Julius-Maximilians-Universität Würzburg
Institut für Informatik
Lehrstuhl für Informatik I
Effiziente Algorithmen und wissensbasierte Systeme

Bachelorarbeit

Beschleunigung von kräftebasierten Graphzeichenalgorithmen mittels wohlseparierten Paardekompositionen

Johannes Zink

Eingereicht am 04.08.2015

Betreuer:
Prof. Dr. Alexander Wolff
Fabian Lipp, M. Sc.

Zusammenfassung

In dieser Arbeit wird eine Möglichkeit vorgestellt, um die Laufzeitklasse eines klassischen kräftebasierten Graphzeichenalgorithmus wie des Algorithmus von Fruchterman und Reingold von $O(|V|^2)$ auf $O(|V| \log |V| + |E|)$ zu senken. Dabei ist $G = (V, E)$ der zu zeichnende Graph. Erreicht wird dies durch die Näherung der abstoßenden Kräfte, die im klassischen kräftebasierten Graphzeichenalgorithmus für jedes Paar von Knoten berechnet werden. Hier werden dafür die Paare einer wohlseparierten Paardekomposition (well-separated pair decomposition) verwendet, in denen mehrere Knoten zusammengefasst werden können. Dass der so angepasste Algorithmus in dieser Laufzeitklasse liegt, wird theoretisch bewiesen. Die experimentellen Ergebnisse entsprechen dem. Zudem wird anhand einer statistischen Auswertung der Zeichnungen, die beim mehrfachen Zeichnen der Rome-Graphen durch verschiedene Algorithmen erzeugt wurden, gezeigt, dass die Zeichnungen, die der angepasste Algorithmus liefert, annähernd so gut sind wie die des zugrunde liegenden kräftebasierten Graphzeichenalgorithmus. Der angepasste Algorithmus wird außerdem mit einer weiteren veränderten Form eines klassischen Graphzeichenalgorithmus, die einen Quadtree verwendet und den Algorithmus von Barnes und Hut implementiert, sowie mit dem Algorithmus von Kamada und Kawai verglichen.

Inhaltsverzeichnis

1 Einführung **4**

2 Grundlagen **6**
2.1 Graphen 6
2.2 Zeichnungen eines Graphen 6
2.3 Quadtrees 8
2.4 Wohlseparierte Paardekompositionen 9
2.5 Kräftebasierte Zeichenalgorithmen 10

3 Algorithmus **19**
3.1 Aufbau 19
3.2 Laufzeitanalyse 22
3.3 Eine einfache Beschleunigungstechnik 23

4 Ergebnisse **26**
4.1 Vergleich von FRLayout mit FRLayout2 26
4.2 Vergleich von FRLayout mit FRLayoutNoMaps 28
4.3 Vergleich von FRLayoutNoMaps mit FRLayoutNoMapsNoFrame 29
4.4 Vergleich von FRLayoutNoMapsNoFrame mit FRLayout++ 32
4.5 Vergleich von FRLayout++ mit Varianten von FRLayout++WSPD 33
4.6 Vergleich von FRLayout++ und FRLayout++WSPD mit FRLayout++Quadtree 38
4.7 Vergleich von FRLayout++ und FRLayout++WSPD mit KKLayout 40
4.8 Vergleich bei größeren Graphen 42
4.9 Abbildungen verschiedener Zeichnungen 44

5 Fazit und Ausblick **53**

1 Einführung

Ein grundsätzliches Problem der Graphentheorie ist die Visualisierung eines Graphen. Ein Graph besteht lediglich aus einer Menge von Knoten und einer Menge von Kanten, die jeweils eine „Verknüpfung“ von zwei Knoten darstellen. Eine explizite Zuordnung jedes Knotens und jeder Kante zu einem geometrischen Objekt wie einem Punkt oder einer Kurve in einem zugehörigen Raum erfolgt durch eine *Zeichnung*.

Für Graphen, die Objekte/Netzwerke unserer echten Welt modellieren, mag es sein, dass eine Zeichnung oft die reale geographische Lage abbilden soll. Landkarten, Pläne für Fahrstrecken öffentlicher Verkehrsmittel usw. sind hier zu nennen. Bei solchen ist eine bestimmte Anordnung der Objekte einer Zeichnung nach geographischen Gegebenheiten erwartet – möglicherweise in einer leicht abgeänderten Form wie bei einem schematisierten Straßenbahnnetzplan.

Es kann auch vorkommen, dass für einen gegebenen Graphen keine Zeichnung vorliegt oder eine vorhandene Zeichnung als für den gewünschten Zweck unbrauchbar eingeschätzt wird. Eine Zeichnung per Hand zu erstellen, ist aufwendig und damit teuer. Besser ist es eine Methode anzuwenden, mit der eine Zeichnung zu einem gegebenen Graphen automatisch erzeugt wird.

Verwandte Arbeiten Es gibt verschiedene solcher Methoden. Wenn beispielsweise eine Hierarchie, die dem innewohnt, was der gegebene Graph modelliert, deutlich gemacht werden soll, würde man den Graphen vermutlich anders zeichnen als dann, wenn eine besonders platzsparende Zeichnung gefordert ist. Außerdem muss berücksichtigt werden, dass bei der Implementierung eines Ansatzes immer auch nur bestimmte Ressourcen (Prozessoren, Speicher, ...) mit ihren jeweiligen Eigenschaften zur Verfügung stehen. Daher ist nicht alles Berechenbare auch in der Praxis praktikabel oder sinnvoll, da in einer, je nach Anwendung, angemessenen Zeit ein Ergebnis benötigt wird. Denkbare Qualitätsanforderungen an einen Algorithmus zum Graphzeichnen sind eine Minimierung der Anzahl der Kantenkreuzungen, ein begrenzter Platzverbrauch, eine bestimmte Struktur der die Kanten repräsentierenden Kurven (als gerade Linien, in orthogonalem Verlauf und Anordnung, ...), eine gleichmäßige Verteilung der Knoten über die verwendete Zeichenfläche, eine gute Winkelauflösung (keine zu kleinen Winkel zwischen zwei ausgehenden Kanten an einem Knoten), eine kleine Varianz der Kantenlängen, Erkennbarkeit von Symmetrie und weitere.

Viele Zeichenalgorithmen sind für speziellere Klassen von Graphen ausgelegt, für die sie entsprechend „gute“ Zeichnungen liefern, indem sie sich Eigenschaften dieser speziellen Graphen zu Nutze machen. So gibt es beispielsweise für Bäume Zeichenalgorithmen, die eine geradlinige Zeichnung für einen Baum $T = (V, E)$ in $O(|V|)$ Zeit kreuzungsfrei auf einem Gitter der Größe $O(|V| \log |V|)$ zurückgeben [DETT99]. Für einen Algo-

rithmus, mit dem eine hierarchische Struktur in einem Graphen möglichst gut sichtbar gemacht werden soll, ist der Algorithmus von Sugiyama et al. [STT81] ein Beispiel.

Auch für Zeichnungen von allgemeinen Graphen gibt es verschiedene Ansätze. Hier sind insbesondere die kräftebasierten Zeichenalgorithmen zu nennen, auf die in Abschnitt 2.5 näher eingegangen wird. In diesen Verfahren werden die die Knoten repräsentierenden Punkte als „Teilchen“ interpretiert, die analog zur Physik eine Wechselwirkung von Kräften ausüben. Diese Kraft ist abhängig von Distanz und Kantenmenge. Eine Methode zum Zeichnen eines Graphen, bei der eine Kraft verwendet wird, wurde bereits 1963 von Tutte [Tut63] veröffentlicht, allerdings nur für dreifach-knotenzusammenhängende, planare Graphen. Ein kräftebasiertes Verfahren zum Zeichnen von allgemeinen Graphen wurde in dieser Form erstmals von Eades als *spring-embedder* 1984 veröffentlicht [Ead84]. Auf diesem baut auch das Verfahren von Fruchterman und Reingold aus dem Jahre 1991 auf [FR91]. Prinzip dieser Verfahren ist es Federkräfte zwischen zwei adjazenten Knoten zu modellieren, mit denen die Knoten auf einen optimalen Abstand gebracht werden sollen und im Gegenzug abstoßende Kräfte zwischen jedem anderen Knotenpaar wirken zu lassen. Für letzteres ist $O(|V|^2)$ Zeit nötig, wobei V die Knotenmenge in einem Graphen $G = (V, E)$ ist. Da diese Laufzeitklasse hinderlich hoch liegt, gibt es bereits Methoden, mit denen die Verbesserung der Laufzeitklasse eines klassischen kräftebasierten Graphzeichenalgorithmus durch Approximation der abstoßenden Kräfte erreicht wird. In dieser Arbeit wird dasselbe angestrebt. Zu nennen ist hier zum einen der Graphzeichenalgorithmus von Quigley und Eades [QE01], die den Algorithmus von Barnes und Hut [BH86] zum Graphzeichnen anwenden. Der Algorithmus von Barnes und Hut nutzt einen Quadtree (siehe Abschnitt 2.3) als Datenstruktur. Zum anderen ist hier der Algorithmus von Hachul und Jünger [HJ04] zu nennen. Diese nutzen die Fast Multipole Method, die auf Greengard und Rokhlin [GR87] zurückgeht, um diese Laufzeitverbesserung zu erreichen. Mit diesen Algorithmen können in der Praxis, wie mit dem in dieser Arbeit vorgestellte Algorithmus, Laufzeiten in $O(|V| \log |V| + |E|)$ erreicht werden.

Eigener Beitrag In dieser Arbeit wird zur Näherung der abstoßenden Kräfte die wohlseparierte Paardekomposition (Well-Separated Pair Decomposition; siehe Abschnitt 2.4) von Callahan und Kosaraju [CK95] verwendet. Gronemann [Gro09] nutzt bei seinem Graphzeichenalgorithmus auch die wohlseparierte Paardekomposition. Er nutzt sie aber für die Anwendung seiner Umsetzung der Fast Multipole Method und nicht, wie im Rahmen dieser Arbeit getestet, als Datenstruktur, mit der unmittelbar die abstoßenden Kräfte approximiert werden. Implementiert und getestet wurde die Approximation der abstoßenden Kräfte durch eine wohlseparierte Paardekomposition hier exemplarisch am Algorithmus nach Fruchterman und Reingold.

Überblick über die Arbeit In Kapitel 2 werden zunächst einige Grundlagen eingeführt. Anschließend wird in Kapitel 3 der neue Ansatz vorgestellt, für welchen die Ergebnisse seiner Implementierung in Kapitel 4 untersucht werden. Abschließend wird in Kapitel 5 noch ein kurzes Fazit gezogen und ein kleiner Ausblick gegeben.

2 Grundlagen

In diesem Kapitel sollen zunächst einige Grundlagen eingeführt werden, auf die der zentrale Teil dieser Bachelorarbeit maßgeblich aufbaut. Nach einer Definition von Graphen und Zeichnungen wird der Quadtree und die wohlseparierte Paardekomposition vorgestellt. Zum Abschluss dieses Kapitels werden die kräftebasierten Graphzeichenalgorithmen eingeführt.

2.1 Graphen

Bereits in der Einführung war von Graphen die Rede. Diese sollen hier formal definiert werden. Ein Graph wird hier gemäß der gängigen Definition eines ungerichteten, einfachen Graphen definiert.

Definition 2.1. Ein *Graph* $G = (V, E)$ ist ein Paar aus einer endlichen Knotenmenge V und einer Kantenmenge E, wobei E eine Teilmenge der zweielementigen Teilmengen von V ist. Eine Kante ist also eine Menge $\{u, v\}$ mit $u, v \in V$ und $u \neq v$.

Eine Kante wurde hier als eine Menge von zwei Knoten definiert. Interpretiert wird sie aber für gewöhnlich als *Verbindung* zwischen diesen zwei Knoten. Deshalb wird im Folgenden auch von Kanten *zwischen* zwei Knoten gesprochen.

Mit dem allgemeinen Begriff Graph wird hier bewusst der ungerichtete, einfache Graph bezeichnet, da dieser der normalerweise in kräftebasierten Zeichenalgorithmen verwendete Typ eines Graphen ist. Die Verwendung dessen macht auch deshalb Sinn, da man andere Typen von Graphen wie gerichtete Graphen oder Graphen mit Mehrfachkanten einfach vor dem Zeichnen in einen ungerichteten, einfachen Graphen umwandeln kann. Beim Anwender liegt es dann die durch den Algorithmus zurückgegebene Zeichnung noch so anzupassen, dass der ursprüngliche Typ wieder erkennbar ist, falls er das möchte.

2.2 Zeichnungen eines Graphen

Ebenfalls bereits in der Einführung wurde der Begriff der Zeichnung verwendet. Dieser soll jetzt noch formal definiert werden. Die Definition orientiert sich am Buch von Di Battista et al. [DETT99].

Definition 2.2. Eine Abbildung ζ heißt *Zeichnung* eines Graphen $G = (V, E)$, wenn

- für alle $w \in V$ gilt: $\zeta(w) \in \mathbb{R}^2$ und
- für alle $\{u, v\} \in E$ gilt: $\zeta(\{u, v\}) = \zeta_{\{u,v\}}([0, 1])$, wobei $\zeta_{\{u,v\}}([0, 1])$ eine offene Jordankurve des Intervalls $[0, 1]$ mit $\zeta_{\{u,v\}}(0) = \zeta(u)$ und $\zeta_{\{u,v\}}(1) = \zeta(v)$ ist.

Eine Jordankurve ist dabei eine stetige und eindeutige Abbildung eines Intervalls auf Punkte des $\mathbb{R}^d$ [EL73], also eine Kurve, die keine Unterbrechung und im Unterschied zur „gewöhnlichen“ Kurve keine Kreuzung oder Berührung mit sich selbst haben darf. Hier ist $d = 2$, da nur Zeichnungen in der euklidischen Ebene betrachtet werden.

Da jedem Knoten und jeder Kante eines Graphen durch eine Zeichnung genau ein Punkt bzw. genau eine Jordankurve zugeordnet wird, werden im Folgenden die Punkte der Zeichnung, die einen Knoten repräsentieren, auch einfach direkt als „Knoten“ bezeichnet und die Jordankurven der Zeichnung, die eine Kante repräsentieren, auch einfach direkt als „Kanten“ bezeichnet, wenn daraus keine Mehrdeutigkeit entsteht. In der Einleitung wurde bereits so verfahren. So müsste der Beispielsatz „Der Abstand der Knoten v_1 und v_2 in dieser Zeichnung entspricht der Länge der geradlinigen Kante zwischen diesen Knoten“ eigentlich heißen „Der Abstand der v_1 und v_2 repräsentierenden Punkte entspricht der Länge der geradlinigen Jordankurve/Verbindungslinie, die die Kante zwischen diesen Knoten repräsentiert“. Diese Ungenauigkeit wird eingeführt um solche Dinge kürzer schreiben zu können.

Des weiteren wird noch die *geradlinige Zeichnung* definiert. Diese ist am Buch von Nishizeki und Rahman [NR04] orientiert.

Definition 2.3 (Geradlinige Zeichnung)**.** Eine Zeichnung eines Graphen G heißt *geradlinig*, wenn die Abbildung jeder Kante einem geraden Liniensegment entspricht.

Wie gerade definiert, ist eine Zeichnung eine Abbildung der Knoten eines Graphen auf Punkte in der Ebene und der Kanten desselben Graphen auf Kurven in der Ebene. Dabei ist zu beachten, dass nach Definition die Endpunkte der Kurven an den Positionen der Knoten in der Zeichnung liegen. Ist nun zu einem Graphen eine Zeichnung gesucht, bei der die Abbildung der Kanten auf Jordankurven bereits gegeben ist, so enthält diese Abbildung implizit auch schon eine Abbildung der Knoten auf Punkte, vorausgesetzt die Abbildung der Kanten ist konsistent bezüglich gleicher Endpunkte zweier Kanten, wenn diese denselben Knoten enthalten. Im umgekehrten Fall sind bei einer gegebenen Abbildung der Knoten auf Punkte die Endpunkte der Kurven vorgegeben, lediglich deren Verlauf ist noch offen. Bei einer geradlinigen Zeichnung ist jedoch auch der Verlauf vorbestimmt, da ein Liniensegment oder eine (gerade) Strecke durch zwei Punkte eindeutig definiert ist. Somit entspricht eine Zuordnung der Knoten auf Punkte in der Ebene immer auch einer geradlinigen Zeichnung. Allgemeiner entspricht sie auch jeder Zeichnung, deren Kantenverläufe eindeutig durch die Abbildung Knotenmenge auf Punkte festgelegt sind. Eine geradlinige Zeichnung kann zudem offenbar in $O(|E|)$ Zeit aus einer Abbildung der Kanten auf Punkte generiert werden. Damit ist eine Umwandlung möglich, die keine Probleme beim Einhalten der ermittelten Laufzeitklasse mit sich bringen.

