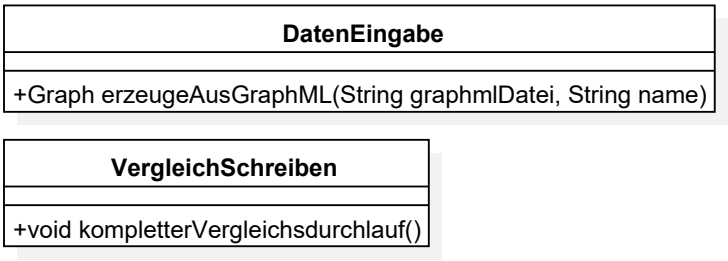


Abbildung 4.6: Klassendiagramme des Pakets `fp.overlapr.daten`

Klasse `DatenEingabe`

In dieser Klasse wird das Auslesen eines Graphen aus einer `graphML`[14]-Datei implementiert. `graphML` ist ein Dateiformat für Graphen und basiert auf der Auszeichnungssprache `xml`[15]. Somit ist es leicht verständlich und gut zu implementieren. Für das Auslesen einer graphML-Datei verwenden wir die JDom-Bibliothek[16]. Wir berücksichtigen hierbei neben Knoten und Kanten auch die x- und y-Koordinaten von Knoten, wenn sie in der Datei vorhanden sind. Somit lassen sich auch Layouts testen, die außerhalb von OverlapR erstellt wurden. Listing 1 zeigt eine `graphML`-Datei, die einen Graphen mit fünf Knoten (`<node>`) und zehn Kanten (`<edge>`) beschreibt, wobei die Platzierung der Knoten sowie Knoten-Label nicht genannt werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
␣http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
<graph id="G" edgedefault="undirected">
<node id="node0"/>
<node id="node1"/>
<node id="node2"/>
<node id="node3"/>
<node id="node4"/>
<edge source="node0" target="node4"/>
<edge source="node0" target="node1"/>
<edge source="node1" target="node2"/>
<edge source="node2" target="node3"/>
<edge source="node3" target="node4"/>
<edge source="node0" target="node3"/>
<edge source="node0" target="node2"/>
<edge source="node4" target="node1"/>
<edge source="node3" target="node1"/>
<edge source="node4" target="node2"/>
</graph>
</graphml>
```

Listing 1: graphML-Datei ohne Koordinaten

Listing 2 dagegen zeigt einen mit `graphML` kodierten Graphen, in dem auch die Bezeich-

[14] http://graphml.graphdrawing.org
[15] https://www.w3.org/XML
[16] http://www.jdom.org

nungen (`key="label"`) und die Platzierungen der Knoten (`key="x"` und `key="y"`) berücksichtigt sind.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
␣http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
<graph id="G" edgedefault="undirected">
<node id="node0">
  <data key="label">Gustav</data>
  <data key="x">34</data>
  <data key="y">45</data>
</node>
<node id="node1">
  <data key="label">Justus</data>
  <data key="x">123</data>
  <data key="y">45.32</data>
</node>
<node id="node2">
 <data key="label">Jana</data>
  <data key="x">32</data>
  <data key="y">-6</data>
</node>
<node id="node3">
  <data key="label">Frau Jansson</data>
  <data key="x">-34</data>
  <data key="y">326.2</data>
</node>
<edge source="node0" target="node1"/>
<edge source="node1" target="node2"/>
<edge source="node2" target="node3"/>
<edge source="node0" target="node3"/>
<edge source="node0" target="node2"/>
<edge source="node3" target="node1"/>
</graph>
</graphml>
```

Listing 2: graphML-Datei mit Koordinaten

Das Dateiformat `graphML` bietet weitaus mehr Auszeichnungen und Möglichkeiten als wir verwenden. Wir verweisen ausdrücklich auf die Spezifikation[17] und auf die Einführung[18] in `graphML`.

- `Graph erzeugeAusGraphML(String graphmlDatei, String name)`

  Diese Methode erzeugt einer `graphML`-Datei ein Objekt der Klasse Graph. Der Pfad zur Datei wird mit der Zeichenkette `graphmlDatei` übergeben. Die Zeichenkette `name` gibt den Namen des Graphen an.

Klasse `VergleichSchreiben`

Diese Klasse implementiert einen gesamten Vergleichsdurchlauf eines Graphen. Dieser Durchlauf wird in Abschnitt 5.2 ausführlich dargestellt.

[17] http://graphml.graphdrawing.org/specification.html
[18] http://graphml.graphdrawing.org/primer/graphml-primer.html

4.5 Benutzeroberfläche

Die Klassen für die Benutzeroberfläche befinden sich im Paket `fp.overlapr.gui`. Um die GUI zu erstellen, verwenden wir das Eclipse Plug-In WindowBuilder[19]. Dieser ist ein What-You-See-Is-What-You-Get Designer, mit dem sich komfortabel Benutzeroberflächen erstellen lassen.
Im Folgenden stellen wir die einzelnen Bedienelemente vor und erläutern kurz die Bedienung von OverlapR.

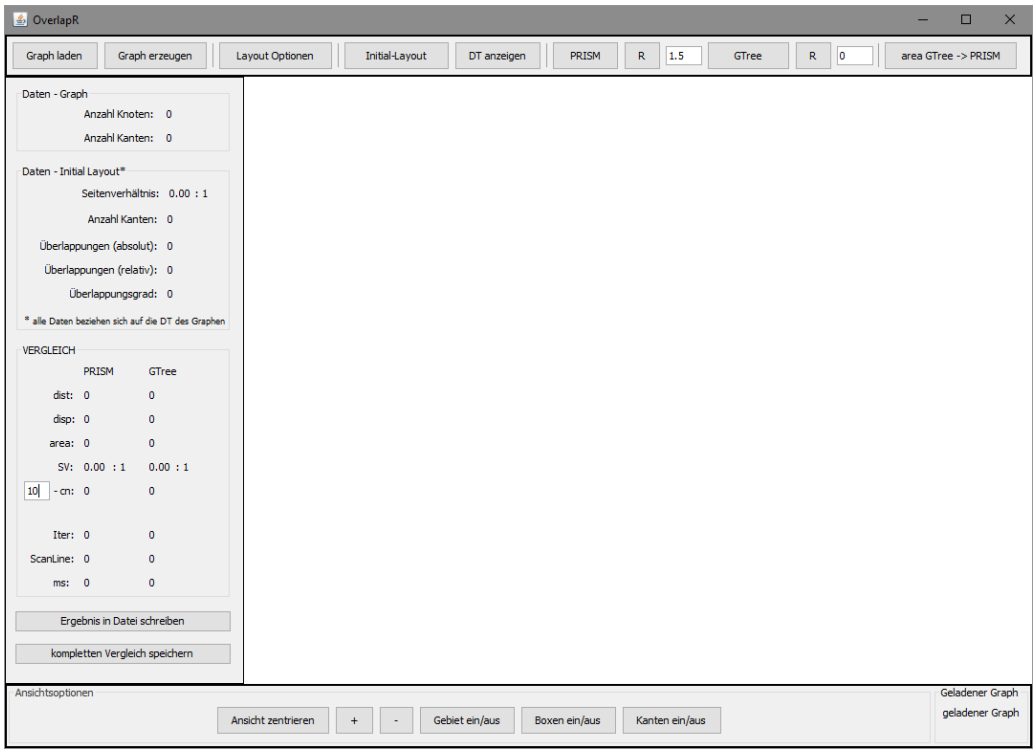


Abbildung 4.7: Hauptfenster OverlapR

Das Hauptfenster (Abbildung 4.7) besteht im Wesentlichen aus vier Komponenten. Oben sind die Bedienelemente, mit denen sich Graphen laden, erzeugen und layouten lassen. Im linken Informationskasten sind Daten zum Graphen und die Vergleichswerte ablesbar. Unten befinden sich Buttons, mit denen wir die Ansicht des Layouts anpassen können sowie der Name des geladenen Graphen. Auf die weiße Fläche in der Mitte wird der jeweilige Graph gezeichnet. Mit der Maus kann auf dieser das Layout verschoben werden und mit dem Mausrad kann rein- bzw. rausgezoomt werden. Wir stellen im folgenden Abschnitt die einzelnen Komponenten näher vor.

[19] https://eclipse.org/windowbuilder

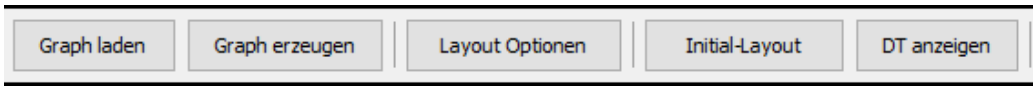


Abbildung 4.8: Bedienelemente oben - linker Teil

Mit dem Button „Graph laden" öffnet sich ein Standard-Dialog zum Laden von Dateien. Hierbei werden wie unter Abschnitt 4.4 nur `graphML`-Dateien akzeptiert. Ein Klick auf „Graph erzeugen" lädt das Fenster aus Abbildung 4.10, mit dem sich ein Erdös-Rényi-Graph [23] erzeugen lässt. Mit dem Button „Layout Optionen" öffnet sich das Fenster aus Abbildung 4.14, welches weiter unten in diesem Abschnitt beschrieben wird. Mit „Initial-Layout" wird das Initial-Layout gezeichnet und dargestellt. Der Button „DT anzeigen" zeigt die Kanten der Delaunay-Triangulierung des Graphen, der gerade auf der Zeichenfläche dargestellt wird.

Abbildung 4.9: Bedienelemente oben - rechter Teil

Der Button „PRISM" berechnet und zeigt das überlappungsfreie Layout des Initial-Layouts, das durch den Algorithmus PRISM berechnet wurde. Der Button „R" rechts daneben setzt ein bereits berechnetes Layout zurück, so dass beispielsweise ein anderer Skalierungsfaktor s_{ij} (2.5), der in das Textfeld daneben gesetzt wird, getestet werden kann. Analog dazu verhalten sich sie Schaltflächen daneben zum Algorithmus GTree. Der Wert 0 als Skalierungsfaktor bedeutet, dass GTree bzw. PRISM ohne diesen ausgeführt wird.
Der Button mit der Beschriftung „area GTree -> PRISM" skaliert das Initial-Layout zunächst auf das 0.85fache des Gebietes, das GTree verwendet und führt dann PRISM aus. In unseren Vergleichen spielt dieser Test eine tragende Rolle (siehe Abschnitt 5.2).

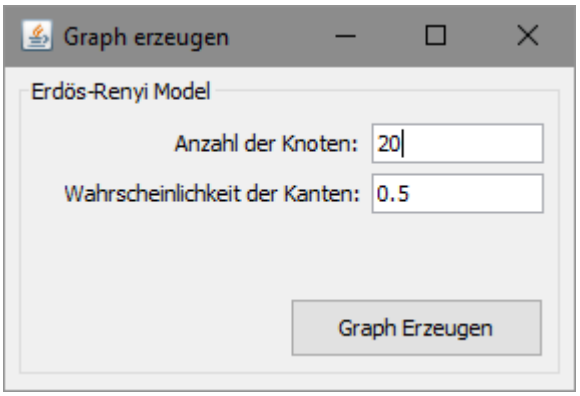


Abbildung 4.10: Fenster „Graph erzeugen"

Im linken Teil des Hauptfensters sind verschiedene Daten des Graphen ablesbar. Die

Daten werden immer dann aktualisiert, wenn das Initial-Layout geändert wird, ein neuer Graph geladen wird oder wenn die Algorithmen PRISM und GTree ausgeführt werden.

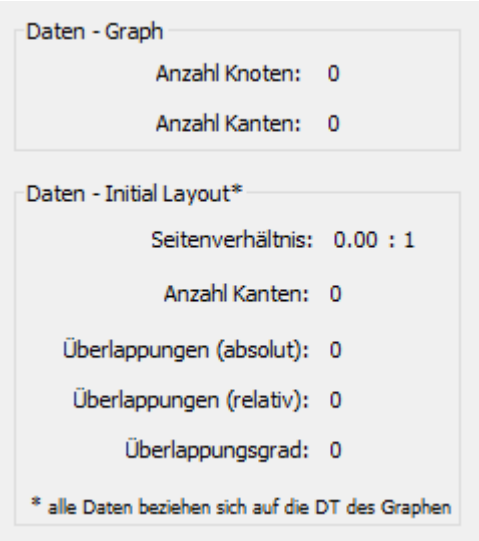


Abbildung 4.11: Linker Informationskasten - oberer Teil

Abbildung 4.11 zeigt den oberen Teil des linken Informationskastens. Hier werden die Knotenanzahl und die Kantenanzahl des betrachteten Graphen gezeigt. Zudem werden unter „Daten - Initial Layout" das Seitenverhältnis, die Anzahl der Kanten der Delaunay-Triangulierung sowie Daten zu Überlappungen dargestellt. Da PRISM und GTRee aus dem Eingabe-Graphen zunächst eine Delaunay-Triangulierung berechnen, beziehen sich diese Daten immer auf Letztgenannte. „Überlappungen (absolut)" gibt hierbei an, an wievielen Kanten der Delaunay-Triangulierung sich Label-Boxen überschneiden. Daraus lässt sich dann der relative Wert berechnen, der unter „Überlappungen (relativ)" angezeigt wird. Den „Überlappungsgrad" erläutern wir unter Abschnitt 5.2 eingehend.