2.3 Quadtrees

Ein *Quadtree* für Punktmengen (auch *Punkt-Quadtree*) ist eine Datenstruktur zur systematischen Speicherung von d-dimensionalen Punkten abhängig von den Koordinaten der einzelnen Punkte. Der Quadtree wurde erstmals 1974 von Finkel und Bentley [FB74] vorgestellt. Ein Quadtree kann auch zur Speicherung anderer (geometrischer) Objekte verwendet werden [Sam90]. Da dies jedoch hier nicht von Belang ist, bezieht sich die folgende Definition nur auf Punkt-Quadtrees und verschärft auch eine mögliche allgemeinere Definition, dass ein Wurzelbaum bzgl. eines d-dimensionalen Raums ein Quadtree ist, wenn jeder Knoten darin entweder 2^d Kinder hat oder ein Blatt ist.

Definition 2.4 (Quadtree)**.** Sei P eine Menge von Punkten im $\mathbb{R}^d$, wobei sich alle Punkte in P paarweise in mindestens einer Koordinate unterscheiden (also verschieden sind). Ein Wurzelbaum $Q = (W, E)$ heißt *Quadtree für P*, wenn

- jedes $w \in W$ einem d-dimensionalen Würfel (*Quadtree-Zelle*) im $\mathbb{R}^d$ entspricht,
- die Wurzel von Q alle Punkte aus P enthält und
- für jedes $w \in W$ gilt:
  - Wenn w mehr als einen Punkt aus P enthält, so hat w in Q genau 2^d Kinder. Diese Kinder sind gleich große d-dimensionale Würfel, die entstehen, wenn w in allen Dimensionen halbiert wird.
  - Wenn w einen oder keinen Punkt aus P enthält, so ist w ein Blatt in Q.

Wenn zwei Punkte nun sehr eng beieinander liegen kann es offenbar vorkommen, dass Quadtree-Zellen wieder und wieder geteilt werden müssen bis beide Punkte ihre „eigene“ Quadtree-Zelle haben. Tatsächlich wird die Tiefe eines Quadtrees vom geringsten Abstand zweier Punkte aus der zugehörigen Punktmenge beschränkt. Für die Tiefe t eines Quadtrees für eine Menge von Punkten P in der Ebene gilt $t \leq \log(s/c) + 3/2$, wobei s die Seitenlänge der Wurzel des Quadtrees ist und c der geringste Abstand zweier Punkte in P ist [dCvO08]. In Abhängigkeit von der Tiefe t lässt sich ein Quadtree für die Punktmenge P in einer Zeit von $O((t + 1) \cdot |P|)$ konstruieren [dCvO08].

Eine zur Konstruktion nötige Laufzeit, die von der Verteilung der Punkte abhängt, scheint wenig interessant, da zwei Punkte beliebig nah beieinander liegen können. Eine Lösung stellt hier ein *komprimierter Quadtree* dar.

Definition 2.5 (Komprimierter Quadtree)**.** Ein *komprimierter Quadtree* ist ein Baum, den man erhält, wenn man bei einem Quadtree diejenigen Pfadabschnitte zu einer einzigen Kante „komprimiert“, die über innere Knoten führen, die drei punktleere Kinder enthalten.

Ein komprimierter Quadtree für eine Punktmenge P kann in $O(|P| \log |P|)$ Zeit unabhängig von der Verteilung der Punkte aus P berechnet werden [Har11].

2.4 Wohlseparierte Paardekompositionen

Die wohlseparierte Paardekomposition ist eine Datenstruktur für Punkte eines d-dimensionalen Raumes. Eingeführt wurde sie erstmals 1992 von Callahan und Kosaraju [CK95] als *well-separated pair decomposition*. Die folgenden Definitionen folgen den Definitionen aus dem Buch von Narasimhan und Smid [NS07] und der Arbeit von Callahan und Kosaraju [CK95].

Definition 2.6 (Wohlsepariertes Paar)**.** Sei $s > 0$ eine reelle Zahl und seien A und B zwei endliche Mengen von Punkten im $\mathbb{R}^d$. A und B heißen *wohlsepariert* bezüglich s oder *s-wohlsepariert* und bilden ein *wohlsepariertes Paar*, wenn es zwei disjunkte d-dimensionale Kugeln C_A und C_B gibt, sodass

- C_A und C_B denselben Radius haben,
- C_A die Bounding-Box $R(A)$ enthält,
- C_B die Bounding-Box $R(B)$ enthält und
- die Distanz zwischen C_A und C_B größer oder gleich s mal dem Radius von C_A bzw. C_B ist.

Dabei sei bei einer gegebenen Punktmenge P von d-dimensionalen Punkten die *Bounding-Box R(P)* der kleinste koordinatenachsenparallele d-dimensionale Quader, der alle Punkte aus P enthält. Auf der Definition der wohlseparierten Paare aufbauend die nächste Definition:

Definition 2.7 (Wohlseparierte Paardekomposition)**.** Es sei P eine Menge von n Punkten im $\mathbb{R}^d$ und $s > 0$ eine reelle Zahl. Eine *wohlseparierte Paardekomposition* (WSPD) bezüglich s ist eine Folge $\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_m, B_m\}$ von Paaren aus nichtleeren Teilmengen von P mit einer natürliche Zahl m, sodass

- A_i und B_i für jedes i mit $1 \leq i \leq m$ wohlsepariert bezüglich s sind und
- es für zwei beliebige Punkte p und q aus P mit $p \neq q$ genau einen Index i mit $1 \leq i \leq m$ gibt, sodass $p \in A_i$ und $q \in B_i$ ist oder $p \in B_i$ und $q \in A_i$ ist.

Definition 2.8 (Split-Tree)**.** Gegeben sei eine nichtleere Menge P aus d-dimensionalen Punkten. Ein *Split-Tree* ist ein gewurzelter Binärbaum, dessen Knoten d-dimensionalen Quadern entsprechen. Wenn P nur aus einem Punkt besteht, so besteht der Split-Tree nur aus einem Knoten, der diesen Punkt enthält. Ansonsten enthält die Wurzel des Split-Trees alle Punkte aus P und hat zwei Kinder, die selbst Split-Tree für je eine nichtleere Teilmenge von P sind. Diese zwei Teilmengen sind disjunkt und werden durch eine Teilung der Punktmenge P entlang einer Hyperebene erhalten.

Durch die mehrfache disjunkte Teilung der Punktmenge enthält offensichtlich jedes Blatt eines Split-Trees genau einen Punkt aus der zugehörigen Punktmenge, den kein anderes Blatt enthält. Es kann daher eine bijektive Zuordnung der Punkte einer Punktmenge auf Blätter des zugehörigen Split-Trees sowie auf Wurzel-Blatt-Pfade des zugehörigen Split-Trees angenommen werden.

Außerdem gilt folgender Satz:

Satz 2.9 (nach Narasimhan und Smid [NS07])**.** *Sei P eine Menge von n Punkten im $\mathbb{R}^d$ und $s > 0$ eine reele Zahl. Die folgenden Punkte gelten nun:*

- *Ein Split-Tree für P kann in $O(n \log n)$ Zeit berechnet werden. Dieser hat eine Größe in $O(n)$ und hängt nicht von s ab.*
- *Bei einem gegebenen Split-Tree kann eine WSPD für P bezüglich s mit $O(s^d n)$ Paaren in $O(s^d n)$ Zeit berechnet werden.*

Narasimhan und Smid verweisen ihrerseits wiederum auf Callahan und Kosaraju [CK95], die dasselbe zeigen. Dort wird als Form des Split-Trees der *Fair-Split-Tree* verwendet, der auch in $O(n \log n)$ Zeit konstruiert werden kann [CK95, Cal95]. Har-Peled [Har11] konstruiert eine WSPD mit denselben Eigenschaften aus einem komprimierten Quadtree.

Für gewöhnlich wird daher ein Baum mit einer WSPD assoziiert. Bei den gerade genannten Arbeiten entsprechen dabei nach Konstruktion der WSPD die Punktmengen A und B eines wohlseparierten Paares $\{A, B\}$ aus der WSPD jeweils genau den Punkten in genau einem Knoten des entsprechenden Baumes. Dass dem so ist, folgt direkt aus den dort angegebenen Algorithmen zur Konstruktion der WSPD. Da ich bei meiner Implementierung der WSPD dem Buch von Narasimhan und Smid folge, wird im Folgenden ein Paarteil eines wohlseparierten Paares auch mit einem Split-Tree-Knoten in Verbindung gebracht und umgekehrt.

2.5 Kräftebasierte Zeichenalgorithmen

Grundgedanke Aus der Physik ist bekannt, dass Teilchen oder Körper in einem physikalischen Raum wechselseitig verschiedene Kräfte wie Gravitationskräfte oder elektrische Kräfte (Coulomb-Kräfte) ausüben. Das heißt sie üben eine Anziehung oder eine Abstoßung (oder nichts von beidem) von unterschiedlicher Größe auf alle anderen Teilchen in diesem Raum aus. Diese kann von Parametern wie der Distanz abhängen. Damit wirkt auf jedes Teilchen eine resultierende Kraft, die der Summe aller Kräfte, die die anderen Teilchen auf dieses Teilchen ausüben, entspricht. Durch die wirkenden Kräfte werden die Teilchen beschleunigt.

Das Problem die Bewegungen der einzelnen Teilchen bzw. Körper in so einem System vorauszuberechnen wird als *N-Körper-Problem* und die Simulation eines solchen Systems als *N-Körper-Simulation* bezeichnet. Für zwei Körper ist eine exakte Lösung des Problems bekannt. Da für drei oder mehr Körper das N-Körper-Problem nicht auf dieselbe Art wie für zwei Körper gelöst werden kann [Mey99], kann man sich helfen, indem die Simulation iterativ durchgeführt wird, also vom aktuellen Zeitpunkt zu einem nächsten diskreten Zeitpunkt.

Nun kann man dieses Prinzip der Wechselwirkung von Kräften auf das Graphzeichnen übertragen. Hierfür ist der Raum die Zeichenebene, die Knoten des Graphen werden als „Teilchen“ modelliert und es werden eine Funktion oder mehrere Funktionen zur Berechnung der wirkenden Kräfte benötigt, die von der Distanz der Knoten im Modell und

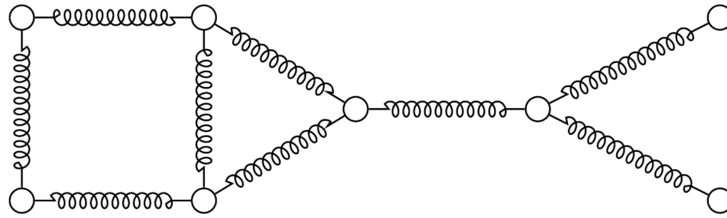


Abb. 2.1: Prinzip eines kräftebasierten Graphzeichenalgorithmus. Entnommen aus der Diplomarbeit von Gronemann [Gro09].

der Kantenmenge E abhängen. Beim Graphzeichnen interessieren jedoch keine physikalischen Größen der Teilchen wie die Geschwindigkeiten oder die Massen, sondern nur die Positionen. Deshalb wird im klassischen kräftebasierten Modell zur Graphzeichnung keine Beschleunigung und eine damit verbundene Geschwindigkeitsänderung der Teilchen berechnet, sondern nur eine Positionsänderung. Daher wird statt der Zuordnung der Knoten auf „Modellteilchen“ einfach eine Zuordnung auf Punkte im $\mathbb{R}^2$ verwendet. Im folgenden wird der Punkt, der den Knoten $v \in V$ im Graphzeichenalgorithmus repräsentiert, auch mit p_v bezeichnet.

Idee ist es nun die Kräfte zwischen diesen Punkten, die die Knoten modellieren, eine Zeit lang wirken zu lassen, um so schließlich einen stabilen Gleichgewichtszustand zu erreichen. Dabei ist allerdings eine große Schwäche solcher kräftebasierten Algorithmen, dass das ganze immer auch einem lokalen modellenergetischen Minimum entgegenstreben kann, das nicht gleichzeitig auch ein globales modellenergetisches Minimum ist. Eine Visualisierung dieses Prinzips kann in Abb. 2.1 betrachtet werden.

Prinzipielle Funktionsweise Die Funktionsweise eines klassischen kräftebasierten Algorithmus ist in Algorithmus 1 skizziert. Der Aufbau ist dabei insbesondere am Algorithmus von Eades [Ead84] und am Algorithmus von Fruchterman und Reingold [FR91] orientiert.

Der Algorithmus nimmt einen Graphen entgegen und gibt eine totale Zuordnung der Knoten auf Punkte in der Ebene zurück. Diese Zuordnung entspricht auch einer geradlinigen Zeichnung und offensichtlich auch jeder anderen beliebigen Zeichnung, deren Abbildung der Kanten eindeutig aus der Abbildung der Knotenmenge folgt. Je nach Umsetzung kann dem Algorithmus noch eine initiale Zuordnung der Knoten auf Punkte übergeben werden. Diese soll dazu dienen ein „besseres“ Ergebnis zu erhalten, da sie möglicherweise eine „bessere“ Ausgangslage für die erste Iteration darstellt. Steht nichts zur Verfügung, ist eine zufällige Anfangsverteilung üblich, die als Ausgangspunkt für die erste Iteration dient. Der Algorithmus funktioniert nämlich iterativ. Das heißt hier, dass die Prozedur des Berechnens und Anwendens der Kräfte mehrfach angewandt wird – jeweils auf die in der vorherigen Iteration berechneten Positionen der Knoten. So soll das Ergebnis jede Iteration besser werden. Das Abbruchkriterium ist im einfachsten Fall einfach eine Überschreitung eines Maximalwerts für die Anzahl der Iterationen, sodass beispielsweise bei jedem Aufruf genau 50 Iterationen durchgeführt werden. In

einer „besseren“ Umsetzung werden so viele Iterationen durchgeführt bis ein (lokales) modellenergetisches Minimum ausreichend gut erreicht ist, wobei eine Höchstzahl an Iterationen zugefügt werden kann, um ein Terminieren (nach konstant vielen Durchläufen) zu garantieren.

Algorithmus 1 : Ablauf eines klassischen kräftebasierten Zeichenalgorithmus

Eingabe : Ein Graph $G = (V, E)$, evtl. eine Zuordnung $V \to \mathbb{R}^2$
Ausgabe : Eine Zuordnung $V \to \mathbb{R}^2$

```
1 Berechne ggf. initiale Postionen (Punkte in der Ebene) für alle v ∈ V
  // Jeder Durchlauf dieser while-Schleife entspricht einer Iteration
2 while Abbruchkriterium nicht erfüllt do
3     Berechne für alle v ∈ V die Summe der wirkenden Kräfte und speichere sie
      // Resultierende Kräfte anwenden
4     foreach v ∈ V do
5         Berechne neue Position von v
6 return Berechnete Zuordnung V → ℝ²
```

In dem Schritt „Berechne für alle $v \in V$ die Summe der wirkenden Kräfte und speichere sie“ wird für jeden Knoten die auf ihn in Summe wirkende Kraft gespeichert. Diese wird im darauffolgenden Durchlauf aller Knoten verwendet, um die neuen Positionen zu berechnen. Wie in der Physik werden hierbei die wirkenden Kräfte mithilfe von Vektoren berechnet. Wenn $u \in V$ eine Kraft auf $v \in V$ ausübt, so wirkt diese Kraft entweder genau in die Richtung von p_v nach p_u (anziehende Kraft) oder genau in die umgekehrte Richtung (abstoßende Kraft). Der Einheitsvektor in einer dieser beiden Richtungen wird daher als richtungsbeschränkendes Element verwendet. In dieser Arbeit wird er in Richtung von p_v nach p_u verwendet. Zur Bestimmung der Kraft wird dieser Einheitsvektor mit dem Wert einer austauschbaren Funktion f multipliziert, die in Abhängigkeit von der Distanz zwischen diesen beiden Punkten eine reellen Zahlenwert zurückgibt. Mit dem Betrag dessen wird die Stärke und mit dem Vorzeichen die Richtungen der Kraft festgelegt. Ist dieser Wert positiv, so wirkt die Kraft anziehend (von p_v nach p_u). Ist sie dagegen negativ, so wird die Richtung des resultierenden Vektors „umgekehrt“ und die Kraft wirkt abstoßend. Insgesamt ergibt sich folgende Formel zur Berechnung der Kraft, die u auf v ausübt:

$$\vec{F}(u,v) = \frac{\overrightarrow{p_v p_u}}{\|\overrightarrow{p_v p_u}\|} \cdot f(\|\overrightarrow{p_v p_u}\|) \tag{2.1}$$

Der Vektor $\overrightarrow{p_v p_u}$ durch den Betrag desselben ergibt den Einheitsvektor in diese Richtung. Die Funktion f zur Berechnung der abstoßenden Kraft wird mit $f_{\text{abstoßend}}$ und die zur Berechnung der wenigstens teils anziehenden Kraft zwischen zwei adjazenten Knoten als f_{adjazent} bezeichnet. Die diese Funktionen verwendende Funktion $\vec{F}(u,v)$ wird analog dazu $\vec{F}_{\text{abstoßend}}(u,v)$ bzw. $\vec{F}_{\text{adjazent}}(u,v)$ genannt. Es wird hier die Bezeichnung f_{adjazent} statt $f_{\text{anziehend}}$ verwendet, da diese Kraft z.B. bei Eades als Federkraft auch abstoßend

wirken kann, wenn zwei adjazente Knoten näher beisammen liegen als bei Einhaltung des optimalen Abstands erwartet. Zudem wird hier $f_{\text{abstoßend}}$ statt $f_{\text{nicht-adjazent}}$ verwendet, da z.B. bei Fruchterman und Reingold diese abstoßende Kraft auch zwischen adjazenten Knotenpaaren wirkt.

Klassische Umsetzungen Solch ein Verfahren wurde erstmals von Eades [Ead84] vorgestellt. Er modelliert zum einen *Federkräfte* zwischen zwei adjazenten Knoten, also zwei Knoten u und v, für die es eine Kante $\{u, v\} \in E$ gibt, und zum anderen abstoßende Kräfte für die Knotenpaare $\{u, v\} \notin E$. Das Prinzip dieser Federkräfte ist den aus der Physik bekannten Federkräften entlehnt. Eine Feder hat eine optimale Länge, bei der sie sich im energieärmsten Zustand befindet. Unter der Aufwendung von Kraft kann die Feder jedoch zusammengedrückt werden, dass diese kürzer wird, oder auseinandergezogen werden, dass diese länger wird. So wird an ihr Spannarbeit verrichtet, die anschließend als potenzielle Energie in der gespannten Feder vorliegt. Die Feder strebt jedoch ihrem energieärmsten Zustand entgegen, sodass sie ihre veränderte Länge nur behält, so lange noch die entsprechende Kraft mit entsprechender Richtung und entsprechendem Betrag anliegt. Wenn dieses Prinzip beim kräftebasierten Graphenzeichenen zwischen zwei adjazenten Knoten angewandt wird, besteht die hilfreiche Eigenschaft, dass es einen optimalen Abstand adjazenter Knoten (optimale Kantenlänge bei geradlinigen Zeichnungen) größer 0 gibt, zu deren Gunst die entsprechende Kraft zwischen diesen adjazenten Knoten wirkt.

Dieser Effekt wird durch die Verwendung der folgenden Funktion erreicht, die für die Funktion f innerhalb der Formel (2.1) verwendet wird.

$$f^{\text{Eades}}_{\text{adjazent}}(d) = c_a \log \frac{d}{l} \tag{2.2}$$

Hierbei ist c_a eine Konstante und l ist der optimale Abstand zweier adjazenter Knoten, also im Modell die ideale Federlänge. Diese logarithmische Formel entspricht nicht realen Formeln zur Berechnung von Federkräften in der Physik.

Wie f_{adjazent} nimmt auch $f_{\text{abstoßend}}$ eine Distanz entgegen und gibt die Größe der wirkenden Kraft zurück. Da der Wert in dieser Funktion hier immer negativ ist, wirkt diese Kraft immer entgegen dem Einheitsvektor in Formel (2.1). Diese abstoßende Kraft zwischen nicht-adjazenten Knoten berechnet sich bei Eades mit:

$$f^{\text{Eades}}_{\text{abstoßend}}(d) = -\frac{c_r}{d^2} \tag{2.3}$$

Dabei ist c_r eine Konstante.

Die insgesamt auf einen Knoten $v \in V$ in einer Iteration wirkende Kraft $\vec{F}_{\text{gesamt}}$ ist, wie zuvor geschrieben, die Summe der Kräfte. Das ist bei Eades:

$$\vec{F}^{\text{Eades}}_{\text{gesamt}}(v) = \sum_{\{u,v\} \in E} \vec{F}_{\text{adjazent}}(u, v) + \sum_{\{u,v\} \notin E, u \neq v} \vec{F}_{\text{abstoßend}}(u, v) \tag{2.4}$$

Als weiterer bedeutender Graphzeichenalgorithmus dieser Art ist noch das Verfahren nach Fruchterman und Reingold [FR91] zu nennen. Das Verfahren ist zeitlich nach

dem von Eades erschienen und basiert im Wesentlichen auch auf ebendiesem. Der Algorithmus verwendet einen *Rahmen* (*frame*) zur Begrenzung der Zeichenfläche. Dieser Rahmen ist rechteckig und sollte am Ende einer Iteration ein Knoten außerhalb dieses Rahmens platziert werden, so werden dessen Koordinaten so vermindert oder erhöht, dass er auf und nicht außerhalb des Rahmens liegt. Im Gegensatz zum Algorithmus von Eades wirken nicht nur zwischen nicht adjazenten Knoten abstoßende Kräfte, sondern zwischen jedem Paar von Knoten. Zusätzlich wirken anziehende Kräfte zwischen adjazenten Knoten. Es wird neben ihren primären Funktionen für f_{adjazent} und $f_{\text{abstoßend}}$ (Der Algorithmus, der diese Funktionen verwendet, wird mit *FR* oder *FR1* abgekürzt) noch ein Paar alternativer Funktionen, die sie aber selbst als schlechter geeignet erachten, erwähnt. Der Algorithmus, der dieses verwendet wird mit *FR2* abgekürzt.
Es ist

$$f^{\text{FR1}}_{\text{adjazent}}(d) = \frac{d^2}{k} \qquad f^{\text{FR1}}_{\text{abstoßend}}(d) = -\frac{k^2}{d} \tag{2.5}$$

und

$$f^{\text{FR2}}_{\text{adjazent}}(d) = \frac{d}{k} \qquad f^{\text{FR2}}_{\text{abstoßend}}(d) = -\frac{k}{d} \tag{2.6}$$

mit k als dem optimalen Abstand adjazenter Knoten (optimale Kantenlänge in einer geradlinigen Zeichnung), der sich errechnet als:

$$k = c\sqrt{\frac{A}{|V|}} \tag{2.7}$$

Dabei ist c eine Konstante, A die Fläche im verwendeten Rahmen und $|V|$, wie sonst auch, die Mächtigkeit der Knotenmenge V. Insgesamt berechnet sich die Kraft in einer Iteration auf einen Knoten $v \in V$ mit:

$$\vec{F}^{\text{FR1/FR2}}_{\text{gesamt}}(v) = \sum_{\{u,v\}\in E} \vec{F}_{\text{adjazent}}(u,v) + \sum_{u\in V, u\neq v} \vec{F}_{\text{abstoßend}}(u,v) \tag{2.8}$$

Im ersten Moment mag es im Hinblick auf das Federmodell von Eades seltsam erscheinen, dass f_{adjazent} hier nicht für sehr kleine d einen Wert mit anderem Vorzeichen zurückgibt als für sehr große d, wie es der Logarithmus bei Eades garantiert und womit dort eine optimaler Abstand adjazenter Knoten größer 0, der auch nicht unendlich ist, gewährleistet wird. Jedoch gibt es auch bei Fruchterman und Reingold eine optimale Kantenlänge größer 0, die dadurch zustande kommt, dass für eine adjazentes Knotenpaar auch die abstoßende Kraft wirkt. Diese überwiegt bei einem d kleiner als der optimalen Länge, während bei einem d größer der optimalen Länge die anziehende Kraft überwiegt. Für die optimale Länge d^* gilt $f_{\text{adjazent}}(d^*) = -f_{\text{abstoßend}}(d^*)$. Wie man sieht, ist diese Gleichung bei $d^* = k$ erfüllt.

Auch führen Fruchterman und Reingold das Konzept des *Simulated Annealing* im Schritt der Neuberechnung der Position eines Knotens am Ende einer Iteration ein. Das ist ein Optimierungsverfahren, das beispielsweise hier angewandt wird, indem in frühen Iterationen große Verschiebungen der Knoten möglich sind, in späteren Iterationen jedoch nur noch immer kleinere Verschiebungen möglich sind. Das hat den Zweck, eine

bereits relativ gute Anordnung durch große Veränderungen gegen Ende der Berechnung nicht wieder zunichtezumachen. Es entlehnt seinen Namen dem physikalischem Prozess des langsamen Abkühlens von Metallen. Deshalb wird im Algorithmus von Fruchterman und Reingold auch von einer *Temperatur* gesprochen, die zu Beginn hoch ist und anschließend sinkt.

Laufzeitanalyse Die Laufzeit des Algorithmus von Eades und des Algorithmus von Fruchterman und Reingold ist beides mal $O(|V|^2)$. Orientiert man sich an den Schritten von Algorithmus 1, so ist die Berechnung von initialen Positionen in $O(|V|)$ möglich (z. B. zufällige Positionen). Im Inneren der while-Schleife benötigt der erste Schritt, das Berechnen und Speichern aller wirkenden Kräfte, $O(|V|^2)$ Zeit. Das ist schnell ersichtlich, wenn man sich beispielsweise den Ablauf dieses Schrittes bei Fruchterman und Reingold, der in Algorithmus 2 schematisiert ist, ansieht.

Algorithmus 2 : Berechnen und Speichern der Summe der wirkenden Kräfte für alle $v \in V$ bei Fruchterman und Reingold

```
  // Abstoßende Kräfte berechnen
1 foreach v ∈ V do
2     v.disp = 0⃗ // v.disp sei der Vektor, der der Summe aller wirkenden
          Kräfte auf v entspricht, also der Wert, der am Ende der
          Iteration als Grundlage für die Verschiebung der Position von v
          dient
3     foreach u ∈ V do
4         if v ≠ u then
5             v.disp = v.disp + F⃗_abstoßend(u, v)
  // Anziehende Kräfte berechnen
6 foreach {u, v} ∈ E do
7     u.disp = u.disp + F⃗_adjazent(v, u)
8     v.disp = v.disp + F⃗_adjazent(u, v)
```

Offensichtlich wird für die Berechnung und Speicherung der abstoßenden Kräfte $O(|V|^2)$ Zeit benötigt und anschließend $O(|E|)$ Zeit für die Berechnung und Speicherung der anziehenden Kräfte. Da in einem ungerichteten, einfachen Graphen höchstens $\binom{|V|}{2} \in O(|V|^2)$ Kanten vorhanden sein können, wird die Laufzeitklasse hier durch $O(|V|^2)$ beschränkt. Das Anwenden der aufsummierten Kraft ist für alle Knoten offenbar in $O(|V|)$ möglich, sodass dies für die Laufzeitklasse keine Probleme macht. Das ganze befindet sich jedoch noch in dieser while-Schleife, sodass die Laufzeitklasse $O(I \cdot |V|^2)$ sein muss, wobei I ein Term für die asymptotische Begrenzung der Anzahl der Iterationen ist. Sowohl Eades als auch Fruchterman und Reingold beschränken die Höchstzahl der Iterationen jedoch mit einer Konstanten, sodass dieser Faktor keinen Einfluss auf die Laufzeitklasse hat.

Fruchterman und Reingold haben in ihrer hier zitierten Arbeit noch eine Methode zur Verbesserung dieser Laufzeit vorgestellt. In der so genannten *Gitter-Variante* (*grid-variant*) werden die abstoßenden Kräfte für einen Knoten v nur in einem Umkreis von $2k$ berechnet. Hierfür wird die Zeichenfläche, also der durch den Rahmen begrenzten Bereich, in Quadrate mit Seitenlänge $2k$ eingeteilt. Ein Knoten u, der dann noch zur Berechnung der abstoßenden Kraft für einen Knoten v betrachtet werden muss, befindet sich im selben Quadrat wie v oder in einem der acht dieses Quadrat einschließenden Quadrate. Das wird damit gerechtfertigt, dass die abstoßenden Kräfte vor allem bei Knoten in der Nähe einen bedeutenden Einfluss haben, während weit entfernte Knoten kaum noch relevant wirken, da deren Betrag relativ klein wird. Das kann jedoch unter Umständen problematisch sein, da viele Knoten, die zusammen genommen auch einen gewissen Einfluss haben, einfach ignoriert werden. Zu einem Vorteil kann es werden, wenn ein Graph mehrere dicht vernetzte Bereiche hat, die untereinander jedoch nur spärlich verbunden sind. Dann würden die wenigen diese Bereiche verbindenden Kanten nicht viel länger als die anderen Kanten gezogen werden. Anders wäre es in der klassischen Variante, da sich dort die „Knotenhaufen“ als ganzes merklich abstoßen [FR91]. Eine Laufzeitklassenverbesserung bringt diese Variante allerdings nur, wenn man von einer einigermaßen gleichmäßige Verteilung der Knoten über die Zeichenfläche in jeder Iteration ausgeht [FR91, QE01]. Nimmt man diese an, so liegt die Laufzeit dieser Variante in $O(|V| + |E|)$. Fruchterman und Reingold konnten selbst nicht sagen, dass die Variante mit dem Gitter generell schlechter oder generell besser ist als die ohne Gitter, sodass sie beide Varianten in ihre Arbeit aufnahmen.

Weitere Verfahren dieses Prinzips Viele Verfahren bauen, mit dem Ziel eine bessere Qualität und/oder eine bessere Laufzeit zu liefern, auf dem Prinzip dieser klassischen Verfahren auf. Graphen, die von einem Graphzeichenalgorithmus gezeichnet werden sollen, haben typischerweise $O(n)$ Kanten und $\Omega(n^2)$ Knotenpaare ohne gemeinsame Kante, wobei n gleich die Anzahl der Knoten ist [QE01]. Damit wird in klassischen Graphzeichenalgorithmen die Berechnung der abstoßenden Kräfte zum „Flaschenhals“ für die Laufzeitklasse. Diesem Problem begegnet man auch in den mit diesen verwandten N-Körper-Simulationen, in denen zu jedem Paar von Teilchen die Wechselwirkung der Kräfte berechnet werden muss. Auch dort soll die Laufzeitklasse von $O(n^2)$ mit n gleich der Anzahl der Teilchen abgesenkt werden. Hier gibt es eine Reihe von Algorithmen wie den Algorithmus nach Barnes und Hut [BH86], die die abstoßenden Kräfte nur näherungsweise berechnen.

Dort werden die Teilchen in einem Octree, das ist ein Quadtree im dreidimensionalen Raum, erfasst. In jeder Iteration wird ein Octree über alle Teilchen gebildet und dann wird für jedes Teilchen beginnend mit der Wurzel verglichen, ob $l/d < \theta$. Dabei ist l die Seitenlänge der gerade betrachteten Octreezelle, d die Distanz vom Schwerpunkt der Teilchen dieser Zelle zum gerade betrachteten Teilchen und θ ein Parameter, der in etwa 1 sein soll. Ist diese Ungleichung wahr, so wird die Kraft, die der Schwerpunkt der Teilchen dieser Octreezelle auf das betrachtete Teilchen ausübt, dem Teilchen zugefügt. Ist sie nicht wahr, so wird der Vergleich für die acht Kinder dieser Octree-

zelle durchgeführt. Im Ergebnis üben so entfernte Teilchen in zusammengefasster Form Kräfte aus, während nahe Teilchen einzeln oder in kleineren Mengen zusammengefasst Kräfte ausüben. Die Laufzeit dieses Algorithmus wird mit $O(n \log n)$ angegeben, wobei n hier und im Folgenden gleich der Anzahl der Teilchen ist. Der klassische Ansatz von Barnes und Hut einfach einen Octree zu konstruieren ist, wie zuvor beschrieben, von der Verteilung der Punkte abhängig und damit im Allgemeinen nicht in $O(n \log n)$ Zeit möglich [APG94]. Ein komprimierter Octree kann als ein komprimierter Quadtree im Dreidimensionalen, wie vorhin erwähnt, jedoch in $O(n \log n)$ Zeit berechnet werden. Der anschließende Durchlauf des Baumes für alle Teilchen läuft nur in sehr konstruierten Fällen (oder bei $\theta = 0$) in $O(n^2)$ statt in $O(n \log n)$ Zeit ab [Sal91], sodass insgesamt von einer Laufzeitklassenverbesserung in jeder Iteration auf $O(n \log n)$ ausgegangen werden kann. Dieser Algorithmus wurde durch Quigley und Eades [QE01] als Methode zum Graphzeichnen eingeführt.