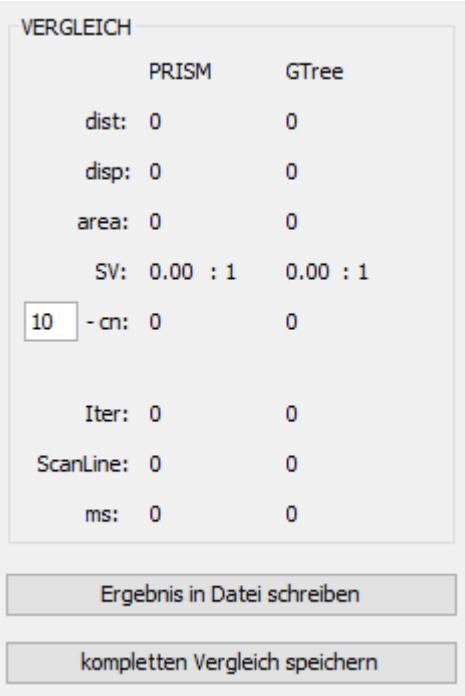


Abbildung 4.12: Linker Informationskasten - unterer Teil

Abbildung 4.12 stellt den unteren Teil des linken Informationskastens dar. Hier werden die Vergleichswerte dargestellt wie sie in Abschnitt 3 definiert wurden. Das Textfeld von „-cn" nimmt Werte für k im Wert cn_k (3.6) an. PRISM und GTree müssen bei einer Änderung dann jedoch neu ausgeführt werden. SV steht hier für Seitenverhältnis. Die unteren drei Zeilen geben Werte zur Anzahl der Iterationen der Algorithmen, zur Anzahl der Iterationen, bei denen der Scanline-Algorithmus verwendet wurde, sowie zur Laufzeit in Millisekunden an. Ein Klick auf den Button „Ergebnis in Datei speichern" sichert die Ergebnisse des Vergleichs in einer Text-Datei mit dem Namen `ergebnis.txt`, welche sich im gleichen Verzeichnis wie das Programm befindet. Ist sie noch nicht vorhanden, so wird sie erzeugt und jedes Ergebnis in einer neuen Zeile gespeichert. Die Datei kann mit einem beliebigen Tabellen-Programm ausgelesen werden, da die Werte mit einem Tabulator getrennt sind. Die Reihenfolge der gespeicherten Ergebnisse ist dabei wie folgt:
Name des Graphen - Anzahl Knoten - Anzahl Kanten - Skalierung - Überlappungen (relativ) - Überlappungsgrad - Seitenverhältnis - dist PRISM - dist GTree - disp PRISM - disp GTree - area PRISM - area GTree - SV PRISM - SV GTree - cn PRISM - cn GTree - Iter PRISM - Iter GTree - Scanline PRISM - Scanline GTree - ms PRISM - ms GTree.
Der Button „kompletten Vergleich speichern" führt einen gesamten Vergleichsdurchlauf (siehe Abschnitt 5.2) durch und speichert diesen in einer Datei namens `ergebnisse.txt`. Auch hier wird sie erzeugt, wenn sie nicht vorhanden ist. Weitere Durchläufe werden dann in dieser gespeichert.
Um schneller ermitteln zu können, welches Layout den einzelnen Werten nach besser ist,

werden die Ergebnisse eingefärbt. Hierbei bedeutet grün, dass der jeweilige Algorithmus bessere Ergebnisse liefert, rot steht für schlechtere Ergebnisse. Gelb eingefärbte Ergebnisse bedeuten, dass beide Werte gleich sind.
Um die Erläuterung der Bedienelemente des Hauptfensters abzuschließen, gehen wir noch kurz auf den unteren Teil der GUI ein.

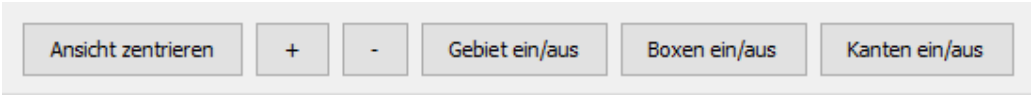


Abbildung 4.13: Bedienelemente unten

Neben den selbsterklärenden Buttons „Ansicht zentrieren" und „+" (Reinzommen) und „−" (Rauszoomen), finden sich hier drei weitere Buttons, mit denen die Ansicht verändert werden kann. Hierbei lässt sich mit dem Button „Gebiet ein/aus" das Gebiet anzeigen, das das Layout einnimmt. Es wird grün dargestellt und gibt die Breite sowie die Höhe an. Mit dem Button „Boxen ein/aus" lassen sich die Label-Boxen der Knoten ausblenden, um beispielsweise die Kanten besser betrachten zu können. Wenn die Boxen „ausgeschaltet" sind, werden Knoten als kleiner roter Kreis dargestellt. Der Button „Kanten ein/aus" schließlich dient dazu, die Kanten des Graphen ein- bzw. auszublenden.
Wesentlicher Bestandteil von OverlapR ist das Erstellen von Layouts, anhand derer PRISM und GTree getestet werden können. Dazu öffnet sich durch den Button „Layout Optionen" im Hauptfenster, das Fenster aus Abbildung 4.14. Dieses bietet diverse Optionen an, um ein Layout zu erzeugen:

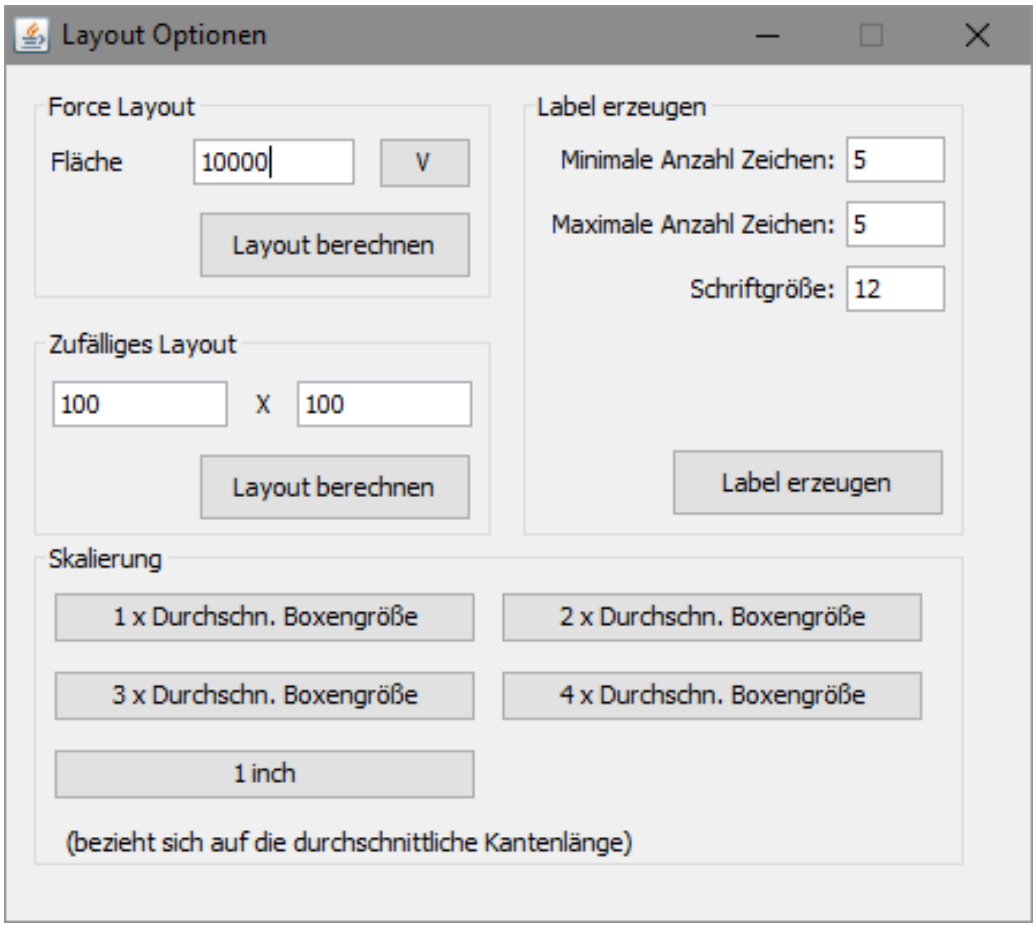


Abbildung 4.14: Fenster „Layout Optionen”

In der Komponente „Force Layout” kann ein neues Layout durch den Force Directed Placement-Algorithmus berechnet werden [10]. Das Textfeld nimmt hierbei die Größe einer Fläche entgegen. Der Button „V” schlägt eine Größe anhand der Knotenanzahl vor und Button „Layout berechnen” erzeugt schließlich ein neues Layout. In der Komponente „Zufälliges Layout” kann mittels der Angaben von Breite (erstes Textfeld) und Höhe (zweites Textfeld) ein Layout erstellt werden, bei dem Knoten zufällig platziert werden. Des Weiteren bietet dieses Fenster die Komponente „Label erzeugen” an, mit deren Hilfe neue Knoten-Label erzeugt werden. Hierbei können die Länge sowie die Schriftgröße bestimmt werden. Als Zeichen werden hierfür große „A”s verwendet.
Als weitere wichtige Funktion kann in diesem Fenster das Layout skaliert werden. Alle Angaben beziehen sich hierbei auf die durchschnittliche Kantenlänge. Es werden Buttons zur Skalierung angeboten, die das Layout auf das 1- bis 4-fache der durchschnittlichen Label-Boxen-Größe oder auf 1 inch skalieren. Näheres dazu siehe Abschnitt 5.2.

4.6 Anwendungsfall-Beispiel

In diesem Abschnitt stellen wir einen typischen Anwendungsfall für das Programm OverlapR vor. Wie bereits beschrieben, dient das Programm hauptsächlich dazu, die überlappungsfreien Layouts, die mit PRISM bzw. GTree erzeugt wurden, zu vergleichen. Wir schildern solch einen Vergleich anhand des Graphen namens xx. Dieser ist gespeichert als `graphML`-datei und enthält Angaben zu Labeln und Koordinaten der Knoten (s.a. Abschnitt 4.4).

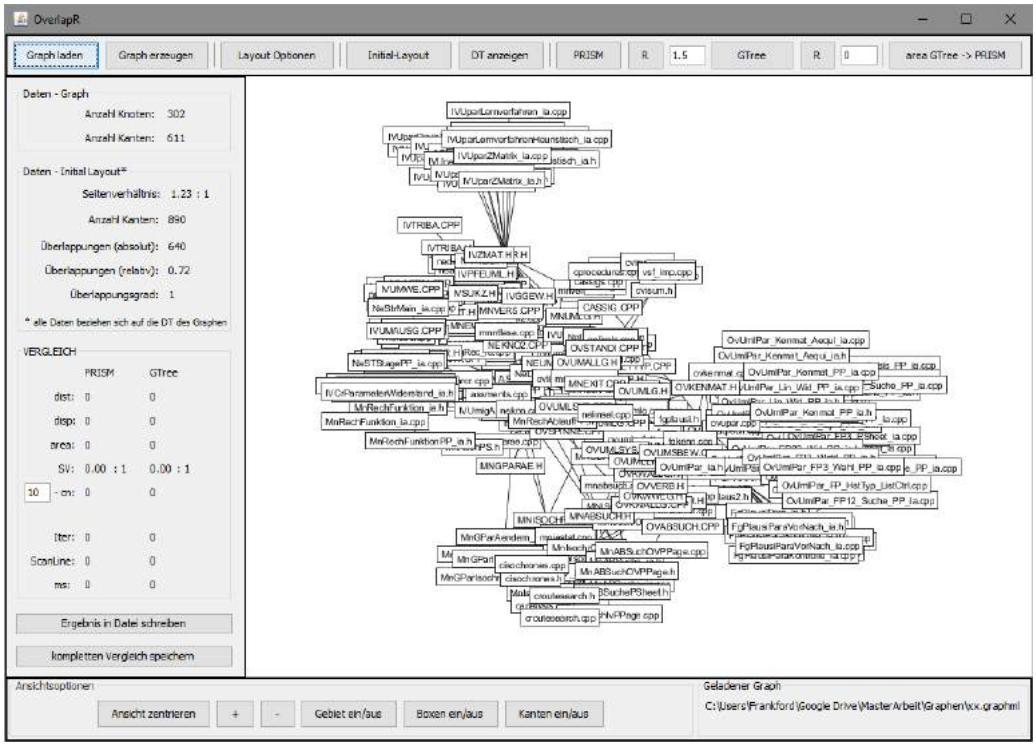


Abbildung 4.15: Graph xx wurde geladen

Nachdem wir den Graph geladen haben, sehen wir links im Hauptfenster die Daten zum Graphen und zum Inital-Layout (Abbildung 4.15). Mithilfe der „Layout Optionen" erstellen wir ein neues Layout: Zunächst erzeugen wir Label, die die Länge 2 haben und skalieren dann das Layout auf das 2-fache der durchschnittlichen Boxengröße. Abbildung 4.16 zeigt das neue Layout. Die Werte links wurden entsprechend aktualisiert.

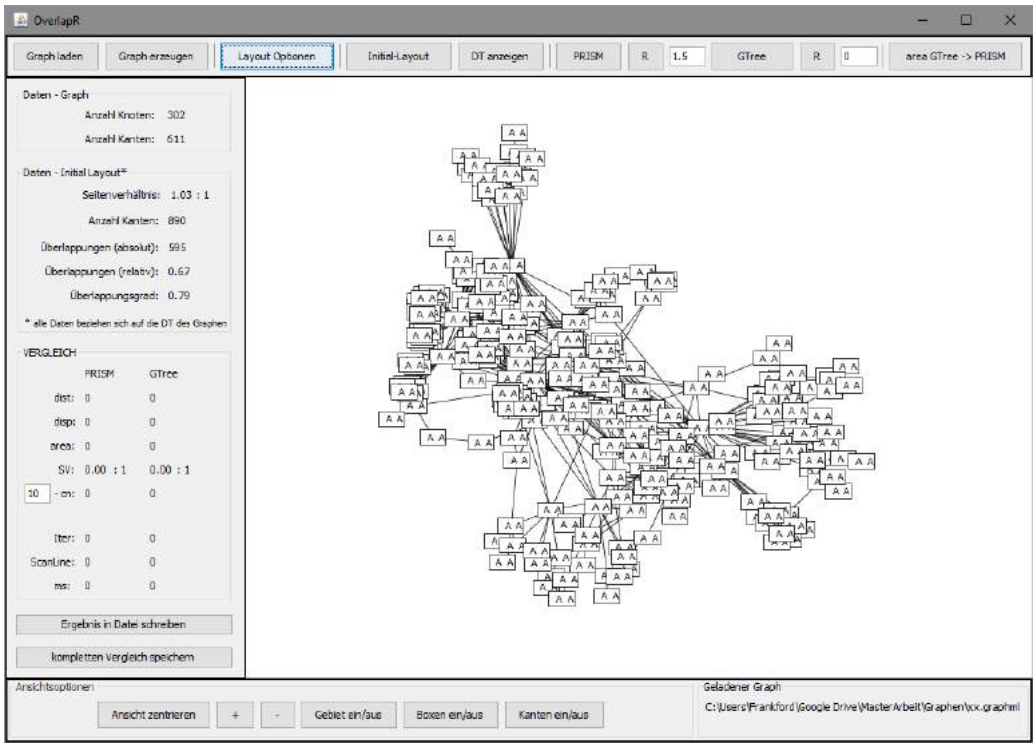


Abbildung 4.16: neues Layout von xx

Die Label-Boxen des so erzeugten Layouts wollen wir nun disjunkt platzieren. Dafür führen wir PRISM mit dem Skalierungsfaktor 1.5. aus. Abbildung 4.17 zeigt das neue Layout, wobei wir rausgezoomt haben, um den gesamten Graphen sehen zu können. Links werden bereits die Vergleichswerte aus Abschnitt 3 angegeben. Außerdem sehen wir hier die Angaben zur Laufzeit.