Ein weiterer Ansatz zur Beschleunigung von N-Körper-Simulationen ist die *Fast Multipole Method* (FMM), die auf Multipolentwicklung beruht und so erstmals von Greengard und Rokhlin [GR87] vorgestellt wurde. Die Laufzeit dieses Algorithmus wird mit $O(n)$ angegeben, was von Aluru et al. [APG94] mit Verweis auf die Abhängigkeit der Verteilung im Raum angezweifelt wird. Sie geben daher für die FMM und den Algorithmus von Barnes und Hut eine alternative Datenstruktur an, die in $O(n \log n)$ Zeit im Zweidimensionalen und in $O(n \log^2 n)$ Zeit im Dreidimensionalen konstruiert werden kann. Ähnlich dazu kann mit einer gegebenen WSPD, die in $O(n \log n)$ konstruiert werden kann, zu einer Punktmenge (die n Teilchen werden nach ihrer Lage als Punkte angenommen) die FMM in $O(n)$ Zeit durchgeführt werden [Cal95]. Hachul und Jünger [HJ04] wenden die FMM zum Graphzeichnen an und kommen damit auf eine Laufzeit in $O(|V| \log |V| + |E|)$ für einen Graphen $G = (V, E)$.

Prinzipiell müssten sich so auch weitere beschleunigte Algorithmen für N-Körper-Simulationen zum Graphzeichnen anwenden lassen.

Ein weiterer bekannter kräftebasierte Graphzeichenalgorithmen ist der von Davidson und Harel [DH96]. Dieser fügt dem Algorithmus, um bessere Ergebnisse zu erhalten, einige Beschränkungen zu, wodurch nur noch eine Laufzeit in $O(|V|^2|E|)$ garantiert wird. Der GEM-Algorithmus von Frick et al. [FLM95] verbessert das Konvergieren des Verfahrens unter anderem durch lokale Temperaturen und Erkennung von Rotation und Oszillation. Weitere verbesserte Algorithmen sind JIGGLE von Tunkelang [Tun98] oder das Verfahren von Hu [Hu05].

Multidimensionale Skalierung Die zweite bedeutende Methode zum Zeichnen von allgemeinen Graphen neben dem klassischen, zuvor vorgestellten Prinzip mit abstoßenden Kräften und Federkräften ist die Anwendung der *Multidimensionalen Skalierung* (MDS) zum Graphzeichnen. Bei der multidimensionalen Skalierung sollen Objekte mit gegebenen Distanzen so im Raum positioniert werden, dass ihr Abstände im Raum so gut wie möglich den gegebenen Distanzen entsprechen. Modellhaft gesprochen wirken hier zwischen allen Knoten Federkräfte und dafür keine abstoßenden Kräfte mehr.

Als erster bedeutender Graphzeichenalgorithmus dieser Art ist der Algorithmus von

Kamada und Kawai [KK89] zu nennen. Der angestrebte Abstand zwischen zwei Knoten hängt von ihrer graphentheoretischen Distanz ab. In jeder Iteration wird nun versucht die Energie des Systems dieser Federkräfte zu senken, indem der am „unpassendsten“ liegende Knoten auf eine energetisch günstigere Position gesetzt wird. Die Berechnung aller kürzesten Distanzen im Graphen kann in $O(|V|^2 \log|V| + |E||V|)$ Zeit durchgeführt werden. Dieser Schritt bestimmt auch die Gesamtlaufzeitklasse des Algorithmus. Zudem benötigt der Algorithmus für die Speicherung dieser Distanzen $O(|V|^2)$ Speicherplatz und ist damit insgesamt, verglichen mit den zuvor vorgestellten Algorithmen, teurer in der Berechnung [Kob12].

Es gibt zahlreiche neuere Verfahren, die sich in einigen Belangen von diesem unterscheiden und auch deutlich bessere Laufzeitklassen erreichen, sodass diese Verfahren auch für große Graphen relevant sind. Beispiele sind die Arbeit von Gansner et al. [GKN04], Landmark MDS von Silva und Tenenbaum [ST02], Pivot MDS von Brandes und Pich [BP07] oder COAST von Gansner et al. [GHK13].

Multilevel Graphzeichenalgorithmen Die klassischen Graphzeichenalgorithmen wie die von Eades, Fruchterman und Reingold oder Kamada und Kawai sind nur für kleine Graphen mit etwa bis zu 100 Knoten ausgelegt [HK02]. Sie haben ein Problem mit großen Graphen, da sie zum einen oft daran scheitern dort ein (gutes) modellenergetisches Minimum zu erreichen und zum anderen aufgrund ihrer Laufzeitklasse viel Zeit benötigen [GGK04]. Eine Möglichkeit dem zu begegnen ist es, so genannte *Multilevel* Graphzeichenalgorithmen zu verwenden. Prinzip derer ist es einen Graphen G durch „Verschmelzen“ von Knoten in mehreren Stufen (Levels) jedes mal „gröber“ und damit kleiner zu machen. Bei k Stufen erhält man daher eine Folge $G_0, G_1, \dots, G_k$ von kleiner werdenden Graphen, wobei $G_0 = G$. Beginnend mit G_k wird dieser mit einem gewöhnlichen Graphzeichenalgorithmus, wie sie zuvor vorgestellt wurden, gezeichnet und anschließend wird jeder Knoten auf den Stand von G_{k-1} „entfaltet“. Ausgehend von der gerade erhaltenen Zeichnung (bzw. Zuordnung $V \to \mathbb{R}^2$) wird nun G_{k-1} gezeichnet usw. bis der komplette Graph gezeichnet ist [BGKM11]. Auf diese Weise können Graphen in Größenordnungen von 100.000 bis Millionen Knoten ansprechend in ertragbarer Zeit gezeichnet werden [GHK13]. Verfahren ähnlich dieser Art verwenden beispielsweise

- Harel und Koren [HK02], bei denen beispielsweise auch auf den Algorithmus nach Kamada und Kawai zurückgegriffen wird.
- Gajer et al. [GGK04], deren Algorithmus vermutlich subquadratische Laufzeit hat.
- Walshaw [Wal03], der einen veränderten Algorithmus nach Fruchterman und Reingold nutzt und dessen Algorithmus in $O(|V|^2)$ Laufzeit abläuft [Kob12].
- Hachul und Jünger [HJ04] mit ihrem *FM³*-Graphzeichenalgorithmus. Als kräftbasierten Zeichenalgorithmus nutzen sie, wie zuvor beschrieben, die FMM so, dass sie auch insgesamt auf eine Laufzeit in $O(|V| \log|V| + |E|)$ kommen.

3 Algorithmus

Ziel dieser Arbeit ist es die klassischen kräftebasierten Graphzeichenalgorithmen durch Näherung der abstoßenden Kräfte wesentlich zu beschleunigen. Dabei soll die Qualität der so entstandenen Zeichnungen gegenüber der Qualität der Zeichnungen des klassischen Algorithmus nicht wesentlich sinken. Die Veränderungen, die dafür zu vollziehen sind, werden in diesem Kapitel vorgestellt. Anschließend wird gezeigt, dass dies die Laufzeitklasse verbessert, und es wird eine weitere für die Praxis sinnvolle Möglichkeit zur Beschleunigung vorgeschlagen.

3.1 Aufbau

Der neue Algorithmus folgt dem Aufbau des klassischen kräftebasierten Algorithmus, wie er in Algorithmus 1 skizziert ist. Übergeben werden muss nicht mehr nur eine Zuordnung der Knoten auf Punkte in der Ebene, sondern auch eine reelle Zahl größer 0 als Parameter s für die WSPD, die zur internen Berechnung verwendet wird. Bezüglich des Ablaufs wird das Innere der while-Schleife innerhalb dieses Algorithmus verändert. Der wesentliche Schritt ist dort die Berechnung der Kräfte, der in Algorithmus 2 für das Verfahren nach Fruchterman und Reingold schematisiert ist. Der Unterschied zu dem von Eades ist hier hauptsächlich, dass die abstoßende Kraft auf jedes Knotenpaar, also auch auf adjazente Knotenpaare, angewandt wird. Die folgende Schematisierung des angepassten Algorithmus folgt in dem Punkt dem Prinzip von Fruchterman und Reingold. Das Verfahren nach Eades würde sich aber auch leicht auf dieses übertragen lassen, indem man bei der Berechnung der adjazenten Kräften die abstoßende Kraft einmal abzieht.

In Algorithmus 3 ist der neue Ablauf des Inneren der while-Schleife von Algorithmus 1 notiert. Der Ablauf dessen entspricht einer Iteration im Zeichenalgorithmus. Die Teile „Adjazente Kräfte berechnen“ und „Resultierende Kräfte anwenden“ sind dabei unverändert übernommen worden. Einzig der Teil „Abstoßende Kräfte berechnen“ wurde verändert.

Als Speichervariable für die auf einen Knoten v wirkende Kraft wird wie in Algorithmus 2 $v.disp$ verwendet. Mit der ersten foreach-Schleife soll lediglich sichergestellt werden, dass der Verschiebungsvektor in jeder Iteration beim Nullvektor startet und keine Veränderung aus vorherigen Iterationen übernimmt. Anschließend wird ein Split-Tree über die den Knoten zugeordnete aktuelle Punktmenge berechnet. Diese Punktmenge war das Ergebnis der vorherigen Iteration oder ist die initiale Punktzuteilung. Aus dem Split-Tree wird darauffolgend eine WSPD bezüglich s gebildet. Hiernach wird für jeden Split-Tree-Knoten S der Schwerpunkt seiner Punktmenge berechnet und in der Speichervariable $S.schwerpunkt$ gespeichert. Die zweite foreach-Schleife durchläuft nun alle wohlseparierten Paare aus der WSPD. Die beiden Teile eines solchen Paares, die Mengen

Algorithmus 3 : Iteration eines kräftebasierten Zeichenalgorithmus mit WSPD

`// Speichervariable für die wirkenden Kräfte zurücksetzen`
1 **foreach** $v \in V$ **do**
2     $v.disp = \vec{0}$
`// Abstoßende Kräfte berechnen`
3 Berechne einen Split-Tree st über die aktuellen Positionen der Knoten
4 Berechne eine WSPD $wspd$ bezüglich s aus st
5 Berechne die Schwerpunkte aller Split-Tree-Knoten
6 **foreach** wohlsepariertes Paar (A,B) aus $wspd$, wobei A, B Punktmenge $\equiv$ Split-Tree-Knoten **do**
7     $c_A := A.schwerpunkt$
8     $c_B := B.schwerpunkt$
9     $A.abstoßendeKraft = A.abstoßendeKraft + |B| \cdot \vec{F}_{abstoßend}(c_B, c_A)$
10     $B.abstoßendeKraft = B.abstoßendeKraft + |A| \cdot \vec{F}_{abstoßend}(c_A, c_B)$
11 propagiereUndVermerkeAbstoßendeKräfte($st, st.wurzel, 0$)
`// Adjazente Kräfte berechnen`
12 **foreach** $\{u, v\} \in E$ **do**
13     $u.disp = u.disp + \vec{F}_{\text{adjazent}}(v, u)$
14     $v.disp = v.disp + \vec{F}_{\text{adjazent}}(u, v)$
`// Resultierende Kräfte anwenden`
15 **foreach** $v \in V$ **do**
16   Berechne neue Position von v

Algorithmus 4 : propagiereUndVermerkeAbstoßendeKräfte(Split-Tree st, Split-Tree-Knoten stk, Vektor *kraftDarüber*)

Eingabe : Ein Split-Tree st, ein Split-Tree-Knoten stk, der Knoten in st ist, und die Summe aus den vermerkten abstoßenden Kräften der Knoten, die oberhalb dieses Knotens den Pfad zur Wurzel bilden, bezeichnet als *kraftDarüber*

1 $neuerKraftvektor :=$ *kraftDarüber* $+ stk.abstoßendeKraft$
2 **if** stk ist Blatt **then**
3     $stk.referenzpunkt.disp = stk.referenzpunkt.disp + neuerKraftvektor$
4 **else**
5   propagiereUndVermerkeAbstoßendeKräfte($st, stk.linkesKind, neuerKraftvektor$)
      propagiereUndVermerkeAbstoßendeKräfte($st, stk.rechtesKind, neuerKraftvektor$)

A und B, sind dabei Punktmengen. Wie am Ende von Abschnitt 2.4 beschrieben, entspricht jede Punktmenge A und jede Punktmenge B aus einem Paar $\{A, B\}$ einer WSPD der Punktmenge genau eines Split-Tree-Knotens aus dem zugehörigen Split-Tree. Da diese Zuordnung eindeutig ist, können im Algorithmus A und B sowohl als Punktmengen als auch als Split-Tree-Knoten interpretiert werden. Implementiert werden kann das beispielsweise, indem in der WSPD Paare von Split-Tree-Knoten gespeichert werden, die jeweils einen Zeiger auf eine Menge von Punkten haben. Jeder Split-Tree-Knoten S besitzt eine Speichervariable *S.abstoßendeKraft*, in der die in einer Iteration berechneten abstoßenden Kräfte analog zu $v.disp$ für einen Knoten v aufsummiert werden.

Der eigentliche Kern der Veränderung ist nun, dass die abstoßenden Kräfte nicht zwischen zwei einzelnen Knoten bzw. den zugehörigen Punkten dieser Knoten wirken, sondern zwischen zwei Punktmengen A und B, bei denen die Abstoßung zwischen ihren Schwerpunkten berechnet wird. Diese berechnete Kraft wirkt auf jeden Knoten in jeder der beiden Mengen. Diese Zusammenfassung macht deshalb Sinn, da für die beiden Punktmengen durch die WSPD ein Mindestabstand relativ zu ihrer Größe und abhängig von s garantiert wird und nach Definition 2.7 jedes Punktepaar in genau einem Paar der WSPD in verschiedenen Mengen vorkommt. Durch diesen Schritt wird die Berechnung der abstoßenden Kräfte für alle geordneten Knotenpaare $A \times B$ und $B \times A$, das sind $2|A||B|$ viele, durch die Berechnung der abstoßenden Kräfte für zwei geordnete Knotenpaare ersetzt. Das sind die Paare $(A.schwerpunkt, B.schwerpunkt)$ und $(B.schwerpunkt, A.schwerpunkt)$. Diese Approximation macht eine Gewichtung notwendig, denn ein Schwerpunkt muss so viel Gewicht haben, wie ihm aufgrund der Anzahl der Punkte, die er vertritt, zusteht. Deshalb wird im Algorithmus die berechnete abstoßende Kraft mit der Mächtigkeit der jeweils anderen Menge, das ist die kraftausübende Menge, gewichtet.

Würde man die so errechnete Kraft jedem einzelnen Punkt aus A und B gleich eintragen, so müssten $|A| + |B|$ Operationen durchgeführt werden. Über alle Paare aus der WSPD, die über die zur Knotenmenge V gehörende Punktmenge gebildet wurde, wären das $\sum_{\{A,B\} \in \text{WSPD}} |A| + |B|$ viele, was im schlimmsten Fall $\Theta(|V|^2)$ viele sein können [CK95]. Deswegen wird die für A bzw. B berechnete abstoßende Kraft in der Variable *A.abstoßendeKraft* bzw. *B.abstoßendeKraft* gespeichert und nach Durchlauf aller Paare aus der WSPD effizient mittels eines Split-Tree-Durchlaufs zu den Blättern propagiert. Dies geschieht durch den Aufruf der rekursiven Methode „propagiereUndVermerkeAbstoßendeKräfte“, deren Ablauf in Algorithmus 4 skizziert ist. Einmal für die Wurzel des Split-Trees aufgerufen werden dort für alle Wurzel-Blatt-Pfade die gespeicherten Werte aufsummiert und anschließend auf die Punkte, welche die Knoten des Graphen repräsentieren, übertragen und dort als aktuell wirkende Kraft abgespeichert. Da jedes Blatt eines Split-Trees genau einem Punkt zugeordnet werden kann, wird dieser mit $stk.referenzpunkt$ adressiert.

Dass bei dieser Näherung der abstoßenden Kräfte die Qualität der Zeichnung gegenüber dem Original nicht wesentlich schlechter ist, wird in Kapitel 4 empirisch gezeigt.

3.2 Laufzeitanalyse

Behauptung 3.1. Der gerade aufgezeigte Algorithmus gibt für einen Graphen $G = (V, E)$ eine Zuordnung $V \to \mathbb{R}^2$ in $O(|V| \log |V| + |E|)$ Zeit zurück, wenn der zugrunde liegende kräftebasierte Graphzeichenalgorithmus in höchstens konstant vielen Iterationen terminiert.