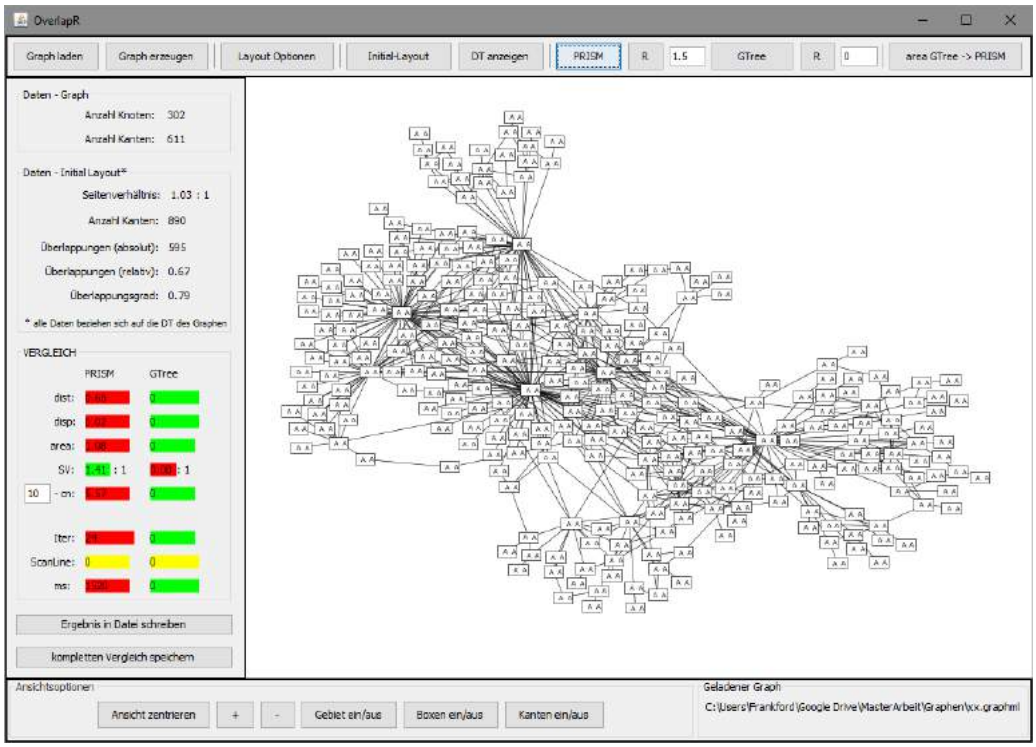


Abbildung 4.17: Überlappungsfreies Layout von xx durch PRISM

Als nächstes lassen wir ein überlappungsfreies Layout durch GTree berechnen. Hierfür verwenden wir auch den Skalierungsfaktor von 1.5. Abbildung 4.18 zeigt das durch GTree berechnete Layout. Zudem ist nun links der Vergleich ablesbar. Es ist in diesem Fall zu sehen, dass GTree für die Werte σ_{dist}, cn_{10}, Seitenverhältnis und für die Laufzeiten besser ist. Die Werte für σ_{disp} sind identisch.

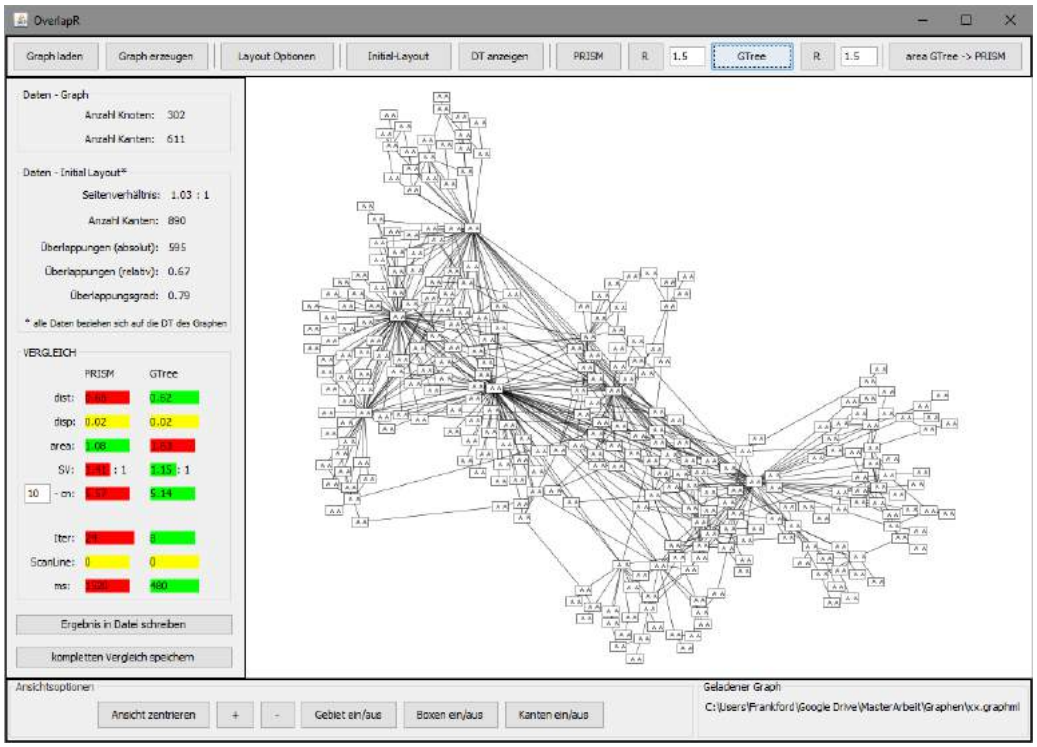


Abbildung 4.18: Überlappungsfreies Layout von xx durch GTree

Schließlich speichern wir unser Ergebnis mit einem Klick auf den Button „Ergebnis in Datei schreiben", um es später auswerten zu können. Die Zeile in der Datei `ergebnis.txt` sieht dann folgendermaßen aus (aus Platzgründen hier mit Zeilenumbrüchen dargestellt):

```
xx.graphml      302     611     2       0.67
0.79    1.03    0.65    0.62    0.02    0.02
1.08    1.63    1.41    1.15    5.57    5.14
24      8       0       0       1520    480
```

Listing 3: ergebnis.txt

5 Vergleich PRISM - GTree

Dieser Abschnitt hat den Vergleich der beiden unter Abschnitt 2 vorgestellten Algorithmen zum Platzieren disjunkter Boxen zum Inhalt. Hierbei werden aufgrund der Vergleiche und Tests, die in den Arbeiten zu PRISM [11] und GTree [22] durchgeführt wurden, Hypothesen (Abschnitt 5.1) entwickelt. Das Vorgehen bei diesem Vergleich ist Thema des Abschnittes 5.2. Hier beschreiben wir unter anderem die Auswahl der Testgraphen. Die Hypothesen werden in Abschnitt 5.3 verifiziert bzw. falsifiziert. Abschnitt 5.3.5 schließt den Vergleich mit einer Zusammenfassung der zentralen Ergebnisse.

5.1 Hypothesen

In der Arbeit zu PRISM [11] werden die 204 Graphen aus der *graphviz test suite*, die dem Quellcode von GraphViz[20] beiliegen sowie die Graphen aus der *rome suite*[21] (11534 Graphen) getestet. Anhand der Messwerte σ_{dist}, σ_{disp} und *area* wurde das durch PRISM berechnete überlappungsfreie Layout mit anderen Algorithmen verglichen, die Layouts mit einer Paltzierung disjunkter Label-Boxen erzeugen und im Text als VPSC [8], VORO [21] und ODNLS [19] bezeichnet werden.
Bei den Vergleichen wurden die Initial-Layouts auf eine durchschnittliche Kantenlänge von 1 inch skaliert, bevor die Algorithmen zum Entfernen von Überlappungen ausgeführt wurden. Die Ergebnisse sind folgende:

- PRISM produziert zum großen Teil bessere Layouts als VPSC und VORO.
- PRISM ist wesentlich schneller als ODNLS, VPSC und VORO.
- PRISM verbraucht weniger Platz als ODNLS, VPSC und VORO.
- ODNLS erstellt leicht bessere Layouts als PRISM, verbraucht aber erheblich mehr Platz und ist signifikant langsamer.
- Je größer das Initial-Layout skaliert wird, desto kleiner ist die Anzahl der Iterationen von PRISM und desto besser die Messwerte.
- Der Wert σ_{disp} kann eher angeben, wie sehr sich Layouts unterscheiden als σ_{dist}.

In der Arbeit zu GTree [22] wird ein direkter Vergleich zu PRISM durchgeführt. Auch dieser benutzt als Testgraphen dieselben Graphen aus der *graphviz test suite*, die auch die Autoren von PRISM verwendet haben. Zudem verwenden sie eine Sammlung von mehr als 10.000 Graphen. Leider lassen sich keine Aussagen über Knoten-Anzahl und Beschaffenheit der Sammlung treffen, da diese zum Zeitpunkt unseres Vergleiches nicht mehr verfügbar war. Es wird lediglich beschrieben, dass diese Kontrollflüsse von Methoden im .NET-Framework[22] visualisieren und die Knotenanzahl keine Tausende umfasst [22].

[20] Der Quellcode kann unter http://www.graphviz.org/Download_source.php heruntergeladen werden.
[21] Die Graphen haben eine Knotenanzahl zwischen 10 und 109 und können als graphML-Dateien unter http://www.graphdrawing.org/data.html heruntergeladen werden.
[22] https://www.microsoft.com/net

In der Arbeit zu GTree werden die Werte σ_{dist}, σ_{disp}, cn_k (wobei $k = 8, 9, 10, 11, 12$) und $area$ betrachtet. Die wichtigsten Ergebnisse des Vergleichs sind:

- GTree ist signifikant schneller als PRISM.
- GTree liefert in ca. 60% der Vergleiche einen besseren cn_k - Wert als PRISM
- GTree liefert in ca. 52% der Vergleiche einen besseren σ_{dist} - Wert als PRISM
- GTree liefert in ca. 59% der Vergleiche einen besseren σ_{disp} - Wert als PRISM
- PRISM braucht in ca. 85% der Vergleiche weniger Platz als GTree

Aus den dargestellten Ergebnissen, den Tabellen der Vergleiche in [11, 22], den in diesen Arbeiten dargestellten Layouts und der Tatsache, wie die beiden Algorithmen eine Überlappung auflösen (siehe Abschnitt 2.3), entwickeln wir folgende Hypothesen:

Hypothese 1 GTree braucht weniger Iterationen als PRISM, um ein überlappungsfreies Layout zu erzeugen.

Hypothese 2 PRISM verändert das Seitenverhältnis des Layouts stärker als GTree. Mit längeren Label-Boxen nimmt diese Verzerrung zu.

Hypothese 3 PRISM neigt eher dazu Knoten aneinander zu „packen" (siehe dazu Abschnitt 3.4) und damit kompaktere Layouts zu erzeugen.

Hypothese 4 Da GTree mehr Platz braucht und mit höherer Anfangsskalierung des Initial-Layouts bessere Ergebnisse erzielt werden (s.o.), gehen wir davon aus, dass PRISM bessere Ergebnisse liefert, wenn dieser Algorithmus ungefähr den Platz erhält, den GTree braucht, um eine Platzierung von disjunkten Boxen zu erreichen.

5.2 Vorgehen

Bei unseren Vergleichen verwenden wir den Force Directed Placement-Algorithmus, wie in Abschnitt 4.3 beschrieben. Als Fläche geben wir immer 10.000 an. Wir benutzen zum Berechnen der neuen Layouts in beiden Algorithmen einen Skalierungsfaktor von $s_{ij} = 1.5$ (2.5). Des Weiteren folgen wir den Autoren von PRISM und GTree und runden die Messwerte auf die zweite Nachkommastelle.
Damit die Reproduzierbarkeit der Ergebnisse so gut wie möglich gewährleistet ist, erzeugen wir in unseren Vergleichen für x^0 neue Label-Boxen und skalieren die durchschnittliche Kantenlänge des Layouts auf eine k-fache durchschnittliche Boxengröße b. Diese berechnen wir analog zu [11] mit

$$b = \frac{\sum_{i=0}^{n} \frac{(w_i + h_i)}{2}}{n} \tag{5.1}$$

- wobei n = Anzahl der Knoten
- w_i = Breite des Knotens v_i

- h_i = Höhe des Knotens v_i

In unseren Vergleichen verwenden wir für $k = \{1, 2, 3, 4\}$. Um Hypothese 4 zu prüfen, skalieren wird x^0 auf das 0.85^{23}-fache von x^G und führen dann PRISM aus, um x^P zu berechnen. Dann vergleichen wir x^G und x^P miteinander. In unserer Auswertung codieren wir den Skalierungsfaktor im letzteren Fall mit 0, sonst mit k.
Um auch verschiedene Boxen-Formate in die Vergleiche einzubeziehen, verwenden wir relativ kurze Label (2 Zeichen, entprechen ungefähr dem Format 2:1) und etwas längere (5 Zeichen, Format 4:1). Dabei codieren wir die kurzen Label mit 0 und die längeren mit 1. Damit wir die Initial-Layouts bezüglich der Überlappung von Label-Boxen numerisch charakerisieren können, berechnen wir die Werte λ_{rel} und λ_{grad}. Ersterer ist ein Wert, der angibt, wie hoch der Anteil der Kanten der Delaunay-Triangulierung ist, an denen sich Boxen überlappen. Letzterer (von uns als Überlappungsgrad bezeichnet) gibt an, wie hoch der Anteil der sich überlappenden Fläche im Bezug zur gesamten Fläche ist, die die Label-Boxen einnehmen. Wir berechnen die Werte wie folgt:

$$\lambda_{\text{rel}} = \frac{\text{anz}_{\text{ol}}}{|E_P|}, \tag{5.2}$$

wobei anz_{ol} die Anzahl der Überlappungen an den Kanten der Triangulierung E_P ist und

$$\lambda_{\text{grad}} = \frac{a_{\text{ol}}}{a}, \tag{5.3}$$

wobei

- a_{ol}= die gesamte Fläche, die die Überlappungen einnehmen. Wir berechnen dafür an jeder Kante der Delaunay-Triangulierung die Überlappungsfläche der jeweiligen Knoten und summieren diese Flächen auf. Um die Schnittfläche zu berechnen nutzen wir die Methode
  `createIntersection(Rectangle2D r)` der Klasse `Rectangle2D`, die wir für unsere Implementierung der Label-Boxen verwenden.

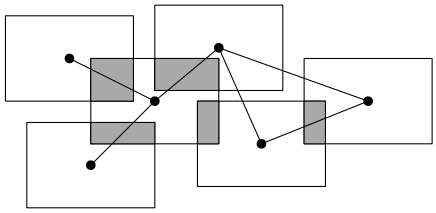
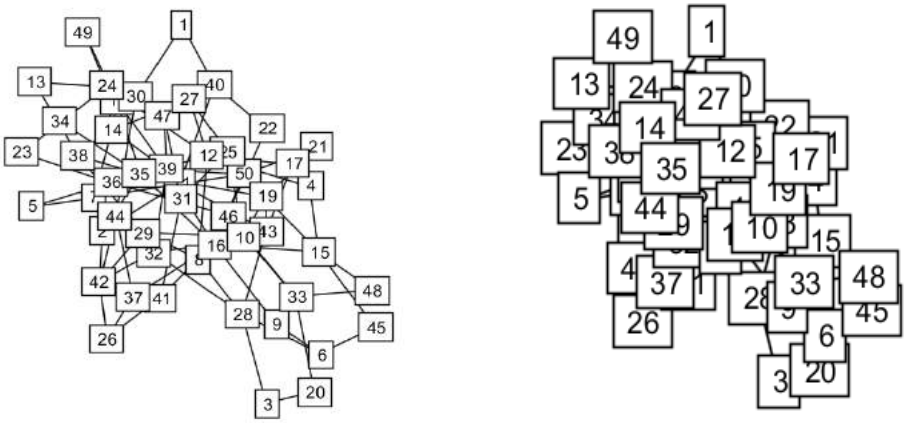


Abbildung 5.1: a_{ol} ist in der Abbildung die eingegraute Fläche

- a = die gesamte Fläche der Label-Boxen.