Beweis. Eine Zuweisung von initialen Punkten zu den Knoten ist offensichtlich in $O(|V|)$ Zeit möglich. Im Inneren der while-Schleife (in Algorithmus 3 aufgezeichnet) werden mehrere Operationen hintereinander ausgeführt. Das sind:

- Das Zurücksetzen der Speichervariable $v.disp$ für alle $v \in V$, das in $O(|V|)$ Zeit möglich ist
- Die Berechnung der abstoßenden Kräfte, die in $O(t)$ Zeit möglich ist, wobei t ein noch zu bestimmender Term ist
- Die Berechnung der adjazenten Kräfte, die in $O(|E|)$ Zeit möglich ist
- Die Berechnung der neuen Positionen für alle $v \in V$, die in $O(|V|)$ möglich ist

Insgesamt ist das Innere der while-Schleife daher in $O(t + |V| + |E|)$ Zeit möglich. Da diese Operationen nach Annahme höchstens konstant viele mal ausgeführt werden, liegt die Gesamtlaufzeit des Algorithmus in derselben Laufzeitklasse.

Es gilt noch den Term t zu bestimmen. An den Schritten zur Berechnung der abstoßenden Kräfte in Algorithmus 3 orientiert, ergeben sich für die einzelnen Teile folgende Laufzeiten:

- Das Berechnen des Split-Trees ist nach Satz 2.9 in $O(|V| \log |V|)$ Zeit möglich.
- Das Berechnen der WSPD ist nach Satz 2.9 für $d = 2$ und ein gegebenes, konstantes s in $O(|V|)$ Zeit möglich.
- Das Berechnen der Schwerpunkte im Split-Tree kann rekursiv von der Wurzel erfolgen. Dabei errechnet sich der Schwerpunkt für einen Split-Tree-Knoten als Summe aus den Schwerpunkten seiner beiden Kinder gewichtet mit den Punktmengenmächtigkeiten dieser Kinder und geteilt durch seine Punktmengenmächtigkeit (das ist die Summe aus den Punktmengenmächtigkeiten seiner beiden Kinder). Die Werte der Punktmengenmächtigkeit und Schwerpunkte werden so durch den Split-Tree propagiert, dass jeder Knoten einmal durchlaufen wird, wobei bei jedem einzelnen die Berechnung offensichtlich in konstanter Zeit möglich ist. Somit hängt dies von der Größe des Split-Trees ab. Diese ist nach Satz 2.9 in $O(|V|)$ und eine Laufzeit für diesen Schritt damit in $O(|V|)$ Zeit möglich.
- In der „foreach wohlsepariertes Paar“-Schleife wird nach Satz 2.9 für $d = 2$ und ein gegebenes, konstantes s die Operation in der Schleife $O(|V|)$ mal ausgeführt. Wenn beim Durchlauf der Schleife der Zugriff auf A und B in konstanter Zeit möglich ist, was durch die Speicherung von Zeigern auch kein Problem sein sollte, kann das

Innere der Schleife offenbar in konstanter Zeit ausgeführt werden. Somit ist eine Laufzeit von $O(|V|)$ für diese Schleife möglich

- Der verbleibende Schritt, die errechneten abstoßenden Kräfte für jeden Knoten zusammenzufassen, geschieht mithilfe des rekursiven Algorithmus 4. Diese Methode wird für jeden Knoten genau einmal aufgerufen und die Ausführung jeder Operation darin ist offensichtlich in konstanter Zeit möglich, insbesondere auch die Überprüfung, ob *stk* ein Blatt ist. Damit ergibt sich analog zur Berechnung der Schwerpunkte eine Laufzeit von $O(|V|)$.

Daher ist $t = |V| \log |V|$ und die Gesamtlaufzeit des Algorithmus liegt in der Laufzeitklasse $O(|V| \log |V| + |E|)$. □

Ein Term zur asymptotischen Begrenzung der Zahl der Iterationen sei I. Terminiert der zugrundeliegende kräftebasierte Graphzeichenalgorithmus nicht in konstant vielen Iterationen, so verbessert diese Änderung die Laufzeitklasse des Algorithmus weiterhin. In diesem Fall wird sie von $O(I|V|^2)$ auf $O(I(|V| \log |V| + |E|))$ verbessert.

3.3 Eine einfache Beschleunigungstechnik

Dieses Verfahren liegt zwar in einer besseren Laufzeitklasse als das herkömmliche Verfahren, doch wird auch in jeder Iteration Zeit benötigt, um den Split-Tree und die WSPD zu erstellen, bevor die abstoßenden Kräfte berechnet werden können. Dabei ist es möglicherweise gar nicht immer nötig in jeder Iteration den Split-Tree und die WSPD aus der vorherigen Iteration zu verwerfen und neu zu berechnen. Deswegen wird hier nun der Gedanke umgesetzt, dass nur in manchen Iterationen der Split-Tree und die WSPD neu aufgebaut werden und in allen anderen Iterationen der vorhandene Split-Tree und die vorhandene WSPD verwendet werden, bei denen eventuell noch Kenngrößen wie die Schwerpunkte der Split-Tree-Knoten aktualisiert werden. In der ersten Iteration muss ein Split-Tree und eine WSPD neu berechnet werden. In jeder folgenden Iteration wird eine Funktion $f_{\text{Neukonstruktion}}$ aufgerufen, die in jeder Iteration einen Wahrheitswert zurückgibt, welcher aussagt, ob in dieser Iteration der Split-Tree und die WSPD neu erstellt werden sollen oder nicht. Eine Funktion dieser Art wird im Folgenden *Neukonstruktionsfunktion* genannt. In dieser Funktion sind Zugriffe auf Größen des Graphzeichenalgorithmus wie die aktuelle Iterationszahl, den Split-Tree oder die WSPD möglich. Sie sollte allerdings in $O(|V| \log |V| + |E|)$ Zeit ablaufen, damit sich die Gesamtlaufzeitklasse des Algorithmus nicht erhöht.

Im zuvor beschriebenen Algorithmus wird in jeder Iteration beides neu berechnet. Das kann als Vorhandensein einer Neukonstruktionsfunktion interpretiert werden, die immer *wahr* zurückgibt. Dieses Modell mit Neukonstruktionsfunktionen stellt daher eine Verallgemeinerung dar. Diese triviale Neukonstruktionsfunktion wird im Folgenden mit $f^{\text{immer-neu}}_{\text{Neukonstruktion}}$ bezeichnet. Viele Umsetzungen der Neukonstruktionfunktion sind denkbar, ich habe zwei weitere Ansätze für diese Arbeit näher untersucht. Beim Verwenden eines Split-Trees und einer WSPD aus einer vorherigen Iteration ist es wichtig anzumerken, dass die Eigenschaften, die die WSPD und der Split-Tree garantieren,

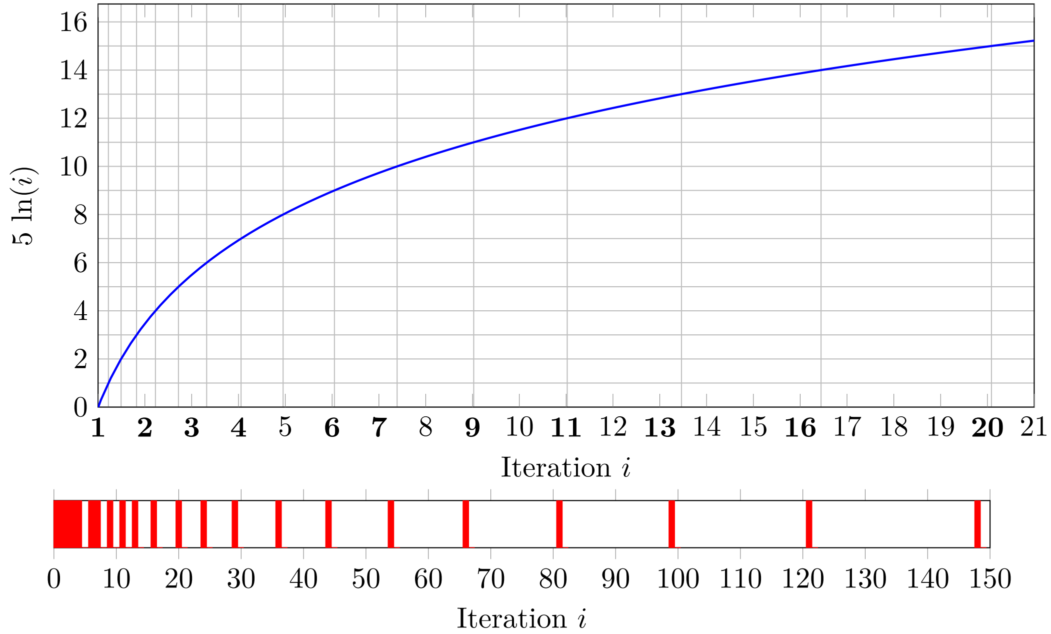


Abb. 3.1: Veranschaulichung von $f^{\text{Log}}_{\text{Neukonstruktion}}$ für $a = 5$ und $b = 0$. In der oberen Grafik sind die Werte des 5-fachen $\ln(i)$ eingezeichnet. Das Gitter zur Orientierung im Hintergrund verläuft bei den Werten, bei denen ein ganzzahliger Wert erreicht wird. Iterationen, in denen Split-Tree und WSPD neu konstruiert werden, sind fett hervorgehoben. In der unteren Grafik ist bei den Iterationen bis 150, bei denen selbige Funktion *wahr* zurückgibt, ein roter Balken eingezeichnet.

wie den s-fachen Mindestabstand zweier Paare der WSPD, also die Wohlsepariertheit bezüglich s, nicht mehr gewährleistet sind. Deshalb ist der erste weitere Ansatz, dass $f^{\text{min-dist}}_{\text{Neukonstruktion}}$ genau dann *wahr* zurückgibt, sobald wenigstens ein Paar aus der WSPD einen Mindestabstand von $c \cdot s \cdot r_{\max}$ nicht mehr einhält. Dabei ist c ein wählbarer Parameter zwischen 0 und 1, s das s aus der WSPD und $r_{\max}$ der größere Radius der beiden kleinsten die Bounding-Boxen der Punktmengen des betrachteten Paares umschließenden Kreise.

Der zweite weitere Ansatz basiert darauf, dass sich die Positionen der Knoten bei so einem Graphzeichenalgorithmus in den ersten Iterationen stark ändern, während in späteren Iterationen hauptsächlich kleine Änderungen zu beobachten sind. Von daher scheint es sinnvoll den Split-Tree und die WSPD zu Beginn häufig neu zu berechnen und mit zunehmender Iterationszahl immer seltener. Dies wird quasi heuristisch nur in Abhängigkeit von der aktuellen Iterationszahl durchgeführt, ohne die tatsächliche Qualität des Split-Trees und der WSPD zu überprüfen, was wieder eine gewisse Zeit dauern würde. Eine logarithmische Funktion steigt für Werte größer 0 erst stark und wächst dann aber immer langsamer. Diese Eigenschaft wird in der folgenden Definition der zweiten Neukonstruktionsfunktion verwendet. Es sei:

$$f^{\text{Log}}_{\text{Neukonstruktion}} = (\lfloor a \ln(b+i) \rfloor \neq \lfloor a \ln(b+i+1) \rfloor) \tag{3.1}$$

Dabei ist i die aktuelle Iteration (eine natürliche Zahl), $a \in \mathbb{R}^+$ ein zu wählender Parameter, der die Häufigkeit der Neukonstruktionen bestimmt (je kleiner, desto weniger oft wird *wahr* zurückgegeben), und $b \in \mathbb{R}_0^+$ ein zu wählender Parameter, der die Funktionswerteabfolge verschiebt (eine Art Phasenverschiebung). Anschaulich gesprochen wird für eine Iteration i genau dann *wahr* zurückgegeben, wenn der mit a und b modifizierte ln im Intervall bis zur nächsten gültigen Iterationszahl eine Ganzzahl erreicht oder passiert (die Vorkommazahl sich ändert). Eine grafische Veranschaulichung ist in Abb. 3.1 zu sehen.

4 Ergebnisse

Um den in Kapitel 3 vorgestellten Vorschlag zur Änderung eines kräftebasierten Graphzeichenalgorithmus in der Praxis zu überprüfen, habe ich diesen in eine Implementierung des Algorithmus von Fruchterman und Reingold integriert. Die Implementierung ist in Java unter Verwendung des Frameworks *JUNG* (Java Universal Network/Graph Framework) [JUN] erfolgt. Getestet wurde hauptsächlich mit den *Rome-Graphen* [Rom]. Das ist eine große und vielseitige Sammlung von Graphen der Größe 10 bis 100 Knoten. Wie bereits angesprochen, eigenen sich die klassischen Graphzeichenalgorithmen in ihrer bloßen Form hauptsächlich für Graphen bis etwa 100 Knoten, sodass dies eine geeignetes Testsammlung darstellt. Nach Entfernen des Ordners „bad", also den „schlechten" Graphen, umfasst die Sammlung der Rome-Graphen noch 11.528 einzelne Graphen, die alle aus einer Zusammenhangskomponente bestehen.

Getestet wurde an einem handelsüblichen Notebook mit Intel Core i5-3210M mit 2 mal 2,50GHz (wobei die Berechnungen jedoch nicht parallelisiert wurden) und 6 GB RAM. Zur Bestimmung des Zeitaufwandes für die jeweiligen Algorithmen wurde die CPU-Zeit des Graphzeichenthreads in Java an entsprechenden Stellen gemessen.

4.1 Vergleich von FRLayout mit FRLayout2

Das Framework JUNG enthält unter anderem verschiedene Graphzeichenalgorithmen. Einer davon ist eine Implementierung des Algorithmus von Fruchterman und Reingold, im Folgenden nach der dortigen Benennung mit *FRLayout* bezeichnet, und ein anderer die Implementierung des Algorithmus von Fruchterman und Reingold mit den alternativen Funktionen zur Größe der Kraft (FR2), im Folgenden mit *FRLayout2* bezeichnet. Um zu überprüfen, welcher die besseren Ergebnisse liefert, habe ich diese beiden unveränderten Algorithmen die Rome-Graphen zeichnen lassen. Da bei jeder Zeichnung eine zufällige initiale Knotenpositionierung erfolgt, wurde jeder der 11.528 Graphen fünf mal je Algorithmus gezeichnet. Ergebnisse für die CPU-Zeit und eine Auswahl von Qualitätskriterien sind in Abb. 4.1 zu sehen.

FRLayout2 ist geringfügig schneller als FRLayout. Als wichtigste Maße zur Bewertung der Qualität einer ausgegebenen Zeichnung werden hier und im Folgenden die Anzahl der Kantenkreuzungen und die Standardabweichung der Kantenlängen betrachtet. Diese Kriterien wurden auch in vorangegangenen Arbeiten zur Qualitätsbewertung genutzt [BHR96, FLM95]. Die von FRLayout2 gelieferten Zeichnungen haben rund ein Drittel mehr Kantenkreuzungen als die von FRLayout gelieferten Zeichnungen. Die Standardabweichung der Kantenlänge ist bei FRLayout2 geringer, dort ist jedoch der Knotenabstand auch kleiner, was eine „schlechtere" Entzerrung des Ganzen nahelegt. Das lässt insgesamt vermuten, dass FRLayout nach diesen Kriterien der geringfügig bessere

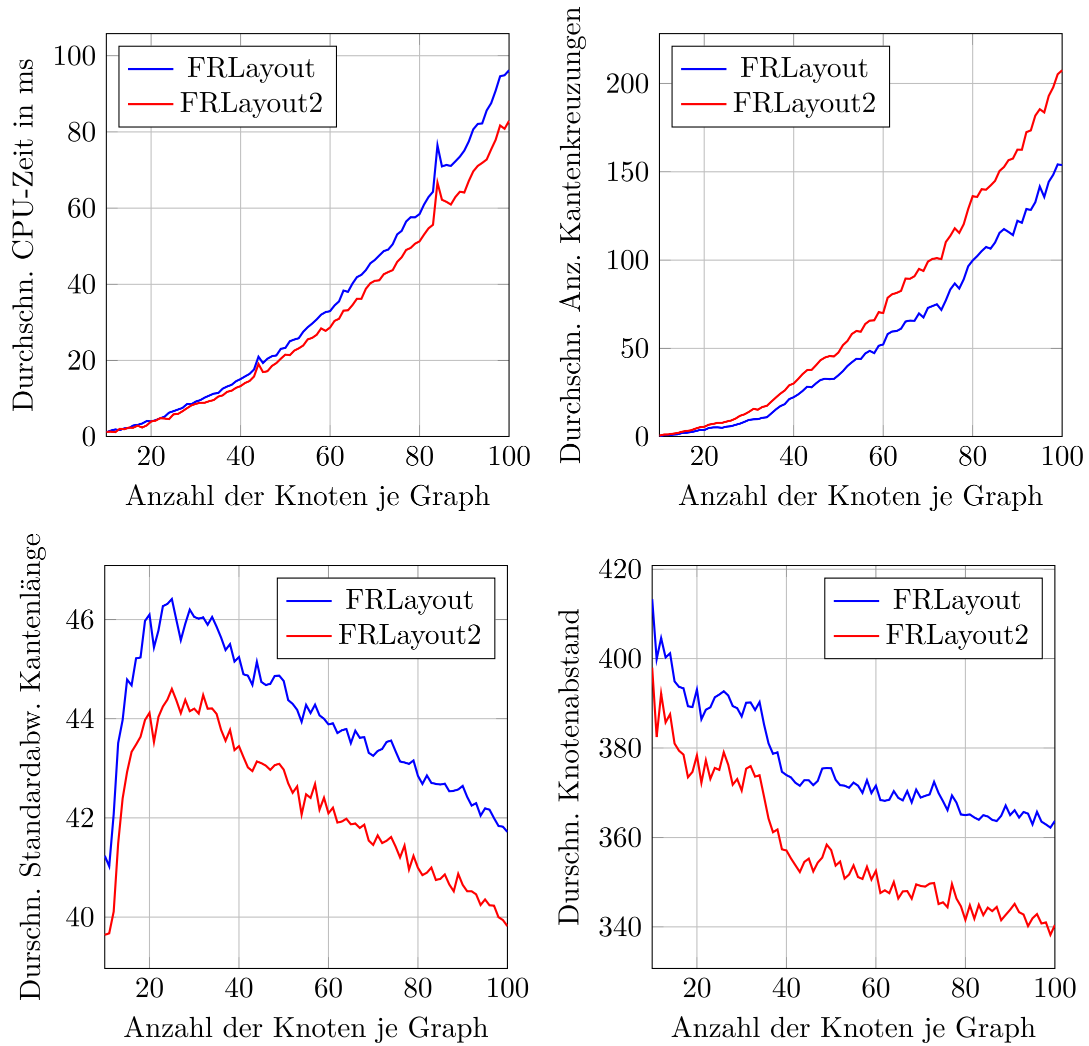


Abb. 4.1: Vergleich von FRLayout und FRLayout2 anhand von CPU-Zeit und einer Auswahl von Qualitätskriterien über Zeichnungen der Rome-Graphen zusammengefasst nach Knotenzahl.