[23] Wir wählen das 0.85-fache und nicht etwa das 1-fache, damit PRISM noch Platz hat, um das Layout überlappungsfrei zu berechnen, so dass in etwa beide durch PRISM und GTree berechneten Layouts ähnlich viel Platz einnehmen.

Diese Werte beziehen sich auf die Delaunay-Triangulierung von x^0, weil PRISM und GTree diesen als Eingabegraphen nutzen. Abbildung 5.2 verdeutlicht beide Werte anhand eines Beispiels:



G_a G_b

Abbildung 5.2: Für G_a sind $\lambda_{\text{rel}} = 0.32$ und $\lambda_{\text{grad}} = 0.16$. Für G_b sind $\lambda_{\text{rel}} = 0.87$ und $\lambda_{\text{grad}} = 0.77$.

Als Vergleichswerte berechnen wir σ_{dist}, σ_{disp}, cn_k, $area$ und das Seitenverhältnis, wie in Abschnitt 3 beschrieben. Wir berechnen cn_k mit $k = 10$. Die Vergleiche aus [22] legen nahe, dass wenn ein Wert für cn_8 schlechter bei PRISM als bei GTree zu bewerten ist, dasselbe auch für beispielweise cn_{12} gilt. Auch in unseren ersten Vergleichen korrespondierten diese Werte mehrheitlich.
Wir definieren nun einen Testdurchlauf, den wir an jedem Graphen unserer Testgraphen durchführen:

1. Initial-Layout x^0 durch den Force Directed Placement-Algorithmus berechnen
2. kurze Label-Boxen für x^0 erzeugen
3. x^0 skalieren ($1b$), PRISM und GTree ausführen, x^G und x^P vergleichen
4. x^0 skalieren ($2b$), PRISM und GTree ausführen, x^G und x^P vergleichen
5. x^0 skalieren ($3b$), PRISM und GTree ausführen, x^G und x^P vergleichen
6. x^0 skalieren ($4b$), PRISM und GTree ausführen, x^G und x^P vergleichen
7. lange Label-Boxen für x^0 erzeugen
8. x^0 skalieren ($1b$), PRISM und GTree ausführen, x^G und x^P vergleichen
9. x^0 skalieren ($2b$), PRISM und GTree ausführen, x^G und x^P vergleichen

10. x^0 skalieren ($3b$), PRISM und GTree ausführen, x^G und x^P vergleichen
11. x^0 skalieren ($4b$), PRISM und GTree ausführen, x^G und x^P vergleichen
12. kurze Label-Boxen für x^0 erzeugen, x^0 skalieren ($1b$), GTree ausführen, x^0 auf das 0.85-fache von x^G skalieren, PRISM ausführen, x^G und x^P vergleichen
13. lange Label-Boxen für x^0 erzeugen, x^0 skalieren ($1b$), GTree ausführen, x^0 auf das 0.85-fache von x^G skalieren, PRISM ausführen, x^G und x^P vergleichen
14. Ergebnis in `ergebnisse.txt` speichern

In OverlapR kann ein gesamter Durchlauf mit dem Button „kompletten Vergleich speichern", der sich im Hauptfenster links befindet, ausgeführt werden. Gerade bei großen Graphen ($|V| > 500$) kann dies allerdings bis zu mehrere Minuten dauern.
Tabellen 1-3 stellen die Spalten eines Testdurchlaufs anhand des Graphen xx dar. Hierbei geben b die Skalierung (s.o.) an, LB die Label-Boxen-Größe (s.o.), SV das Seitenverhältnis, Iter die Iterationen der ersten Schleife in den Algorithmen, SL die Iterationen, in denen der ScanLine-Algorithmus verwendet wird und ms die Laufzeit in Millisekunden an.

Graph	$\|V\|$	$\|E\|$	b	LB	λ_{rel}	λ_{grad}	SV	σ_{dist} PR	σ_{dist} GT
xx	302	611	1	0	0.91	1.5	1.05	0.67	0.59
xx	302	611	2	0	0.67	0.79	1.03	0.65	0.61
xx	302	611	3	0	0.44	0.48	1.02	0.59	0.56
xx	302	611	4	0	0.33	0.33	1.02	0.49	0.48
xx	302	611	1	1	0.8	1.18	1.12	0.77	0.61
xx	302	611	2	1	0.49	0.59	1.06	0.74	0.6
xx	302	611	3	1	0.35	0.35	1.04	0.66	0.56
xx	302	611	4	1	0.26	0.21	1.04	0.55	0.46
xx	302	611	0	0	0.91	1.5	1.05	0.54	0.59
xx	302	611	0	1	0.8	1.18	1.12	0.6	0.61

Tabelle 1: Daten eines Durchlaufs - Teil 1

σ_{disp} PR	σ_{disp} GT	*area* PR	*area* GT	SV-PR	SV-GT	cn_{10} PR	cn_{10} GT
0.04	0.03	0.88	1.72	1.64	1.16	8.44	5.61
0.02	0.02	1.08	1.66	1.41	1.15	5.57	5.11
0.01	0.01	1.41	1.82	1.2	1.09	3.71	3.93
0.01	0.01	1.99	2.29	1.12	1.09	1.84	2.39
0.13	0.03	2.12	3.43	3.12	1.39	15.95	8.32
0.06	0.03	2.79	3.81	1.95	1.42	9.1	5.93
0.03	0.02	3.96	4.67	1.45	1.27	5.19	4.41
0.01	0.01	5.62	5.86	1.24	1.15	3.18	2.49
0.01	0.03	1.58	1.72	1.28	1.16	2.96	5.61
0.03	0.03	3.58	3.43	1.86	1.39	5.42	8.32

Tabelle 2: Daten eines Durchlaufs - Teil 2

Iter PR	Iter GT	SL PR	SL GT	ms PR	ms GT
25	17	0	0	1638	941
24	8	0	0	1545	441
21	10	0	0	1339	548
19	6	0	0	1274	327
42	9	17	81	3776	4686
33	6	16	5	3117	595
26	7	20	2	3030	490
19	5	16	1	2381	330
18	17	0	0	1178	891
28	9	20	81	3190	4642

Tabelle 3: Daten eines Durchlaufs - Teil 3

Nachdem wir unser Vorgehen beim Vergleich erklärt haben, gehen wir noch kurz auf unsere gewählten Testgraphen ein.
Auch wir betrachten Graphen aus der *rome suite*. Hierbei haben wir 200 Graphen zufällig ausgewählt. Die Anzahl der Knoten ist hierbei $10 \leq |V| \leq 100$. Zudem betrachten wir 200 Graphen, die mit dem Graphen Generator DAGmar [3] erzeugt wurden. Sie haben eine Mächtigkeit von $20 \leq |V| \leq 400$.
Um auch große Graphen bzw. Netzwerke zu vergleichen, vergleichen wir 200 sogenannte *small world*-Graphen, die nach dem Watts-Strogatz-Model [25] mittels der Software Mathematica[24] erzeugt wurden[25]. Die getesteten *small world*-Graphen haben eine Knotenanzahl von $550 \leq |V| \leq 1000$. Da diese Graphen sich stets ähnlich sehen, betrachten wir noch eine vierte Gruppe von Graphen, die wir ebenfalls mit Mathematica erzeugt haben[26]. Es handelt sich dabei um Graphen, die nach dem Barabási–Albert Modell generiert wurden [1]. Auch hier wählten wir eher größere Netzwerke mit $550 \leq |V| \leq 1000$.
Im weiteren Verlauf der Arbeit werden wir die vorgestellten Gruppen mit den Vergleichsgraphen wie folgt nennen: Rome, DAGmar, SmallWorld, BarabasiAlbert. Wenn wir alle Testgruppen meinen, verwenden wir auch den Begriff Stichprobe. Die von uns getesteten Graphen liegen dieser Thesis in elektronischer Form bei.
Abbildung 5.3 zeigt jeweils 3 Graphen aus den Testgruppen. Dabei verwenden wir das von uns implementierte Force Directed Placement (siehe Abschnitt 4.3) und skalieren die durchschnittliche Kantenlänge auf das 1-fache der durchschnittlichen Boxengröße. Um die Struktur der Graphen besser erkennen zu können, haben wir die Label-Boxen entfernt. Zudem sind die Ansichten der Graphen unterschiedlich skaliert.

[24] https://www.wolfram.com/mathematica/
[25] mit dem Befehl `RandomGraph[WattsStrogatzGraphDistribution[n, 0.05]]`
[26] mit dem Befehl `RandomGraph[BarabasiAlbertGraphDistribution[n, 1]]`

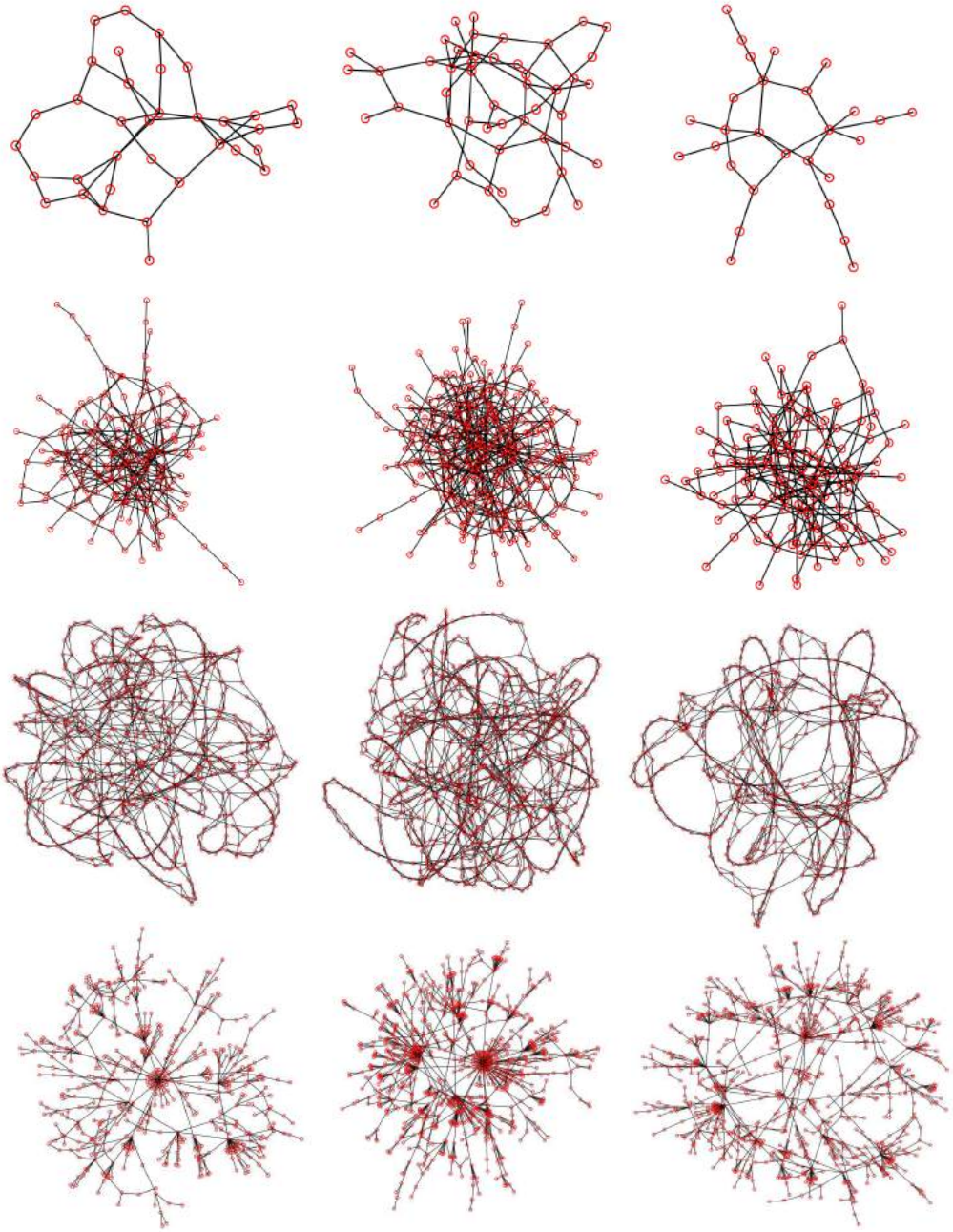


Abbildung 5.3: Beispielgraphen unserer vier Testgruppen Rome, DAGmar, SmallWorld, BarabasiAlbert (von oben nach unten)

5.3 Auswertung

Bei unserer Auswertung werden wir nicht näher auf die Laufzeiten eingehen, da unsere Implementierung wesentlich höhere Laufzeiten hat als die Laufzeiten, die in den Arbeiten zu PRISM und GTree angegebenen werden [22, 11]. Als Indikator für die Performanz

eines Algorithmus können die Iterationen herangezogen werden: Hier erzeugt ein niedrigerer Wert auch eine niedrigere Laufzeit [22, 11]. Dabei ist darauf zu achten, dass die zweite Schleife der Algorithmen, in der der Scanline-Algorithmus genutzt wird, wesentlich weniger Iterationen benötigt als die erste Schleife [11]. Aus den eben genannten Gründen werden wir hauptsächlich auf die qualitativen Messwerte eingehen. Die Auswertung wurde mit LibreOffice Calc[27] angefertigt. Die entsprechenden Tabellen unserer gesamten Auswertung liegen dieser Masterthesis in elektronischer Form bei.
Wir wollen zunächst auf die beiden Werte λ_{rel} und λ_{grad} im Allgemeinen eingehen. Beide hängen unmittelbar zusammen (der Korrelationskoeffizient nach Pearson ist $r \approx 0.98$), so dass wir im Folgenden nur noch λ_{grad} betrachten. Der Zusammenhang zwischen λ_{grad} und den Vergleichswerten σ_{dist}, σ_{disp} und cn_{10} ist ähnlich, wenn auch nicht so eindeutig. Über unsere gesamte Stichprobe zeigt sich, dass der Überlappungsgrad mit den Werten $\sigma_{\text{dist}}(r \approx 0.43)$, $\sigma_{\text{disp}}(r \approx 0.49)$ und $cn_{10}(r \approx 0.68)$ korreliert. Wir können folglich allgemein zusammenfassen: Je höher der Überlappungsgrad λ_{grad}, desto höher auch die Werte σ_{dist}, σ_{disp} und cn_{10}. Oder anders ausgedrückt: Je höher der Überlappungsgrad, desto mehr weichen die durch GTree bzw. PRISM berechneten Layouts x^G und x^P vom Initial-Layout x^0 ab. Da dieses Ergebnis nicht überrascht und zudem der Überlappungsgrad λ_{grad} auch mit der Skalierung korreliert ($r = -0.74$), werden wir ihn in unserer Auswertung nur bedingt berücksichtigen. Stattdessen werden wir die Skalierung von x^0 als Gruppierungskriterium verwenden. Zum einen, weil es die Auswertung der Ergebnisse erleichtert und zum anderen, weil eine Skalierung der durchschnittlichen Kantenlänge in x^0 einfach implementiert werden kann, was die Reproduzierbarkeit unseres Vergleiches erhöht.
Im Folgenden gehen wir in einzelnen Unterabschnitten auf die Testgruppen ein. Hier werden wir zunächst allgemein die Ergebnisse darstellen und anschließend die Hypothesen verifizieren bzw. falsifizieren. Wir schließen jeden Abschnitt mit einer Abbildung dreier zufällig gewählter Testgraphen aus der jeweiligen Gruppe.

5.3.1 Testgruppe Rome

Zunächst gehen wir bei der Auswertung auf die Testgruppe Rome ein. Die folgende Tabelle zeigt die Ergebnisse des Vergleichs als absolute Zahlen an. Hierbei bedeutet die Zahl in einer Zelle, dass der betreffende Algorithmus (PR für PRISM und GT für GTree) in dieser Anzahl von Fällen bessere Ergebnisse erzeugt hat. Waren die Werte gleich, so berücksichtigen wir diese nicht. Dies erklärt, warum die Summe nicht stets 200 ist. In Tabelle 4 beispielweise lässt sich ablesen, dass GTree bei einer Skalierung von $1b$ und einer Boxengröße LB = 0 (kleine Label-Boxen; siehe Abschnitt 5.2) in 96 Fällen einen besseren σ_{Disp}-Wert liefert als PRISM.

[27] LibreOffice Calc ist eine Tabellenkalkulations-Software und kann unter https://de.libreoffice.org/discover/calc/ kostenlos heruntergeladen werden.

		σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**		**Iter**	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	23	165	44	96	172	1	67	110	26	172	3	197
	1	58	125	29	132	164	10	51	125	21	177	1	199
2	**0**	104	22	62	0	127	7	151	10	85	67	0	173
	1	118	17	54	4	107	26	137	19	53	93	0	176
3	**0**	82	5	2	0	69	10	130	9	74	28	0	149
	1	89	3	9	0	79	35	121	11	55	38	0	143
4	**0**	52	0	0	0	44	7	83	12	39	7	0	112
	1	49	0	1	0	58	32	83	16	42	22	0	114
0	**0**	134	49	148	12	147	4	163	18	43	155	3	197
	1	139	47	103	37	116	54	139	39	1	199	1	199

Tabelle 4: Auswertung Testgruppe Rome

Vergleichen wir zunächst die Werte σ_{dist}, σ_{disp} und cn_{10} zwischen den beiden Algorithmen. Bei einer Skalierung von $1b$ weisen die Layouts von GTree in 72.5% der Fälle einen besseren σ_{dist}-Wert auf. Bereits ab einer Skalierung von $2b$ jedoch erzeugt PRISM in 55% ein besseres Ergebnis als GTree (0.09%). Die Werte von σ_{disp} und cn_{10} verhalten sich ähnlich: Bei σ_{disp} hat GTree bei einer Skalierung von $1b$ in 57% der Fälle niedrigere Werte, bei $2b$ nur noch in 1% der Fälle. Ähnliches ergibt sich bei cn_{10}: Hier liefert bei einer Skalierung von $1b$ GTree in 58.75% der Fälle niedrigere Werte. Bei einer Skalierung von $2b$ nur noch 0.07%. Insgesamt sind die σ_{dist}, σ_{disp} und cn_{10}-Werte eher klein. Das spricht dafür, dass beide Algorithmen bezogen auf die Messwerte „gute" Ergebnisse liefern. Tabelle 5 zeigt die Mittelwerte von σ_{dist}, σ_{disp} und cn_{10} in den jeweiligen Gruppierungen.

		σ_{dist}		σ_{disp}		cn_{10}	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	0.35	0.31	0.03	0.03	2.52	2.24
	1	0.40	0.37	0.08	0.05	4.28	3.37
2	**0**	0.17	0.17	0.00	0.01	0.61	0.97
	1	0.17	0.18	0.01	0.01	0.77	1.11
3	**0**	0.07	0.07	0.00	0.00	0.19	0.32
	1	0.07	0.08	0.00	0.00	0.21	0.34
4	**0**	0.03	0.03	0.00	0.00	0.09	0.13
	1	0.03	0.03	0.00	0.00	0.10	0.14
0	**0**	0.29	0.31	0.01	0.03	1.35	2.24
	1	0.33	0.37	0.04	0.05	2.40	3.37

Tabelle 5: Mittelwerte von σ_{dist}, σ_{disp} und cn_{10} der Testgruppe Rome

Hypothese 1 kann anhand der Tabelle 4 für die Testgruppe Rome verifiziert werden: Bei 82.95 % der Graphen braucht GTree weniger Iterationen als PRISM. Während PRISM im Durchschnitt 11.44 Iterationen in der ersten Schleife benötigt, braucht GTree lediglich 2.90. Auch Hypothese 2 kann bestätigt werden: PRISM verändert das Seitenverhältnis bei kurzen Boxen (LB = 0) im Durchschnitt um 0.1, GTree lediglich um 0.06. Bei LB = 1, also längeren Boxen, verändert PRISM das Seitenverhältnis um 0.18, GTree nur um

0.09. Für Hypothese 3 sprechen einerseits die Layouts aus Abbildung 5.4, andererseits der Platzbedarf, der beim Algorithmus GTree bei den Skalierungen $1b$ bis $4b$ in 51% aller Vergleiche mehr Platz benötigt als PRISM. Bei einer Skalierung von $1b$ benötigt GTree in 84% aller Fälle mehr Platz. Dabei braucht GTree im Schnitt 5.5% mehr Platz. Bezüglich Hypothese 4 können wir feststellen, dass PRISM (im Sinne unserer Messwerte) bessere Layouts erzeugt: Skalieren wir das Initial-Layout auf das 0.85-fache der Fläche, die GTree benötigt hat, so erzeugt PRISM in 68.3% aller Fälle einen besseren σ_{dist}-Wert, in 62.8% einen besseren σ_{disp}-Wert und in 75.5% einen besseren cn_{10} - Wert.

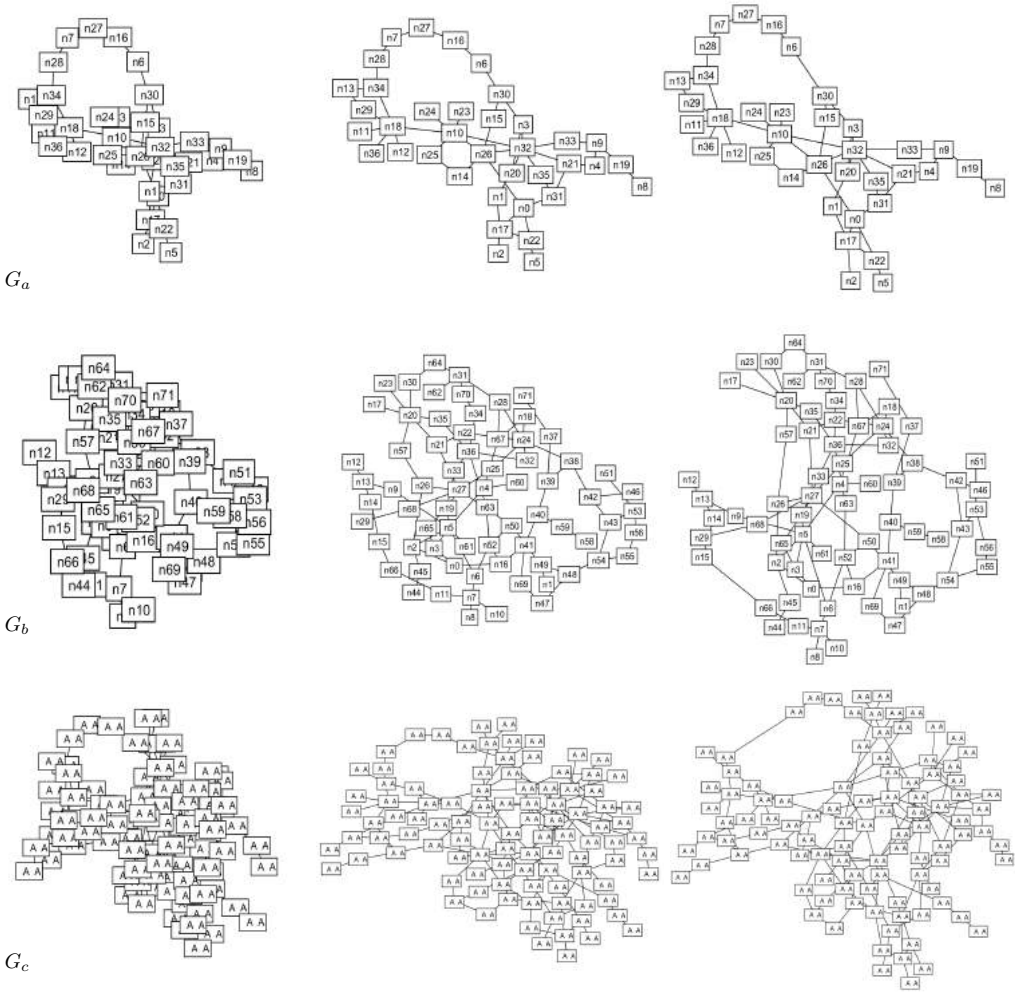


Abbildung 5.4: Graphen der Testgruppe Rome. Die linke Spalte zeigt das Initial-Layout. Die mittlere das durch PRISM, die rechte das durch GTree berechnete Layout.

			σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**	
	$\|V\|$	**SV**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
G_a	37	1.05	0.29	0.24	0.02	0.02	0.11	0.14	1.38	1.27	1.31	1.27
G_b	72	1.01	0.41	0.37	0.02	0.02	0.15	0.20	2.86	3.29	1.23	1.09
G_c	100	1.17	0.47	0.39	0.03	0.03	0.24	0.34	3.27	3.10	1.48	1.25

Tabelle 6: Vergleichsdaten der Graphen aus Abbildung 5.4

5.3.2 Testgruppe DAGmar

		σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**		**Iter**	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	5	191	4	184	198	0	22	178	3	197	19	181
	1	24	170	4	195	198	0	7	193	1	199	6	194
2	**0**	30	149	38	101	187	0	152	45	22	175	16	182
	1	61	122	18	154	185	8	98	99	13	183	6	193
3	**0**	86	76	64	13	176	0	186	4	39	143	11	179
	1	109	62	31	87	164	23	184	6	27	156	3	189
4	**0**	124	33	55	0	157	9	181	2	55	100	1	183
	1	135	23	39	26	157	24	178	5	47	115	0	189
0	**0**	135	48	177	2	145	36	192	7	44	153	25	173
	1	133	48	68	100	60	129	183	16	0	200	8	192

Tabelle 7: Auswertung Testgruppe DAGmar

Bei einer Skalierung von 1b liefert GTree erheblich bessere Ergebnisse als PRISM: bei 90.3% aller getesteten Graphen einen niedrigeren σ_{dist}-Wert, in 94.8% einen niedrigeren σ_{disp}-Wert und in 92.8% einen niedrigeren cn_{10}-Wert. Auch bei einer Skalierung von 2b liefert GTree bei einem Großteil der Testgraphen bessere Ergebnisse bei den Werten σ_{dist} (67.8%) und σ_{disp} (63.8%). Bei cn_{10} liefert PRISM bereits in 62.5% bessere Werte. Ab einer Skalierung von 4b erzeugt PRISM bei den Werten σ_{dist} und cn_{10} für einen Großteil der getesteten Graphen bessere Werte (64.8% und 89.8%). Tabelle 8 zeigt die Mittelwerte für σ_{dist}, σ_{disp} und cn_{10} für die Testgruppe DAGmar. Es ist zu sehen, dass die Werte höher sind als bei Rome. Vor allem bei den längeren Labelboxen bei den Skalierungen 1b und 2b erzeugt PRISM deutlich höhere Werte als GTree. Besonders wird dies deutlich bei dem Wert σ_{disp}, der bei PRISM bis zu dreimal höher ist als bei GTree.

		σ_{dist}		σ_{disp}		cn_{10}	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	0.53	0.47	0.07	0.04	5.96	4.91
	1	0.62	0.54	0.21	0.07	13.40	7.42
2	**0**	0.46	0.43	0.03	0.03	3.72	4.25
	1	0.51	0.48	0.09	0.04	6.09	5.88
3	**0**	0.36	0.36	0.01	0.01	1.92	3.10
	1	0.37	0.38	0.03	0.02	2.56	3.86
4	**0**	0.27	0.28	0.00	0.01	0.95	1.98
	1	0.27	0.29	0.01	0.01	1.16	2.18
0	**0**	0.45	0.47	0.02	0.04	2.82	4.91
	1	0.51	0.54	0.07	0.07	5.10	7.42

Tabelle 8: Mittelwerte von σ_{dist}, σ_{disp} und cn_{10} der Testgruppe DAGmar

Hypothese 1 kann auch für die Testgruppe DAGmar bestätigt werden: GTree benötigt in 92.8% aller Fälle weniger Iterationen als PRISM. Im Durchschnitt sind dies bei GTree 10.16, bei PRISM 26.34. Auch Hypothese 2 kann verifiziert werden: Während PRISM das Seitenverhältnis für LB = 0 im Durchschnitt um 0.2 ändert, liegt die Änderung bei GTree nur bei 0.07. Für LB = 1 ändert PRISM das Seitenverhältnis im Mittel um 0.59, GTree lediglich um 0.15. Für Hypothese 3 sprechen analog zur Testgruppe Rome die Abbildungen 5.6 und 5.7 und der Platzbedarf, der bei GTree in 88.9% der Fälle höher ist. Im Durchschnitt benötigt GTree 13% mehr Platz. Skalieren wir das Initial-Layout auf das 0.85-fache des Platzbedarfes von GTree hoch und berechnen dann ein überlappungsfreies Layout mittels PRISM, so werden in 67% der Testfälle bessere σ_{dist}-Werte erzeugt und in 93.8% bessere cn_{10}-Werte. Dies gilt für LB = 0 und LB = 1 insgesamt. Lediglich beim Wert σ_{disp} gibt es einen Unterschied: Während bei den kurzen Boxen-Label (LB = 0) Prism besser abschneidet (in 88.5% der Fälle), erzeugt GTree bei LB = 1 beim größeren Teil (50%) bessere Werte. Hypothese 4 kann also teilweise verifiziert werden.