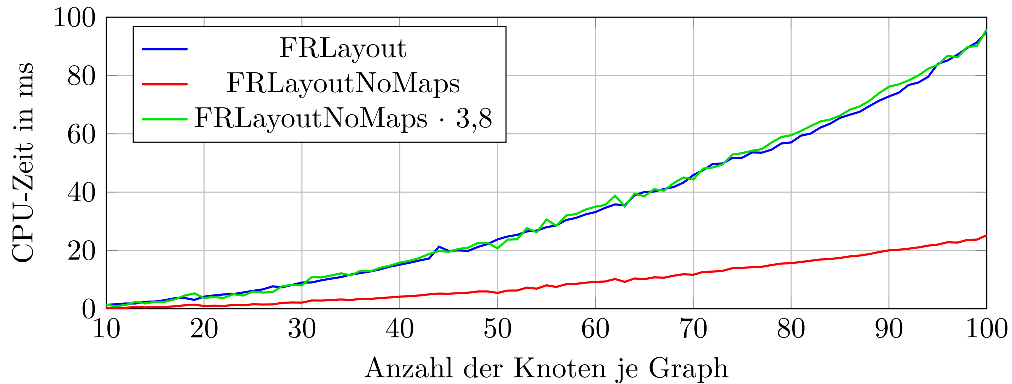


Abb. 4.2: Durchschnittliche CPU-Zeit zum Zeichnen der Rome-Graphen in einem 5-fachen Durchlauf zusammengefasst nach Knotenzahl.

Algorithmus ist. Deswegen wird im Folgenden nur noch FRLayout behandelt und gemäß den vorgeschlagenen Änderungen angepasst. Bei der Betrachtung einer geringen Menge von Zeichnungen konnte ich auch keinen klar besseren Algorithmus ausmachen, sodass für diesen Vergleich kein abschließendes Ergebnis ausgegeben wird.

4.2 Vergleich von FRLayout mit FRLayoutNoMaps

Bei den vorangegangenen Analysen zur Laufzeitklasse ist immer davon ausgegangen worden, dass innerhalb des Graphzeichenalgorithmus ein Zugriff von einem Knoten auf den Punkt, der ihm als Position zugeordnet ist, in einer Zeit von $O(1)$ möglich ist. In der Implementierung des JUNG-Pakets werden die Knoten in einer Datenstruktur ohne die Zuordnung der Positionen gespeichert. Die Positionen (Punkte) werden als Werte einer Map gespeichert, als deren Keys die Knoten fungieren. Der Zugriff mit einer Map über die Schlüsselmenge V kann zwar mit Hashing in einer erwarteten Zeit von $O(1)$ erfolgen, was jedoch mit einer Worst-Case-Zeit in $O(|V|)$ verbunden ist [CLRS09]. Zudem kann selbst unter der wohl berechtigten Annahme eines in der Praxis üblichen Zugriffs in $O(1)$ die Bearbeitungsdauer einen konstanten Faktor länger dauern. Genau dies scheint hier auch der Fall zu sein. Auf die iterationsinternen Verschiebungswerte ($v.disp$ in den vorangegangenen Algorithmen) wird ebenfalls mit Maps zugegriffen. In den Implementierungen der Graphen in JUNG werden Maps auch zur Zuordnung der Kanten auf ihre inzidenten Knoten verwendet.

Um zu überprüfen, ob dies in der Praxis einen relevanten Einfluss auf die Laufzeit hat, habe ich eine Java-Klasse *FRLayoutNoMaps* erstellt, die die Klasse FRLayout so abändert, dass beim Graphzeichenvorgang keine Maps und JUNG-Graphen mehr verwendet werden. Gearbeitet wird dort mit Tripeln (3-Tupeln) für die Knoten und Kanten. Ein Knoten-Tripel besteht aus dem eigentlichen Knotenobjekt (z.B. ein Name), der aktuellen Position und dem iterationsinternen Verschiebungswert. Ein Kanten-Tripel besteht aus dem eigentlichen Kantenobjekt (z.B. ein Name) und den beiden Knoten-Tripeln der

beiden inzidenten Knoten. Gegeben ein Tripel ist damit der Zugriff auf die zusammengehörigen Daten in $O(1)$ möglich.

In Abb. 4.2 ist die CPU-Zeit zum Zeichnen der Rome-Graphen für FRLayout und FRLayoutNoMaps aufgezeichnet. FRLayout benötigt durchgehend fast 4 mal mehr Zeit. Die CPU-Zeit von FRLayoutNoMaps multipliziert mit 3,8 liegt sehr nahe bei der von FRLayout. Deshalb bauen die folgenden Algorithmen auf FRLayoutNoMaps auf.

4.3 Vergleich von FRLayoutNoMaps mit FRLayoutNoMapsNoFrame

Eine weitere große Schwäche ist der Rahmen (frame), der im Algorithmus von Fruchterman und Reingold vorgeschlagen wird [FR91]. In ihrer Arbeit schreiben sie auch, dass sie die Kräfte oft so wählen, dass die Knoten in der Praxis normalerweise nicht den Rahmen erreichen und die Zeichnung am Ende so skalieren, dass sie den Rahmen ausfüllt. In der JUNG-Implementierung werden die Ränder jedoch wohl meist erreicht, sodass es an den Rändern zu einer unübersichtlichen Ansammlung von Knoten und Kanten kommt. Man könnte erwarten, dass die Knoten am Rand alles mehr zurück in die Mitte „drücken“ und so dennoch eine ansprechende Zeichnung zurückgegeben wird. Die zurückgegebene Zeichnung ist tatsächlich aber nicht sehr ansprechend. Besser wäre es, keinen Rahmen zu verwenden, damit sich die Zeichnung unbeschränkt entfalten kann und sie am Ende zurück auf die vorgesehene Zeichenfläche zu skalieren. Diese Anpassung habe ich in der Java-Klasse *FRLayoutNoMapsNoFrame* implementiert. In Abb. 4.3 ist ein Beispiel für einen Graphen, der mit der Variante mit und der Variante ohne Rahmen gezeichnet wurde. Der Rahmen ist klar erkennbar und zwingt dem Graphen seine eigene, quadratische Form auf. Die Zeichnung ohne Rahmen hat 2 Kantenkreuzungen, während die mit Rahmen hier 13 Kantenkreuzungen hat. Dieser numerische Unterschied ist in diesem Beispiel besonders groß.

Die Skalierung am Ende funktioniert hier gut, weil der Graph nur aus einer Zusammenhangskomponente besteht. Besteht ein Graph aus mehr als einer Zusammenhangskomponente, so hält der Rahmen den Graphen dennoch beisammen, während sich die Komponenten bei einer Variante ohne Rahmen ungehalten weit voneinander entfernen, sodass bei einer Rückskalierung große Lücken klaffen. Eine Lösung hierfür schlagen bereits Kamada und Kawai [KK89] vor, auf die von Frucherman und Reingold [FR91] verwiesen wird. Ein Graph $G = (V, E)$ sollte zu Beginn in seine Zusammenhangskomponenten zerlegt werden, was in $O(|V| + |E|)$ Zeit möglich ist [KN12], und jede Komponente separat auf einer Zeichenfläche gezeichnet werden. Dabei erhält jede Zusammenhangskomponente einen Teil der verfügbaren Zeichenfläche, sodass die Größe dieses Teils proportional zur entsprechenden Knotenzahl ist.

Bei meiner Umsetzung in FRLayoutNoMapsNoFrame bin ich davon ausgegangen, dass die Zeichenfläche quadratisch ist, eine Anwendung auf insbesondere Rechtecke sollte aber auch kein Problem darstellen. Zu Beginn werden die Knoten nach Zusammenhangskomponenten gruppiert und ihnen ein Rechteck mit einer Größe proportional zu ihrer Knotenzahl zugewiesen. Ich habe hier die einfache Lösung gewählt die Zeichenfläche nur

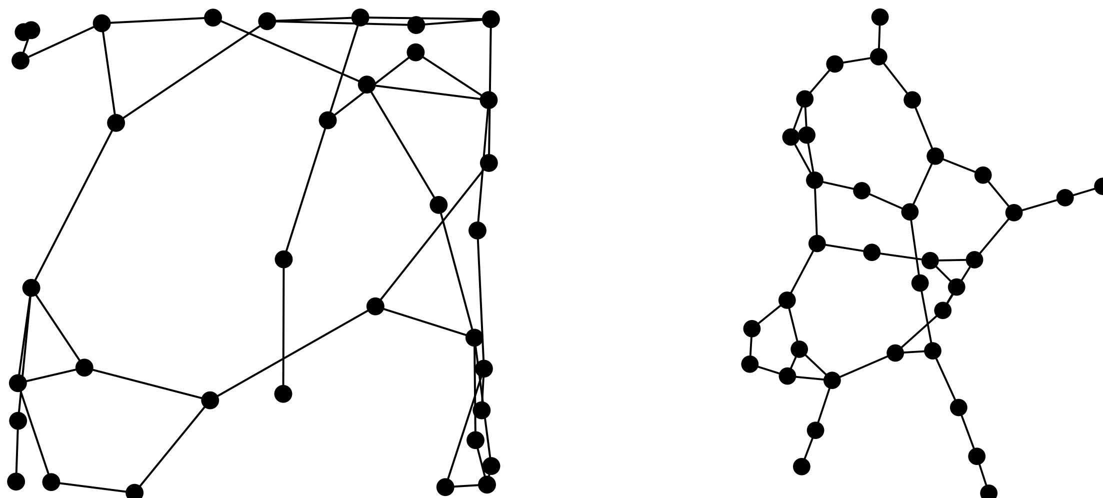


Abb. 4.3: Die linke Zeichnung wurde mit FRLayoutNoMaps und die rechte mit FRLayoutNoMapsNoFrame erzeugt. In beiden Fällen wurde der Rome-Graph grafo5419.35 mit denselben initialen Startpunkten für die 35 Knoten gezeichnet.

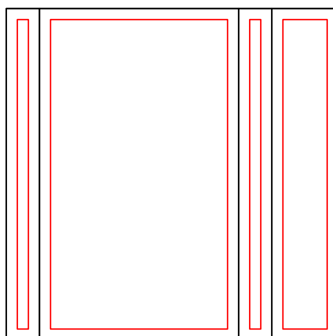


Abb. 4.4: Mögliche Aufteilung einer quadratischen Zeichnfläche in Rechtecke für jede Zusammenhangskomponente gemäß ihrer Größe in FRLayoutNoMapsNoFrame.

in einer Dimension zu teilen. Jede Teilzeichenfläche hat daher dieselbe Höhe wie die Gesamtzeichenfläche, allerdings eine variable Breite. Dieses Prinzip ist schematisiert in Abb. 4.4 zu sehen. Jede Komponente darf sich frei entfalten, wird am Ende jedoch in ihr Rechteck skaliert, wobei in jedem Rechteck ein Abstand zum Rand eingehalten wird, um Abstände groß genug zu halten (Rotes Rechteck in der Abbildung). Schwäche dieses Vorgehens ist, dass teils langgezogene, schmale Rechtecke als Zeichenfläche dienen. Zwar lasse ich x- und y-Koordinaten in $O(|V|)$ Zeit vertauschen, falls eine fertige Teilzeichnung mehr breit als hoch ist, doch wäre es besser, wenn auch kleine Teilrechtecke annähernd quadratisch wären. Vermutlich gibt es hier bereits deutlich bessere Ansätze zur Aufteilung einer Fläche, die man hier stattdessen nutzen könnte.

Die Ergebnisse eines fünffachen Durchlaufs aller Rome-Graphen mit den Algorithmen FRLayoutNoMaps, in dem die Rahmen unverändert verwendet werden, und FRLay-

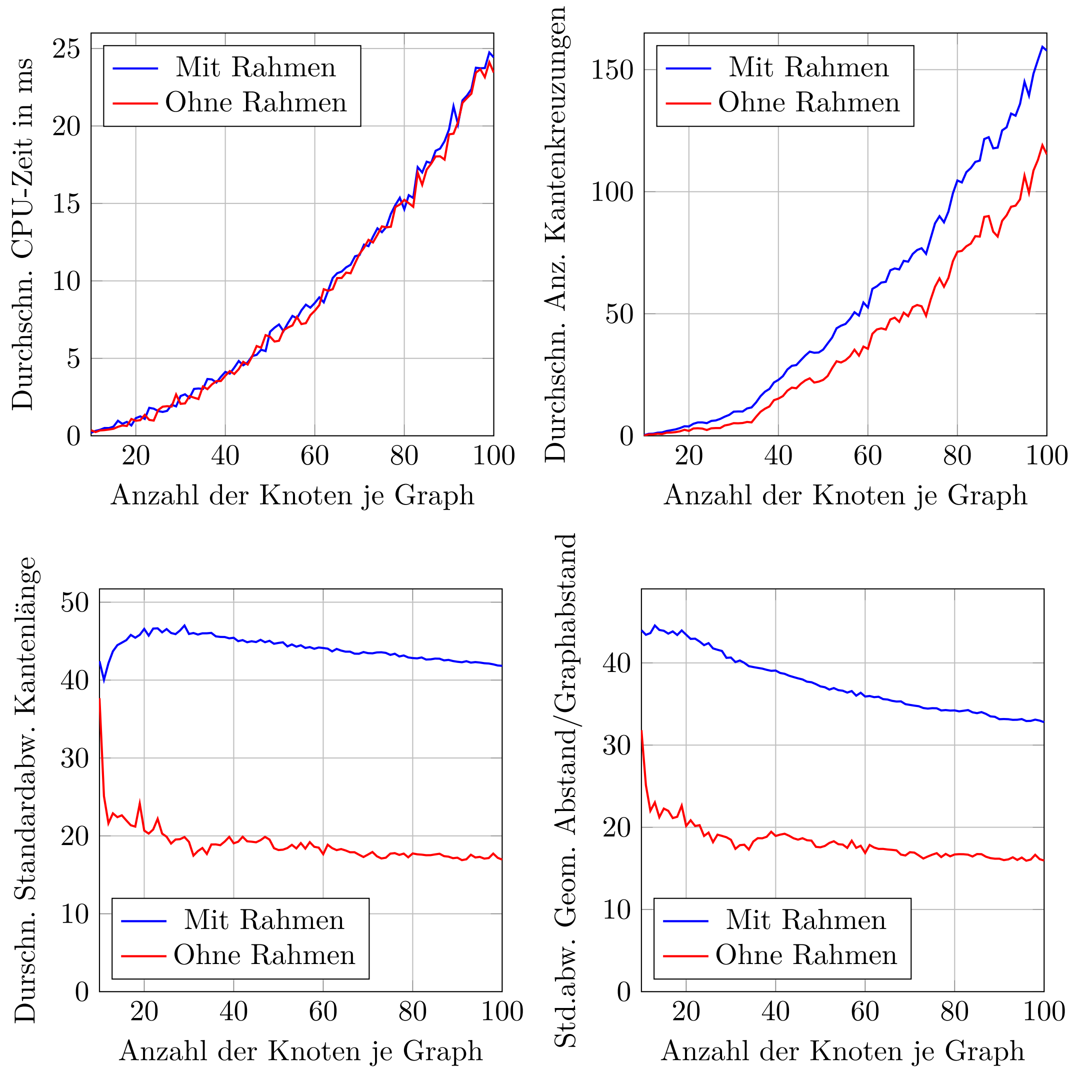


Abb. 4.5: Vergleich von FRLayoutNoMaps (Mit Rahmen) und FRLayoutNoMapsNoFrame (Ohne Rahmen) anhand von CPU-Zeit und einer Auswahl von Qualitätskriterien über Zeichnungen eines 5-fachen Durchlaufs der Rome-Graphen zusammengefasst nach Knotenzahl.

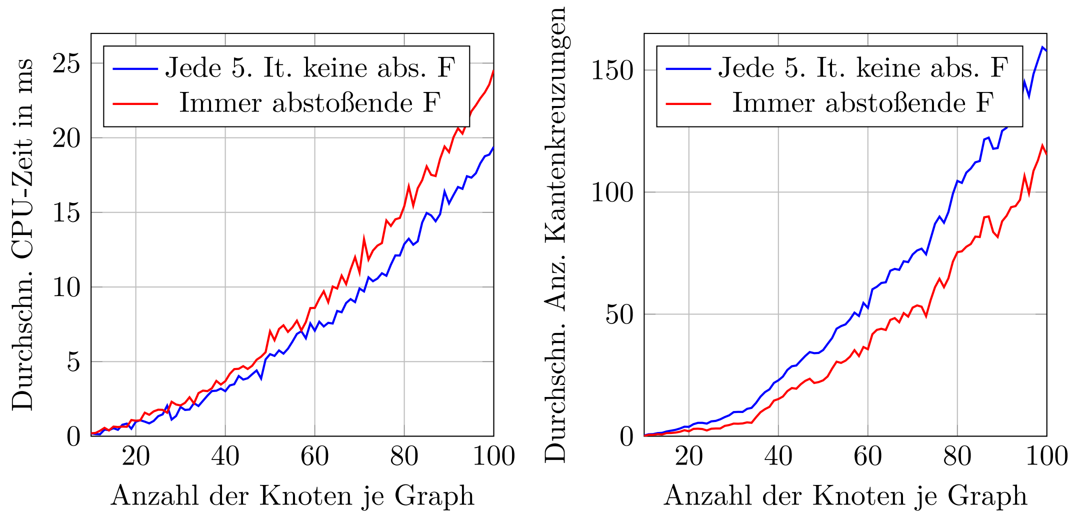


Abb. 4.6: Ergebnisse eines 5-fachen Durchlaufs der Rome-Graphen mit FRLayoutNoMapsNoFrame (Immer abstoßende F) und FRLayout++ (Jede 5. It. keine abs. F).

outNoMapsNoFrame bestätigen die Vermutungen. Eine Auswahl der Ergebnisse ist in Abb. 4.5 abgebildet. Bei der benötigten Zeit ändert sich so gut wie nichts, die Qualität steigt jedoch signifikant. Es entstehen weniger Kantenkreuzungen, die Standardabweichung der Kantenlängen ist nur noch etwa halb so hoch und auch die mittlere Standardabweichung des Verhältnisses von geometrischem Abstand in der Zeichenfläche zu graphentheoretischer Distanz zweier Knoten eines Graphen halbiert sich etwa.

4.4 Vergleich von FRLayoutNoMapsNoFrame mit FRLayout++

Außerdem schlagen Fruchterman und Reingold vor in jeder fünften Iteration keine abstoßenden Kräfte zu berechnen, um lokale Minima zu verlassen und damit die Zahl der Kantenkreuzungen zu reduzieren. In der JUNG-Implementierung wird dies nicht getan. Ich habe FRLayoutNoMapsNoFrame bearbeitet, dieses Aussetzen in jeder fünften Iteration zugefügt und diesen Algorithmus *FRLayout++* genannt. Von Interesse ist das Abschneiden dieses Algorithmus gegenüber dem vorherigen Algorithmus FRLayoutNoMapsNoFrame, in dem in jeder Iteration die abstoßenden Kräfte berechnet werden. Wie in Abb. 4.6 zu sehen ist, können die Kantenkreuzungen so noch geringfügig gesenkt werden (ca. 10% für Graphen bis etwa 35 Knoten, dann ca. 7,5%) und als netter Nebeneffekt sinkt die Laufzeit noch in geringem Maße. Auf andere Qualitätsmerkmale hat diese Anpassung keinen merklich großen Einfluss.

4.5 Vergleich von FRLayout++ mit Varianten von FRLayout++WSPD

Die Implementierung des in Kapitel 3 vorgeschlagenen Algorithmus auf Grundlage des herkömmlichen Zeichenalgorithmus FRLayout++ sei *FRLayout++WSPD*. Verschiedene Varianten von FRLayout++WSPD unterscheiden sich im Parameter s für die WSPD oder in der Neukonstruktionsfunktion. Um die beste(n) Kombination(en) aus s und Neukonstruktionsfunktion zu finden, habe ich in einem einfachen Test aller Rome-Graphen FRLayout++ und verschiedene Varianten von FRLayout++WSPD verglichen. Die getesteten Varianten von FRLayout++WSPD waren eine Kombination aus:

- Diesen Neukonstruktionsfunktionen:
  - $f^{\text{immer-neu}}_{\text{Neukonstruktion}}$
  - $f^{\text{min-dist}}_{\text{Neukonstruktion}}$ mit $c \in \{0,6; 0,8; 0,9; 0,95\}$
  - $f^{\text{Log}}_{\text{Neukonstruktion}}$ mit $a \in \{1; 2; 4; 8; 16\}$ und $b \in \{0; 5\}$
- Dem Vorgehen, dass, wenn die Neukonstruktionsfunktion `falsch` zurückgibt, entweder immer die Schwerpunkte aller Paare aus der WSPD aktualisiert wurden oder dies nie geschah. (Hinfällig bei $f^{\text{immer-neu}}_{\text{Neukonstruktion}}$)
- Einem $s \in \{0,015625; 0,03125; 0,0625; 0,125; 0,25; 0,5; 1; 2; 4; 8; 16\}$

Das sind insgesamt 319 verschiedene Varianten. Dabei ist anzumerken, dass in der JUNG-Implementierung die Zahl der Iterationen von der eingegebenen Zeichenfläche abhängt, bei gegebener Zeichenfläche also konstant ist. Um das unabhängig davon zu machen, habe ich den Code so geändert, dass der Algorithmus terminiert, wenn die Veränderungen der Knotenpositionen im Mittel unter einen gewissen Schwellenwert fallen (sich kaum noch etwas an der Knotenanordnung ändert). Tatsächlich haben die verschiedenen Algorithmen auch hier dieselbe Iterationszahl benötigt. Das liegt daran, dass die Implementierung des Simulated Annealing (die Temperatur) die Veränderungen in späteren Iterationen so sehr einschränkt, dass die verschiedenen Algorithmen in derselben Iteration abhängig von der Größe des gewählten Schwellenwerts terminieren. Ob das relativ einfache Simulated Annealing mit einer globalen Temperatur, die in jeder Iteration sinkt, sinnvoll und die konkrete Implementierung dessen in den JUNG-Algorithmen gut gelungen ist, habe ich nicht untersucht. Mit verbesserten Verfahren zur Begrenzung der Änderung haben sich beispielsweise Frick et al. [FLM95] beschäftigt. Daher habe ich diese Veränderung wieder rückgängig gemacht und das ursprüngliche Abbruchkriterium gelassen.

Bei dem Test hat sich herausgestellt, dass das aktualisieren der Schwerpunkte in Iterationen, in denen Split-Tree und WSPD nicht neu berechnet werden, Sinn macht. Zwar erhöht es die Laufzeit etwas, doch werden die Ergebnisse dadurch wesentlich besser. Alle im Folgenden erwähnten Varianten verwenden diese Schwerpunktaktualisierung. Außerdem hat sich herausgestellt, dass sich das Verwenden von s-Werten größer als 1 zumindest in dieser Implementierung und dieser Graphengrößenordnung kaum lohnt.

Algorithmus	CPU-Zeit in ms	Kanten-kreuz.	Stdabw. Kantenl.	% Zeitgw. pro % mehr K.krz.
FRLayout++	6,316	31,63	20,42	0
$4\ln(0+i)$, $s = 0,0625$	3,006	34,76	21,17	5,29
$< 0,6 \cdot s \cdot r_{max}$, $s = 0,03125$	9,151	34,06	21,10	-5,85
Jede Iter. neu, $s = 0,03125$	9,188	34,06	21,10	-5,92

Tab. 4.1: Übersicht über die Mittelwerte einiger Qualitätskriterien beim einfachen Durchlauf der Rome-Graphen mit den Algorithmen FRLayout++ (erste Datenzeile), FRLayout++WSPD mit $f^{\text{Log}}_{\text{Neukonstruktion}}$ mit $a = 4$, $b = 0$ und $s = 0,0625$ (zweite Datenzeile), FRLayout++WSPD mit $f^{\text{min-dist}}_{\text{Neukonstruktion}}$ mit $b = 0,6$ und $s = 0,03125$ (dritte Datenzeile) und FRLayout++WSPD mit $f^{\text{immer-neu}}_{\text{Neukonstruktion}}$ mit $s = 0,03125$ (letzte Zeile).

Die Algorithmen mit s-Werten größer als 1 verbessern die Qualität zwar geringfügig, aber gleichzeitig erhöhen sie die CPU-Zeit des Algorithmus unverhältnismäßig stark. Der geringe Qualitätsgewinn steht damit in keinem guten Verhältnis zum Geschwindigkeitsverlust. Geeigneter erscheinen s-Werte bis etwa 1. Bei kleiner als etwa 1 werdenden s-Werten sinkt die Laufzeit und die Qualität nur in geringem Maße.

FRLayout++ erzielt bei den Kantenkreuzungen, der Standardabweichung der Kantenlänge und mutmaßlich auch in weiteren Kriterien die besten Werte. Für die drei getesteten Neukonstruktionsfunktionen habe ich exemplarisch je einen Algorithmus ausgewählt, der diese verwendet. Diese sind mit einigen Ergebnisse in Tabelle 4.1 aufgelistet. Die Werte sind gemittelt über alle erstellten Graphzeichnungen. Ausgewählt wurden die Varianten, die mit ihrer Neukonstruktionsfunktion den höchsten relativen Zeitgewinn pro Prozent mehr Kantenkreuzungen gegenüber FRLayout++ ausweisen konnten (letzte Spalte in der Tabelle). Dieses Auswahlkriterium habe ich ausgewählt, um drei Beispielvarianten vorzustellen. Die Wahl eines anderen Auswahlkriteriums wäre auch denkbar. Nur Algorithmen, die die logarithmische Neukonstruktionsfunktion verwendeten, waren in diesem Test im Mittel über alle Rome-Graphen schneller als der herkömmliche Algorithmus FRLayout++, der ohne WSPD arbeitet. Die vier aufgelisteten Beispielalgorithmen habe ich erneut die Rome-Graphen zeichnen lassen, wobei jeder Graph mit 10 verschiedenen, zufälligen initialen Knotenpositionen gezeichnet wurde. Die Ergebnisse sind in Abb. 4.7 zu sehen.

Der optimale Winkel an einem Knoten ist der graphentheoretische Grad des Knotens durch 360°. Als durchschnittliche Abweichung vom optimalen Winkel im Quadrat definiere ich hier den Mittelwert aller Quadrate aus Differenzen von optimalem Winkel und tatsächlichem Winkel durch den optimalen Winkel, wobei die tatsächlichen Winkel alle Winkel an den Knoten einer geradlinigen Zeichnung eines Graphen sind. Dabei wird der Sonderfall an Knoten mit Grad 0 und der Trivialfall an Knoten mit Grad 1 übergangen. Vorteil der Quadratur ist, dass größere Abweichungen stärker ins Gewicht fallen. Nachteil ist, dass die Interpretation nicht mehr so intuitiv einfach ist.

Die Grafik zur CPU-Zeit legt nahe, dass FRLayout++ mit dem stärkeren Wachstum der CPU-Zeit bei zunehmender Knotenzahl auch in der Praxis deutlich erkennbar in einer schlechteren Laufzeitklasse als die Varianten von FRLayout++WSPD liegt. Die Al-

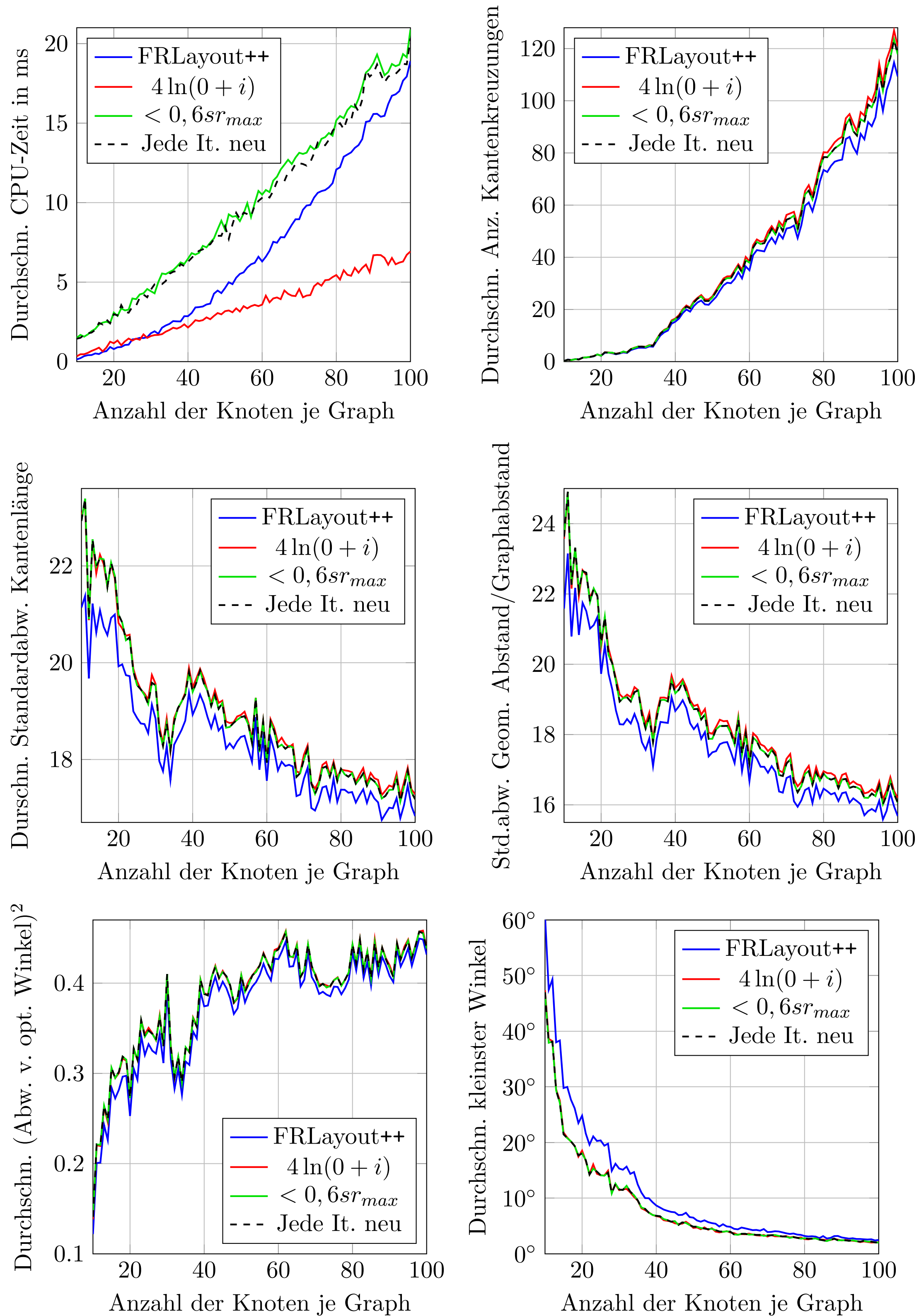


Abb. 4.7: Ergebnisse eines 10-fachen Durchlaufs der Rome-Graphen mit den Algorithmen aus Abb.4.1.

gorithmen mit $f^{\text{immer-neu}}_{\text{Neukonstruktion}}$ und $f^{\text{min-dist}}_{\text{Neukonstruktion}}$ scheinen hier etwa 3 mal mehr Zeit als der Algorithmus mit $f^{\text{Log}}_{\text{Neukonstruktion}}$ zu benötigen. Das macht sie in diesem Fall für die Graphen der getesteten Größenordnung als Alternative uninteressant, denn sie benötigen mehr Zeit und liefern schlechtere Ergebnisse als FRLayout++. Allerdings ist zu erwarten, dass sie für Graphen über 100 Knoten durch die bessere Laufzeitklasse zunehmend schneller sind als FRLayout++. Bei den Qualitätskriterien liefert FRLayout++ zwar jeweils bessere Ergebnisse als die exemplarischen Varianten von FRLayout++WSPD, doch liegen die Ergebnisse von FRLayout++WSPD relativ eng bei denen von FRLayout++ und auch mit größer werdenden Knotenzahlen ist ein „Abhängen" der Qualtität der Varianten mit WSPD nicht zu erkennen. Genauer aufgeschlüsselt sind zwei Qualitätsmerkmale für die Rome-Graphen mit 100 Knoten in Abb. 4.8. Dort können verschiedene Varianten von FRLayout++WSPD untereinander und mit FRLayout++, FRLayout++Quadtree und KKLayout (siehe zu diesen die beiden folgenden Abschnitte) anhand ihres Mittelwerts (über die 10 Durchläufe) der Standardabweichung der Kantenlänge und der Anzahl der Kantenkreuzungen bei jedem der Graphen verglichen werden.