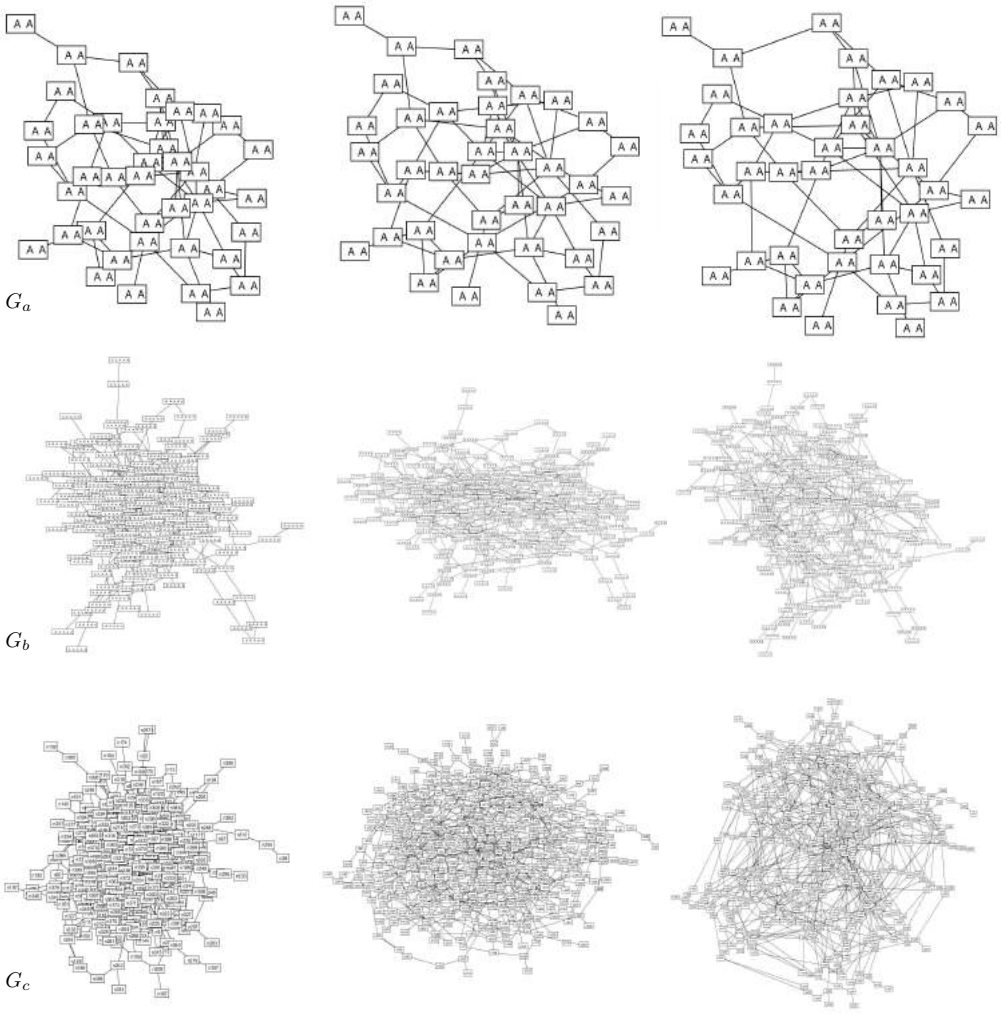


Abbildung 5.5: Graphen der Testgruppe DAGmar. Die linke Spalte zeigt das Initial-Layout. Die mittlere das durch PRISM, die rechte das durch GTree berechnete Layout.

			σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**	
	$\|V\|$	**SV**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
G_a	40	0.96	0.19	0.21	0.01	0.02	0.13	0.14	0.72	1.88	1.06	1.14
G_b	240	0.92	0.52	0.46	0.15	0.05	2.75	2.31	8.32	7.54	1.63	1.15
G_c	400	1.15	0.85	0.76	0.07	0.04	1.30	1.90	6.67	6.73	1.52	1.13

Tabelle 9: Vergleichsdaten der Graphen aus Abbildung 5.5

5.3.3 Testgruppe BarabasiAlbert

In der Testgruppe BarabasiAlbert, die Graphen der Größe $550 \leq |V| \leq 1000$ enthält, lassen sich folgende Ergebnisse bezüglich der Werte σ_{dist}, σ_{disp} und cn_{10} festhalten. Bei einer Skalierung von $1b$ liefert GTree in allen Fällen einen besseren σ_{dist}-Wert und in 99%

		σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**		**Iter**	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	0	200	1	196	200	0	7	193	0	200	53	146
	1	0	200	0	200	200	0	0	200	0	200	18	180
2	**0**	1	199	43	46	200	0	199	1	3	195	45	152
	1	1	198	0	199	199	1	136	62	0	200	11	188
3	**0**	15	156	104	3	199	1	200	0	19	177	23	176
	1	10	173	14	72	186	13	195	5	4	193	8	192
4	**0**	93	58	38	0	200	0	200	0	35	140	20	180
	1	72	65	35	24	175	25	200	0	22	162	4	196
0	**0**	160	28	165	0	133	58	200	0	65	133	71	129
	1	94	75	31	130	9	190	200	0	1	199	31	168

Tabelle 10: Auswertung Testgruppe BarabasiAlbert

		σ_{dist}		σ_{disp}		cn_{10}	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	0.74	0.63	0.05	0.02	8.63	7.16
	1	0.81	0.68	0.18	0.03	14.77	10.20
2	**0**	0.67	0.60	0.02	0.02	4.48	5.93
	1	0.72	0.64	0.06	0.02	6.44	6.87
3	**0**	0.55	0.52	0.00	0.01	2.19	3.76
	1	0.56	0.53	0.01	0.01	2.90	3.78
4	**0**	0.42	0.42	0.00	0.00	1.12	2.10
	1	0.42	0.41	0.00	0.00	1.37	2.06
0	**0**	0.61	0.63	0.02	0.02	2.98	7.16
	1	0.68	0.68	0.03	0.03	5.33	10.20

Tabelle 11: Mittelwerte von σ_{dist}, σ_{disp} und cn_{10} der Testgruppe BarabasiAlbert

einen besseren σ_{disp}-Wert. Auch bezüglich des Wertes cn_{10} ist das Ergebnis eindeutig: In 98.3% ist hier GTree besser. Dieses Verhältnis bleibt bei einer Skalierung von $2b$ bei σ_{dist} und σ_{disp} in etwa bestehen. Lediglich bei cn_{10} liefert PRISM in 83.8% der Fälle niedrigere Werte. Erst ab einer Skalierung von $4b$ erzeugt PRISM leicht bessere Ergebnisse bei σ_{dist} und σ_{disp}.

Auch bei den Mittelwerten aus Tabelle 11 sind erhebliche Unterschiede zwischen PRISM und GTree erkennbar. Vor allem bei LB = 1 unterscheiden sich die einzelnen Werte stark: Der σ_{disp}-Wert bei PRISM ist bei einer Skalierung von $1b$ sechsmal höher als der bei GTree.

Bei der Testgruppe BarabasiAlbert kann Hypothese 1 analog zu den bisher verglichenen Testgruppen - wenn auch nicht so deutlich - verifiziert werden: In 85.4% der Testfälle benötigt GTree (im Durchschnitt 18.69) weniger Iterationen als PRISM (30.39). Bezüglich Hypothese 2 lässt sich feststellen, dass GTree in 90% der Fälle eine geringere Verzerrung aufweist. Bei LB = 0 verändert PRISM das Seitenverhältnis im Schnitt um 0.21. GTree ändert das Seitenverhältnis lediglich um 0.06. Auch in dieser Testgruppe erhöht sich diese Veränderung bei LB = 1. Während PRISM das Verhältnis um 0.57 ändert, weisen die durch GTree erzeugten Layouts im Durchschnitt eine geringere Verzerrung (0.09) auf.

Somit kann auch in dieser Gruppe Hypothese 2 bestätigt werden.
Betrachten wir nun Hypothese 3. Der Platzbedarf von GTree ist auch hier höher als bei PRISM. Wie Tabelle 8 zeigt, benötigt GTree bei den Skalierungen $1b$ bis $4b$ in 97.5% der Fälle mehr Platz. Der Platzbedarf ist hierbei im Mittel um 13.46% größer als bei PRISM. Auch die Layouts aus Abbildung 5.6 deuten an, dass PRISM eher kompaktere Layouts erzeugt, wenn auch mit einem abweichenden Seitenverhältnis. Wir gehen nun zuletzt noch auf die vierte Hypothese ein. Hier ähneln die Ergebnisse denen aus der Testgruppe DAGmar. Während PRISM in 63.4% der Fälle bessere σ_{dist}-Werte und in 100% der Fälle bessere cn_{10}-Werte erzeugt, wenn das Initial-Layout auf das 0.85-fache skaliert wird, ergeben sich bei σ_{dist} anders gelagerte Werte: Im Falle von LB = 0 ist PRISM in 82.5% der Testfälle besser. Bei LB = 1 jedoch ist GTree in 65% der Fälle besser. Hypothese 4 kann also auch hier nur teilweise verifiziert werden.

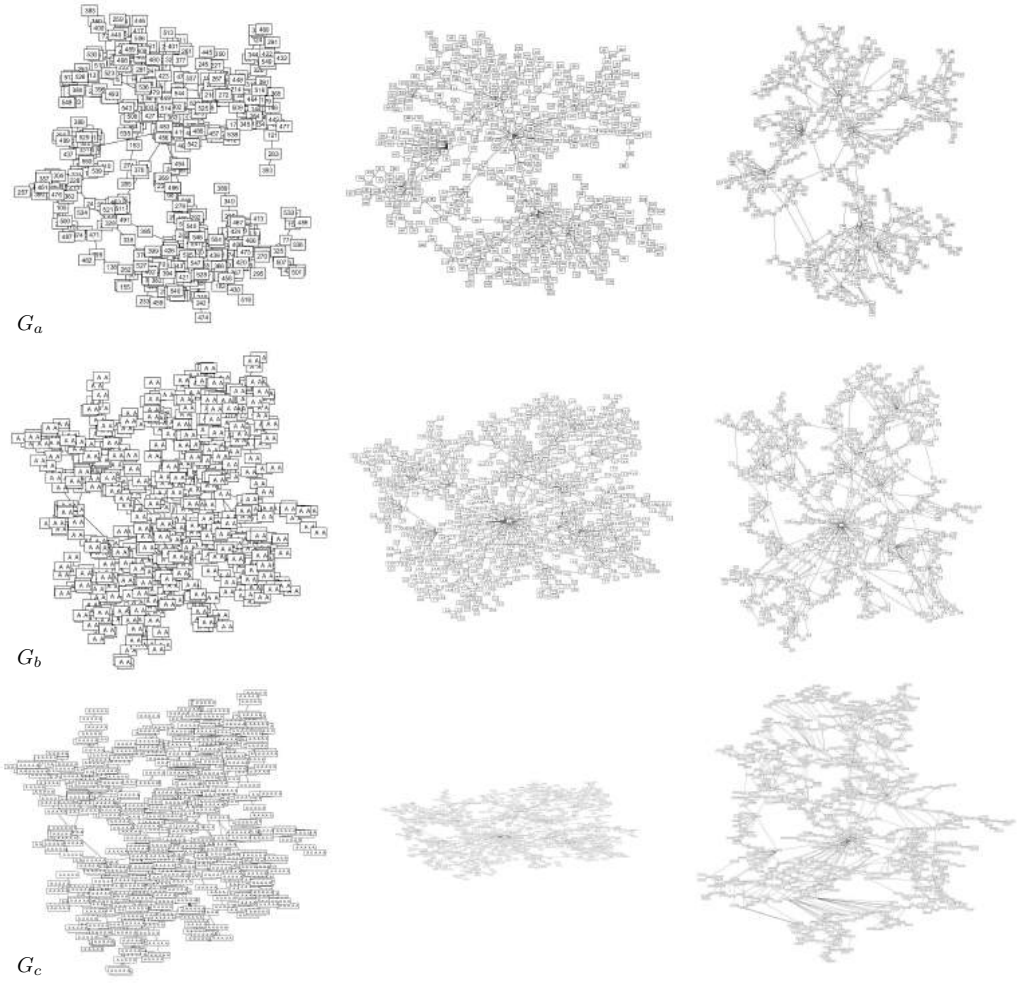


Abbildung 5.6: Graphen der Testgruppe BarabasiAlbert. Die linke Spalte zeigt das Initial-Layout. Die mittlere das durch PRISM, die rechte das durch GTree berechnete Layout.

			σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**	
	$\|V\|$	**SV**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
G_a	550	0.97	0.62	0.52	0.03	0.01	1.27	2.09	6.38	5.96	1.27	0.90
G_b	750	1.07	0.64	0.51	0.06	0.01	2.04	3.45	8.39	6.35	1.51	0.97
G_c	750	1.11	0.73	0.66	0.20	0.06	5.12	8.16	14.47	10.59	2.62	1.14

Tabelle 12: Vergleichsdaten der Graphen aus Abbildung 5.6

5.3.4 Testgruppe SmallWorld

		σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**		**Iter**	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	0	200	0	200	200	0	24	175	0	200	43	152
	1	2	197	0	200	200	0	2	198	0	200	20	180
2	**0**	1	195	23	89	200	0	200	0	2	197	43	157
	1	19	175	2	188	199	1	200	0	0	199	7	193
3	**0**	16	162	113	0	200	0	200	0	31	157	7	192
	1	45	124	32	39	198	1	200	0	14	179	3	197
4	**0**	47	103	22	0	200	0	200	0	76	91	1	198
	1	70	76	38	0	183	17	200	0	59	116	0	200
0	**0**	128	60	150	0	200	0	200	0	46	152	74	120
	1	127	64	68	67	117	79	200	0	0	200	35	165

Tabelle 13: Auswertung Testgruppe SmallWorld

Die Ergebnisse der Testgruppe SmallWorld ähneln den Ergebnissen der Gruppe BarabasiAlbert: Bei einer Skalierung von $1b$ liefert GTree in nahezu allen Fällen (99.3% bei σ_{dist}, 100% bei σ_{disp} und cn_{10}) bessere Werte. Ab einer Skalierung von $2b$ produziert PRISM in 100% aller Testfälle einen besseren cn_{10}-Wert. Bei einer Skalierung von $3b$ haben die von PRISM produzierten Layouts in 56.5% der Fälle einen besseren σ_{disp}-Wert, wenn die Label-Boxen kurz sind. Für LB $= 1$ gilt dies nicht mehr, hier weist PRISM nur noch in 16% der Fälle einen geringeren σ_{disp}-Wert als GTree auf.

		σ_{dist}		σ_{disp}		cn_{10}	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	0.95	0.81	0.06	0.02	6.86	6.11
	1	0.98	0.83	0.19	0.03	13.77	9.62
2	**0**	0.89	0.79	0.02	0.01	3.39	5.59
	1	0.89	0.79	0.05	0.02	4.70	6.97
3	**0**	0.75	0.71	0.00	0.01	1.42	3.45
	1	0.69	0.67	0.01	0.01	1.70	3.49
4	**0**	0.60	0.59	0.00	0.00	0.66	1.81
	1	0.53	0.52	0.00	0.00	0.71	1.68
0	**0**	0.80	0.81	0.01	0.02	1.81	6.11
	1	0.84	0.83	0.03	0.03	3.28	9.62

Tabelle 14: Mittelwerte von σ_{dist}, σ_{disp} und cn_{10} der Testgruppe SmallWorld

Tabelle 14 zeigt die Mittelwerte der Messwerte σ_{dist}, σ_{disp} und cn_{10} in der Testgruppe SmallWorld. Analog zur Gruppe BarabasiAlbert lässt sich auch hier ablesen, dass der σ_{disp}-Wert vor allem bei einer Skalierung von $1b$ und LB = 1 deutlich höher ist als bei GTree.

Abschließend gehen wir auf die Hypothesen ein: Da GTree in 87.7% der Fälle weniger Iterationen, im Durchschnitt 18.38, als PRISM (29.38) benötigt, können wir Hypothese 1 verifizieren. Bezüglich der Verzerrung des Seitenverhältnisses liefert GTree in 84.6% der Fälle eine geringere Abweichung. Bei LB = 0 verändert PRISM hierbei das Seitenverhältnis im Durchschnitt um 0.21, GTree dagegen um 0.06. Auch hier wird bei LB = 1 die Verzerrung größer. PRISM verzerrt das Seitenverhältnis im Schnitt um 0.59, GTree dagegen nur um 0.14. Wir können also auch in der Testgruppe SmallWorld die zweite Hypothese bestätigen.

Mit Blick auf den Platzbedarf der beiden Algorithmen, der ähnlich zu der Gruppe BarabasiAlbert ist, und mit Blick auf die Abbildung 5.7 können wir auch Hypothese 3 verifizieren: Bei einer Skalierung von $1b$ bis $4b$ benötigt GTree in 98.8% aller Testfälle mehr Platz als PRISM. Der Platzbedarf ist hierbei im Mittel um 16.6% größer als bei PRISM. Im Gegensatz zu den Testgruppen DAGmar und BarabasiAlbert können wir Hypothese 4 wieder im Ganzen bestätigen: Skalieren wir das Initial-Layout um das 0.85-fache des Platzbedarfs von GTree produziert PRISM zu größeren Teilen bessere Ergebnisse. So werden in 63.8% bessere σ_{dist}-Werte und in 54.5% der Fälle bessere σ_{disp}-Werte erzeugt.

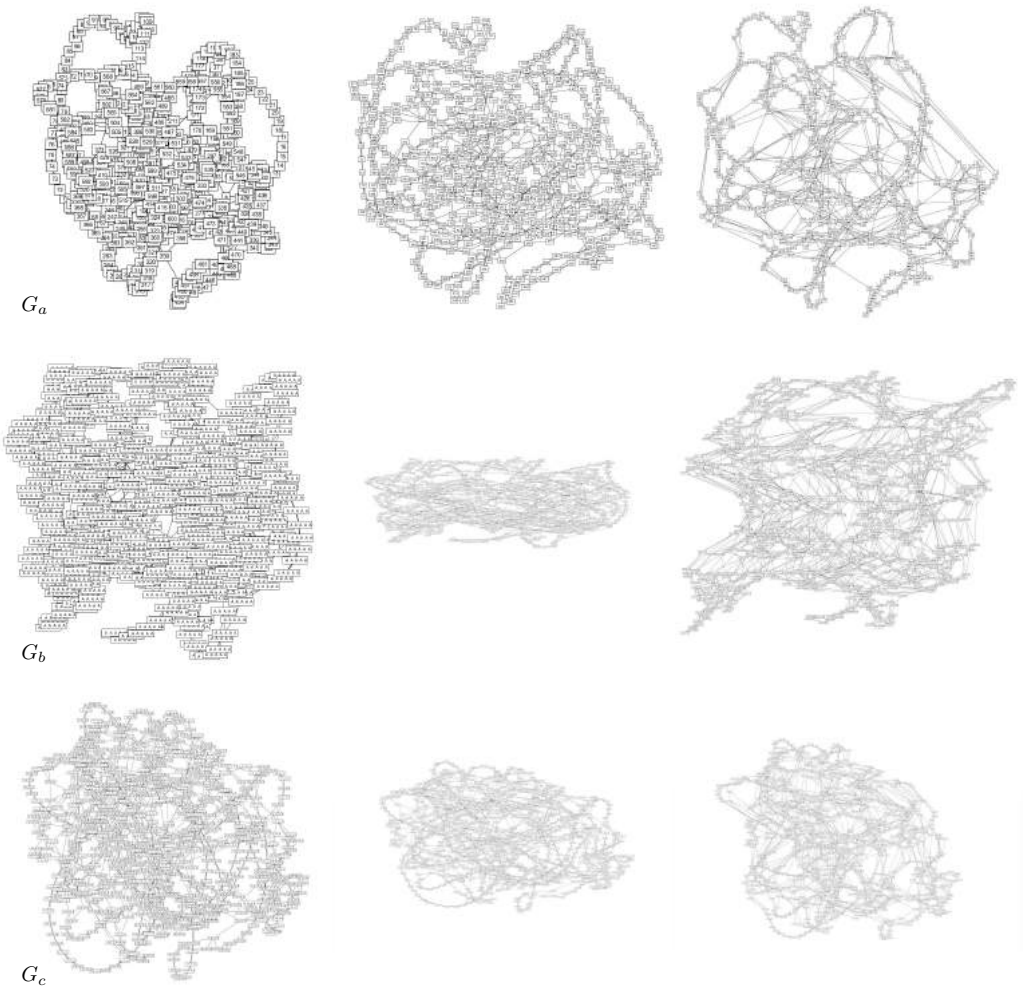


Abbildung 5.7: Graphen der Testgruppe SmallWorld. Die linke Spalte zeigt das Initial-Layout. Die mittlere das durch PRISM, die rechte das durch GTree berechnete Layout.

			σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**	
	$\|V\|$	**SV**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
G_a	600	0.92	1.72	1.49	0.03	0.03	1.36	2.72	5.03	5.77	1.15	1.05
G_b	800	1.11	0.87	0.75	0.19	0.04	4.37	6.95	14.63	11.65	3.08	1.36
G_c	900	1.10	0.71	0.62	0.06	0.01	6.99	8.81	5.87	7.55	1.79	1.28

Tabelle 15: Vergleichsdaten der Graphen aus Abbildung 5.7

5.3.5 Zusammenfassung

Bevor wir unsere Ergebnisse zusammenfassen, stellen wir abschließend unsere gesamte Auswertung in den folgenden Tabellen dar.

		σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**		**Iter**	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	28	753	49	673	767	1	121	652	29	766	117	674
	1	84	692	33	727	762	10	57	719	22	776	45	753
2	**0**	136	565	166	236	714	7	703	55	112	634	104	664
	1	199	512	74	545	692	34	572	179	66	675	24	750
3	**0**	199	399	283	19	644	11	716	13	193	505	41	696
	1	253	362	86	198	627	72	700	22	100	566	14	721
4	**0**	316	194	115	0	601	16	664	14	205	338	22	673
	1	326	164	113	50	573	98	661	21	170	415	4	699
0	**0**	555	184	637	14	623	98	755	22	197	591	172	617
	1	493	234	270	334	302	452	724	53	2	798	75	724

Tabelle 16: Auswertung der gesamten Stichprobe

		σ_{dist}		σ_{disp}		cn_{10}	
b	**LB**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
1	**0**	0.64	0.55	0.05	0.03	5.99	5.10
	1	0.70	0.61	0.16	0.04	11.56	7.65
2	**0**	0.55	0.50	0.02	0.02	3.05	4.18
	1	0.57	0.52	0.05	0.02	4.50	5.21
3	**0**	0.43	0.42	0.00	0.01	1.43	2.66
	1	0.43	0.41	0.01	0.01	1.84	2.87
4	**0**	0.33	0.33	0.00	0.00	0.71	1.51
	1	0.31	0.31	0.00	0.00	0.83	1.52
0	**0**	0.54	0.55	0.01	0.03	2.24	5.10
	1	0.59	0.61	0.04	0.04	4.03	7.65

Tabelle 17: Mittelwerte von σ_{dist}, σ_{disp} und cn_{10} der gesamten Stichprobe

Mit Blick auf die Iterationen können wir festhalten, dass GTree in 87.1% unserer 8000 Testfälle weniger Iterationen als PRISM benötigt, um ein überlappungsfreies Layout zu erzeugen. Dafür benötigt GTree aber auch in 89.7% aller Fälle mehr Platz. Betrachten wir vor allem eine Skalierung von $1b$ erzeugt GTree bei 90% der Testgraphen bessere σ_{dist}-Werte und bei 87.5% bessere σ_{disp}-Werte. Im Mittel sind die Messwerte σ_{dist} und σ_{disp} bei den Skalierungen $1b$ bis $3b$ stets niedriger bei GTree, siehe Tabelle 17. Bei einer Skalierung von $4b$ gibt es keine Unterschiede der Mittelwerte von σ_{dist} und σ_{disp} zwischen PRISM und GTree. Ein anderes Verhältnis können wir bei den cn_{10}-Werten ablesen: Ab einer Skalierung von $2b$ erzeugt hier PRISM im Mittel bessere Werte.
Wir fassen unsere Ergebnisse hinisichtlich der in Abschnitt 5.1 formulierten Hypothesen abschließend zusammen: Hypothese 1, die die Vemutung aufgestellt hat, dass GTree in der Regel weniger Iterationen als PRISM benötigt, kann für unsere Stichprobe verifiziert

werden. Auch Hypothese 2 kann bestätigt werden: PRISM ändert das Seitenverhältnis stärker als GTree. Diese Verzerrung nimmt mit der horizontalen Länge der Label-Boxen zu. Im Bezug zur Hypothese 3 können wir festhalten, dass GTree in den meisten Fällen mehr Platz braucht und PRISM eher kompaktere Layouts erzeugt.
Die Auswertung unserer Testgruppen hat ergeben, dass Hypothese 4 in Bezug zu den Werten σ_{dist} und cn_{10} verifiziert werden kann: Skalieren wir das Initial-Layout auf das 0.85-fache der Fläche, die GTree genutzt hat, um eine Platzierung disjunkter Label-Boxen zu erreichen, erzeugt PRISM bessere Ergebnisse bei diesen Werten. Bei dem Messwert σ_{disp} konnte keine klare Aussage getroffen werden. Da sich in unseren Vergleichen und dem Experimentieren in OverlapR herausgestellt hat, dass Hypothese 4 in vielen Fällen bestätigt werden kann, stellen wir zum Schluss dieses Abschnittes noch sechs zufällig gewählte Graphen gesondert vor, bei denen wir das Initial-Layout für die Berechnung von PRISM auf das 0.85-fache des Platzbedarfes von GTree skalieren. Dabei verwenden wir kurze und lange Label-Boxen gleichermaßen. Die darauffolgende Tabelle enthält die Vergleichswerte der Graphen.

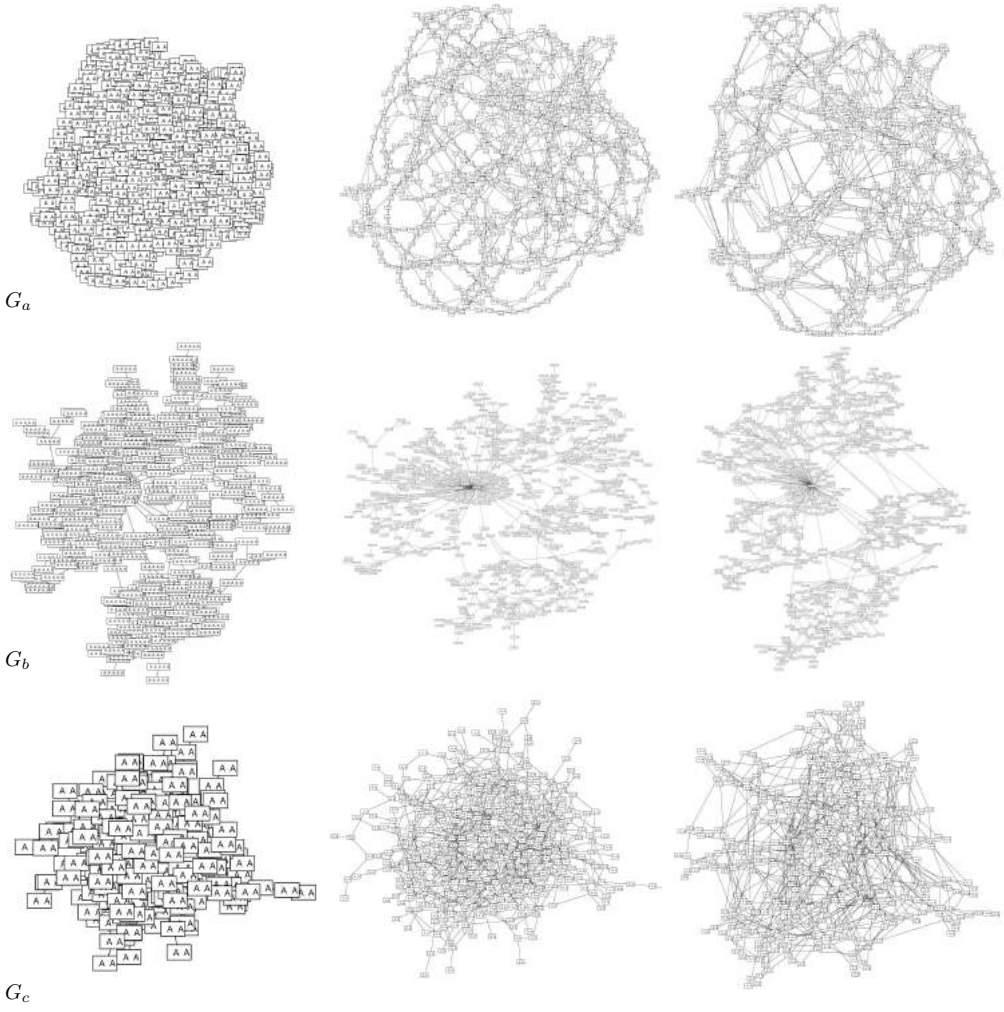


Abbildung 5.8: Visualisierung Hypothese 4 - Teil 1. Die linke Spalte zeigt das Initial-Layout. Die mittlere das durch PRISM, die rechte das durch GTree berechnete Layout.

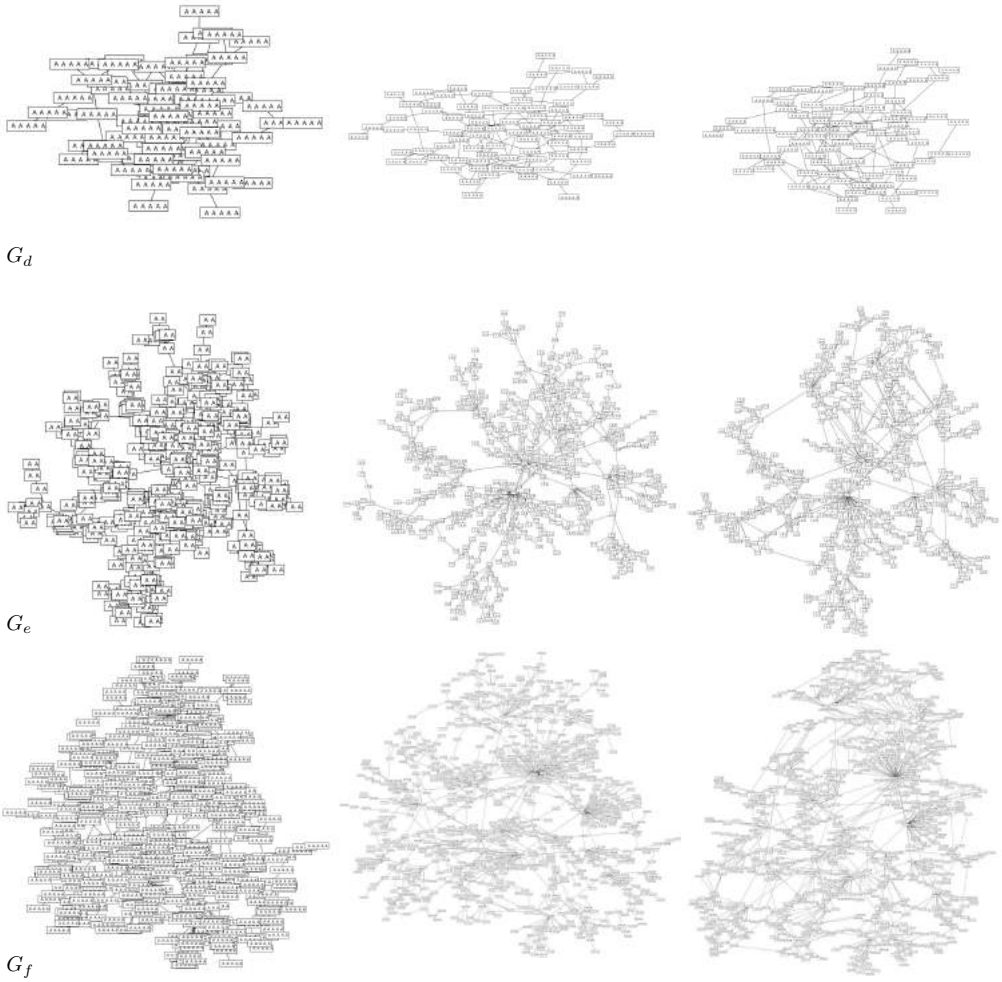


Abbildung 5.9: Visualisierung Hypothese 4 - Teil 2. Die linke Spalte zeigt das Initial-Layout. Die mittlere das durch PRISM, die rechte das durch GTree berechnete Layout.

			σ_{dist}		σ_{disp}		$area$		cn_{10}		**SV**	
	$\|V\|$	**SV**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**	**PR**	**GT**
G_a	750	1.06	0.57	0.59	0.00	0.01	2.84	3.18	1.63	5.09	1.10	1.01
G_b	650	0.89	0.63	0.64	0.04	0.04	7.01	6.87	4.37	8.38	1.18	0.92
G_c	400	1.29	0.50	0.53	0.03	0.04	1.88	1.89	3.96	6.30	1.30	1.14
G_d	84	1.55	0.49	0.54	0.06	0.06	0.67	0.66	5.20	6.06	2.19	1.67
G_e	550	1.00	0.54	0.54	0.01	0.02	2.97	2.93	3.10	5.92	1.12	0.99
G_f	850	1.11	0.67	0.63	0.07	0.03	8.62	7.81	7.01	11.20	1.37	1.02

Abbildung 5.10: Vergleichsdaten der Graphen aus Abbildungen 5.8 und 5.9

6 Fazit und Ausblick

In der vorliegenden Masterthesis mit dem Titel „Vergleich von Algorithmen zur Platzierung disjunkter Boxen" haben wir mit dem von uns implementierten Programm OverlapR die beiden Algorithmen PRISM und GTree detailliert beschrieben und anhand von 8000 Graphenlayouts verglichen. Ein zentrales Ergebnis ist, dass PRISM im Großteil der Fälle bessere Ergebnisse in Hinblick auf die Abstände und die Verschiebung von Knoten erzeugt, wenn die Platzausnutzung von GTree und PRISM ungefähr gleich groß ist. Im direkten Vergleich jedoch verändert PRISM das Seitenverhältnis wesentlich stärker als GTree. Diese Verzerrung vergrößert sich bei PRISM mit zunehmender Länge der Label-Boxen. Betrachten wir hingegen die Nachbarknoten der einzelnen Knoten, so ergibt sich bei PRISM eine geringere Verzerrung als bei GTree.
Aus diesen Gründen und auf Grundlage der Ergebnisse aus Abschnitt 5.3 formulieren wir folgende Empfehlungen, die immer abhängig sind von den Anforderungen an das Layout:

Empfehlung 1 Soll das Seitenverhältnis möglichst wenig verändert werden, empfehlen wir GTree.

Empfehlung 2 Ist eine geringe Laufzeit nötig, empfehlen wir GTree.

Empfehlung 3 Sollen die Knoten innerhalb ihrer „Knoten-Nachbarschaft" bleiben, empfehlen wir PRISM.

Empfehlung 4 Spielt die Platzausnutzung eine wichtige Rolle, empfehlen wir PRISM.

Empfehlung 5 Kommen im Layout eher längere Label-Boxen vor, empfehlen wir GTree.

Im Verlauf dieser Arbeit und der Auswertung des Vergleiches zwischen PRISM und GTree hat sich herausgestellt, dass das Format der Label-Boxen eine große Rolle spielt bei der Frage danach, in welchen Fällen die Algorithmen bessere Ergebnisse liefern. Hier könnten weitergehende Untersuchungen, die diesen Aspekt priorisieren, gewinnbringend sein. So könnte eventuell das Initial-Layout in seinen x- und y-Dimensionen bei PRISM in Abhängigkeit zum Label-Boxen-Format skaliert werden, um auch gute Ergebnisse bei langen Label-Boxen zu erzeugen. Auch GTree könnte in diesem Aspekt eingehender untersucht werden.
Das Messen von Gleichheit bzw. Unterschiedlichkeit von Graphenlayouts ist keine triviale Aufgabe. Wir folgen den Autoren von PRISM [11] und betrachten den Wert σ_{disp} als aussagekräftiger als σ_{dist}. Dies müsste jedoch durch einen Vergleich verifiziert bzw. falsifiziert werden. Vielleicht lässt sich dies aber auch nicht abschließend bewerten, da das Beurteilen von Graphenlayouts häufig ein eher subjektives Unterfangen ist. Uns haben im Großen und Ganzen die von PRISM produzierten Layouts „besser gefallen" als die von GTree produzierten Layouts. Dies liegt vor allem daran, dass GTree dazu neigt, stark verzerrte Layouts zu erstellen, sodass relativ große Freiflächen im Layout entstehen (siehe Abbildungen 5.8 und 5.9).

Abschließend gehen wir kurz auf die Implementierung beider Algorithmen und auf OverlapR ein. Wir empfanden die Implementierung von GTree als einfacher, weil hier im Wesentlichen nur drei Algorithmen zum Tragen kommen: Das Erstellen einer Delaunay-Triangulierung, der Scanline-Algorithmus und das Aufbauen eines minimal aufspannenden Baumes. Letzterer kann durch Prim's Algorithmus relativ leicht implementiert werden. Dagegen benutzt PRISM die Stress Majorization, die wiederum das CG-Verfahren anwendet. Beide Algorithmen sind in unseren Augen nicht so leicht zu implementieren, weil diverse Matrizen aufgebaut und berechnet werden müssen und es einige Toleranzwerte gibt, deren Veränderung die erstellten Layouts erheblich voneinander abweichen lassen können.
Das Programm OverlapR erzeugt aus Graphenlayouts überlappungsfreie Layouts. Dabei ist es in unseren Vergleichen nicht vorgekommen, dass Überlappungen nicht entfernt wurden. Durch die offene, objektorientierte Programmierung sollte es keine Schwierigkeiten machen, das Programm mit anderen Algorithmen zur Platzierung disjunkter Boxen zu erweitern - ein Anpassen der Benutzeroberfläche vorausgesetzt.

Literatur

[1] Albert, R. ; Barabási, A.-L. : Statistical mechanics of complex networks. In: *Reviews of Modern Physics* 74 (2002), Nr. 1, S. 47–97

[2] Asimow, L. ; Roth, B. : The rigidity of graphs. In: *Transactions of the American Mathematical Society* 245 (1978), S. 279–279

[3] Bachmaier, C. ; Gleissner, A. ; Hofmeier, A. : *DAGmar: Library for DAGs.* Technical Report, Number MIP-1202 Department of Informatics and Mathematics University of Passau, 2012

[4] Bennett, C. ; Ryall, J. ; Spalteholz, L. ; Gooch, A. : The Aesthetics of Graph Visualization. In: *Proceedings of the Third Eurographics Conference on Computational Aesthetics in Graphics, Visualization and Imaging.* Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2007 (Computational Aesthetics'07), S. 57–64

[5] Cox, M. A. A. ; Cox, T. F.: Multidimensional Scaling. In: Chen, C. (Hrsg.) ; Haerdle, W. K. (Hrsg.) ; Unwin, A. (Hrsg.): *Handbook of Data Visualization.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, S. 315–347

[6] Dehmer, M. ; Emmert-Streib, F. ; Kilian, J. : A similarity measure for graphs with low computational complexity. In: *Applied Mathematics and Computation* 182 (2006), Nr. 1, S. 447–459

[7] Dwyer, T. ; Koren, Y. ; Marriott, K. : IPSep-CoLa: An Incremental Procedure for Separation Constraint Layout of Graphs. In: *IEEE Transactions on Visualization and Computer Graphics* 12 (2006), Nr. 5, S. 821–828

[8] Dwyer, T. ; Marriott, K. ; Stuckey, P. J.: Fast Node Overlap Removal. In: Healy, P. (Hrsg.) ; Nikolov, N. S. (Hrsg.): *Graph Drawing: 13th International Symposium, GD 2005, Limerick, Ireland, September 12-14, 2005. Revised Papers.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, S. 153–164

[9] Emmert-Streib, F. ; Dehmer, M. ; Shi, Y. : Fifty years of graph matching, network alignment and network comparison. In: *Information Sciences* 346-347 (2016), S. 180–197

[10] Fruchterman, T. M. ; Reingold, E. : Graph Drawing by Force-directed Placement. In: *Softw. Pract. Exper.* 21 (1991), Nr. 11, S. 1129–1164

[11] Gansner, E. R. ; Hu, Y. : Efficient, Proximity-Preserving Node Overlap Removal. In: *Journal of Graph Algorithms and Applications* 14 (2010), Nr. 1, S. 53–74

[12] Gansner, E. R. ; Koren, Y. ; North, S. : Graph Drawing by Stress Majorization. In: Pach, J. (Hrsg.): *Graph Drawing: 12th International Symposium, GD 2004, New*

York, NY, USA, September 29-October 2, 2004, Revised Selected Papers. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005, S. 239–250

[13] Gansner, E. R. ; North, S. C.: An open graph visualization system and its applications to software engineering. In: *Software: Practice and Experience* 30 (2000), Nr. 11, S. 1203–1233

[14] Huppert, B. ; Willems, W. : Lineare Abbildungen und Matrizen. In: *Lineare Algebra*. Springer, 2010

[15] I., B. ; Groenen, P. : *Modern Multidimensional Scaling. Theory and Applications*. Springer, 2005

[16] Kelley, C. : *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, 1995 (Frontiers in Applied Mathematics)

[17] Kobourov, S. G.: Force-Directed Drawing Algorithms. In: Tamassia, R. (Hrsg.): *Graph Drawing: 12th International Symposium, GD 2004, New York, NY, USA, September 29-October 2, 2004, Revised Selected Papers*. CRC Press, 2013, S. 383–408

[18] Lee, D. T. ; Schachter, B. J.: Two algorithms for constructing a Delaunay triangulation. In: *International Journal of Computer & Information Sciences* 9 (1980), Nr. 3, S. 219

[19] Li, W. ; Eades, P. ; Nikolov, N. : Using Spring Algorithms to Remove Node Overlapping. In: *Proceedings of the 2005 Asia-Pacific Symposium on Information Visualisation - Volume 45*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2005 (APVis '05), S. 131–140

[20] Lischinski, D. : Incremental Delaunay Triangulation. In: Heckbert, P. S. (Hrsg.): *Graphics Gems IV*. San Diego, CA, USA : Academic Press Professional, Inc., 1994, S. 47–59

[21] Lyons, K. A. ; Meijer, H. ; Rappaport, D. : Algorithms for Cluster Busting in Anchored Graph Drawing. In: *Journal of Graph Algorithms and Applications* 2 (1998), Nr. 1, S. 1–24

[22] Nachmanson, L. ; Nocaj, A. ; Bereg, S. ; Zhang, L. ; Holroyd, A. E.: Node Overlap Removal by Growing a Tree. In: *CoRR* abs/1608.02653 (2016)

[23] Nobari, S. ; Lu, X. ; Karras, P. ; Bressan, S. : Fast Random Graph Generation. In: *Proceedings of the 14th International Conference on Extending Database Technology*. New York, NY, USA : ACM, 2011 (EDBT/ICDT '11), S. 331–342

[24] Turau, V. : *Algorithmische Graphentheorie*. 3. Oldenbourg, 2009

[25] Watts, D. J. ; Strogatz, S. H.: Collective dynamics of 'small-world' networks. In: *Nature* 393 (1998), Nr. 6684, S. 440–442

[26] Zager, L. A. ; Verghese, G. C.: Graph similarity scoring and matching. In: *Applied Mathematics Letters* 21 (2008), Nr. 1, S. 86–94

Anhang

Inhalt der beiliegenden CD-Rom:

- die Masterthesis im .pdf-Format
- das Programm OverlapR als ausführbare .jar-Datei
- Ordner `QuellCode` mit den Unterordnern
  - `Bibliotheken`: enthält alle externen Bibliotheken, die in OverlapR verwendet werden
  - `OverlapR`: enthält den Quellcode des Programmes OverlapR
- Ordner `Auswertung` mit den von OverlapR erzeugten .txt-Dateien sowie die Tabellen der Auswertung als .ods-Datei
- Ordner `Testgraphen` mit den Unterordnern
  - `BarabasiAlbert`: enthält die 200 Testgraphen der Gruppe BarabasiAlbert im .graphml-Format
  - `DAGmar`: enthält die 200 Testgraphen der Gruppe DAGmar im .graphml-Format
  - `Rome`: enthält die 200 Testgraphen der Gruppe Rome im .graphml-Format
  - `SmallWorld`: enthält die 200 Testgraphen der Gruppe SmallWorldt im .graphml-Format