Aufgrund des Umfangs des Testsatzes der Rome-Graphen, der zudem als Datensatz für umfangreiche Testzwecke ausgelegt ist, und dem klaren Verlauf der Testergebnisse kann es zumindest für Graphen dieser Größenordnung als empirisch nachgewiesen angesehen werden, dass ein auf diese Art beschleunigter Graphzeichenalgorithmus annährend so gute Ergebnisse liefert wie der zugrunde liegende Algorithmus. Die konkreten Implementierungen FRLayout++ und FRLayout++WSPD werden dabei als repräsentativ angesehen, da sie das allgemeine Konzept im Grundsätzlichen umsetzen und zudem gerade die Veränderung des ursprünglichen Algorithmus, auf die sich diese Aussage bezieht, genau der in Kap. 3 formulierten Veränderung folgt.

Die beste Verbesserung erreicht der Algorithmus mit $f^{\text{Log}}_{\text{Neukonstruktion}}$. Der Verlust an Qualität ist nur geringfügig größer als bei den anderen vorgeschlagenen Varianten von FRLayout++WSPD, während die Laufzeit am deutlichsten verbessert wird. Eine gute Wahl für einen a-Wert scheint hier in der Größenordnung 4-10 zu liegen, je nach Priorität von Geschwindigkeit (kleineres a und/oder größeres b) und Qualität(umgekehrt). Ein b-Wert der Größenordnung 5-10 in Kombination mit einem angemessenen a kann meiner Einschätzung nach auch Sinn machen. Problem von Algorithmus $f^{\text{min-dist}}_{\text{Neukonstruktion}}$ ist, dass die Überprüfung jede Iteration eine feste, zusätzliche Zeit benötigt und es zudem fraglich ist, ob das gewählt Kriterium zur Neukonstruktion (ein minimaler Abstand ist mindestens ein mal unterschritten) der noch vorhandenen Güte der WSPD gerecht wird, denn hier scheint fast immer auch bei meiner schwächsten getesteten Bedingung neu konstruiert zu werden. Hier könnte das Kriterium umformuliert werden und weitere Ansätze untersucht werden.

Insgesamt lässt sich sagen, dass man mit $f^{\text{Log}}_{\text{Neukonstruktion}}$ eine Neukonstruktionsfunktion für FRLayout++WSPD hat, mit der sehr schnell Ergebnisse geliefert werden können, die qualitativ einem hohen Mindeststandard genügen. Will man diesen qualitativen Mindeststandard mit den hier vorgestellten Methoden weiter verbessern, muss man dafür relativ teuer mit zusätzlicher Laufzeit bezahlen, sei es durch andere Neukonstruktionsfunktionen, andere Neukonstruktionsfunktionsparameter, größere s-Werte oder dem ver-

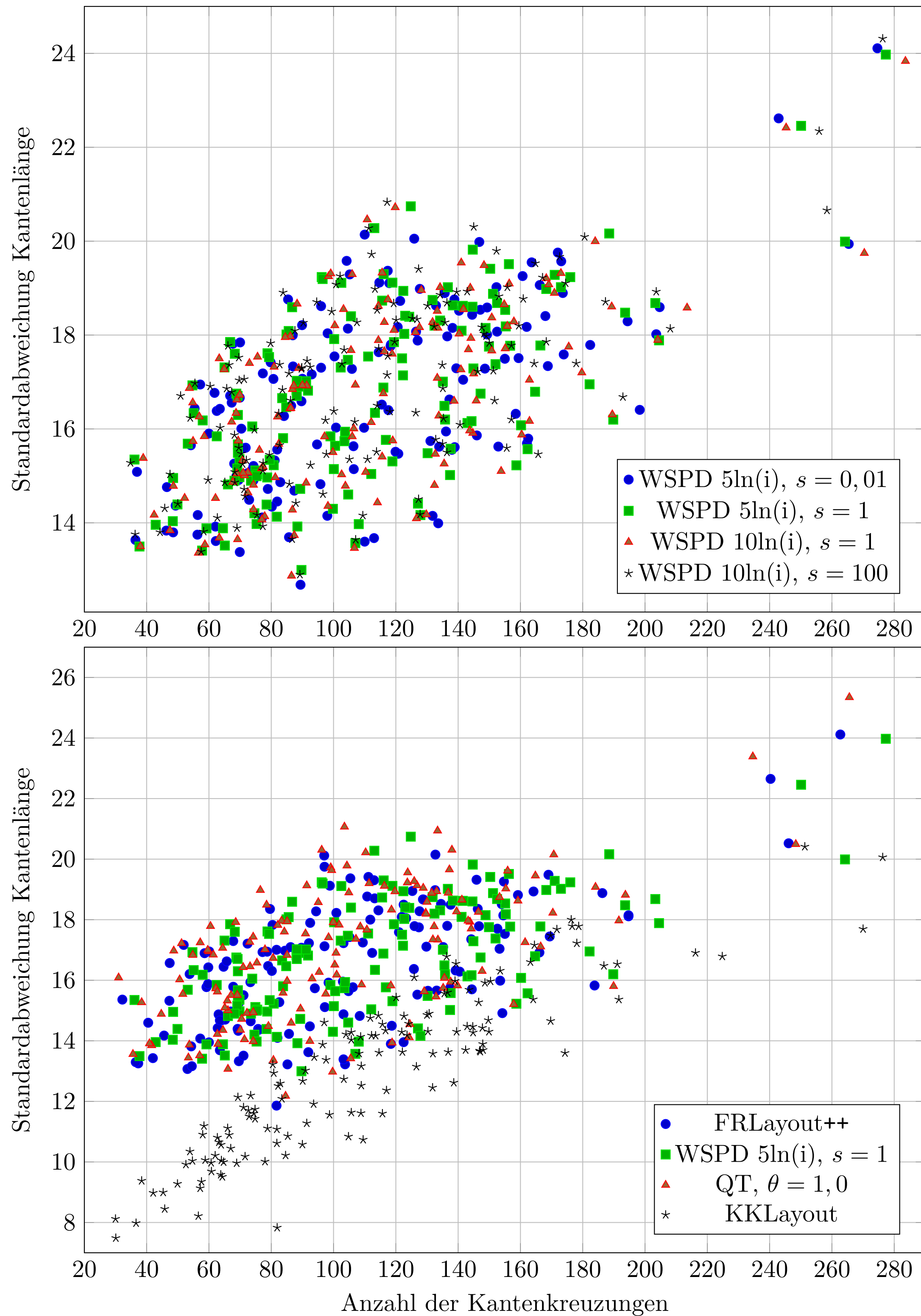


Abb. 4.8: Ergebnisse eines 10-fachen Durchlaufs der Rome-Graphen mit 100 Knoten. Für jeden Graphen ist zu den entsprechenden Algorithmen aus diesen Durchläufen der Mittelwert bzgl. Standardabweichung der Kantenlänge gegenüber der Anzahl der Kantenkreuzungen vermerkt.

wenden des herkömmlichen Algorithmus ohne WSPD. Die schlechtere Laufzeitklasse von FRLayout++ und der zeitliche Aufwand, der allein für das Konstruieren von Split-Tree und WSPD immer benötigt werden, haben zur Folge, dass sich FRLayout++WSPD gegenüber FRLayout++ abhängig von den Einstellungen im Mittel erst ab einer bestimmten Knotenzahl lohnt (zeitliche Einsparungen mit sich bringt). Daher sollte, wenn diese Knotenzahl bekannt ist oder abgeschätzt werden kann, im Graphzeichenalgorithmus eine Zusammenhangskomponente erst ab dieser Knotenzahl mit WSPD gezeichnet werden und kleinere Komponenten auf herkömmliche Art und Weise. FRLayout++WSPD mit $4\ln(i)$ lohnt sich zeitlich hier ab etwa 30 Knoten.

4.6 Vergleich von FRLayout++ und FRLayout++WSPD mit FRLayout++Quadtree

Als ein anderes Verfahren zur Beschleunigung, das herkömmliche Graphzeichenalgorithmen beschleunigen soll, habe ich den Algorithmus von Quigley und Eades, der eine Anwendung des Verfahrens von Barnes und Hut zum Graphzeichnen darstellt, implementiert. In diesem Verfahren wird als Datenstruktur zur Beschleunigen der Quadtree genutzt. Meine Implementierung basiert ebenfalls auf FRLayout++. Der Algorithmus wird im Folgenden *FRLayout++Quadtree* genannt. Die Ergebnisse eines fünffachen Durchlaufs der Rome-Graphen sind in Abb. 4.9 zu sehen. Dort und in folgenden Vergleichen wird für FRLayout++WSPD ein s-Wert von 1 gewählt, da dieser ähnliche Ergebnisse wie ein s-Wert kleiner 1 mit sich bringt.

Von der Laufzeit her kann dieser Algorithmus bei FRLayout++WSPD mithalten. Die Laufzeitklasse scheint in etwa dieselbe zu sein und je nach θ-Wert, s-Wert und Neukonstruktionsfunktion ist das eine oder das andere schneller. Hier ist der Repräsentant von FRLayout++WSPD zwar schneller, dieser ist mit der Neukonstruktionsfunktion $5\ln(i)$ aber im Vergleich zu anderen getesteten Varianten von FRLayout++WSPD wie der mit der Neukonstruktionsfunktion, die immer *wahr* zurückgibt, ohnehin schnell. Erstaunlicherweise können die Varianten von FRLayout++Quadtree anders als die von FRLayout++WSPD das Niveau der Kantenkreuzungen von FRLayout++ bei den getesteten Graphen in voller Höhe halten, ja sogar in der Größenordnung 0-5% verbessern.

Anders verhält es sich mit den weiteren getesteten Qualitätskriterien. Hier schneidet FRLayout++Quadtree besonders bei den kleinen Graphen bis etwa 30-60 Knoten deutlich schlechter ab als FRLayout++ und FRLayout++WSPD. So ist die Standardabweichung der Kantenlänge bis etwa 30 Knoten im zweistelligen Prozentbereich größer und Richtung 100 Knoten immer noch rund 2-3% höher. Besonders sticht die schlechtere Winkelauflösung hervor. So ist der durchschnittlich kleinste Winkel einer Zeichnung von FRLayout++Quadtree durchweg etwa 70-80% kleiner als der von FRLayout++. Auch die θ-Werte überraschen, denn der langsamste Algorithmus mit $\theta = 0,6$ erreicht nur bei der Zahl der Kantenkreuzungen eine etwas bessere oder gleiche Qualität als die Algorithmen mit den größeren θ-Werten.

Insgesamt lässt sich sagen, dass FRLayout++ sowohl durch FRLayout++WSPD als auch durch FRLayout++Quadtree sinnvoll beschleunigt werden kann. In der Qualität

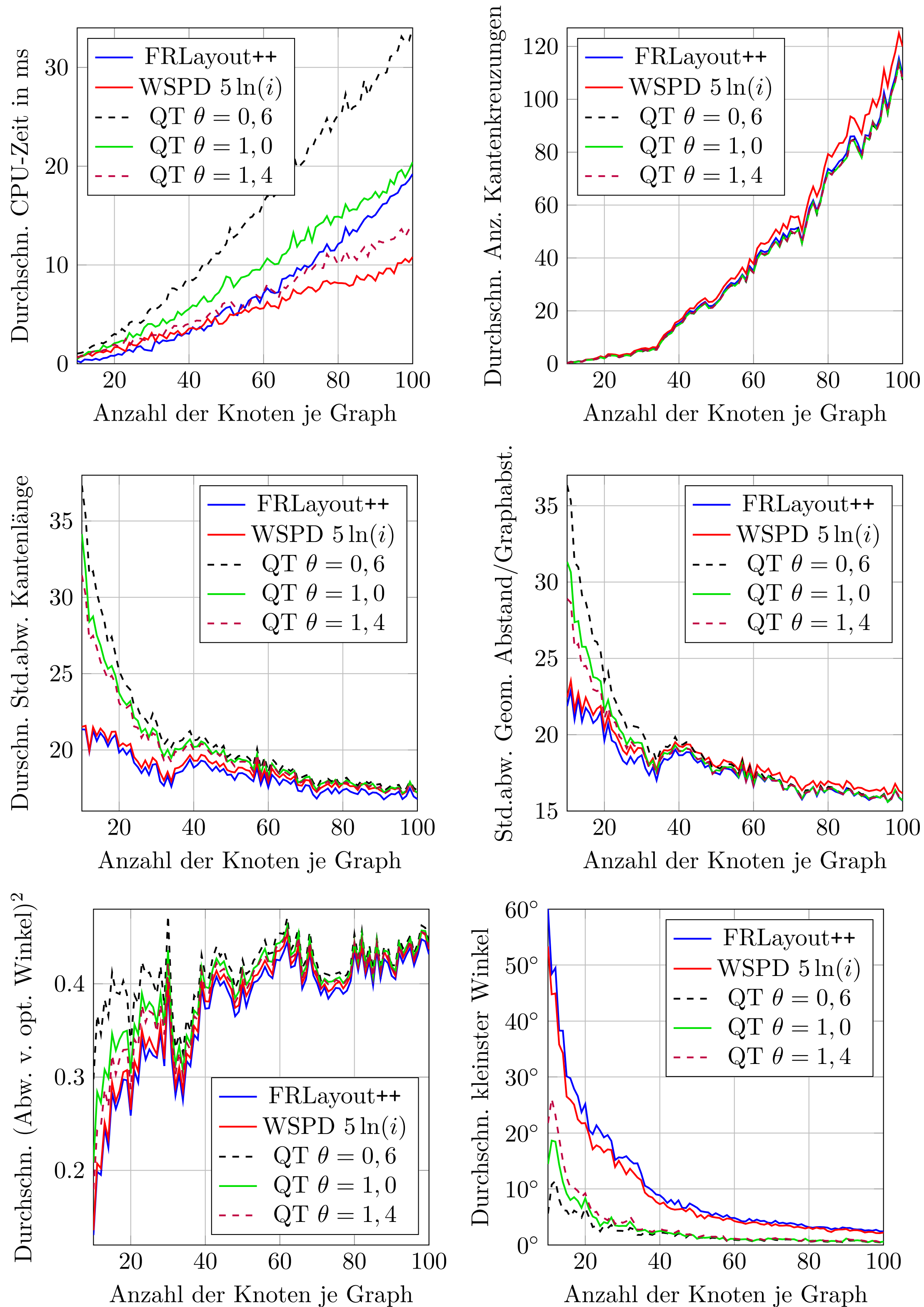


Abb. 4.9: 5-facher Durchlauf der Rome-Graphen mit FRLayout++, FRLayout++WSPD mit ln-Neukonstruktionsfunktion und $s = 1,0$ und FRLayout++Quadtree mit drei verschiedenen Werten für θ.

unterscheiden sie sich jedoch mit ihren Stärken und Schwächen. Während die Ergebnisse von FRLayout++WSPD etwas schlechter sind, jedoch insgesamt eng bei denen von FRLayout++ liegen, liefert FRLayout++Quadtree kriterienabhängig bessere oder schlechtere Ergebnisse. Wie sich das in den zurückgegebenen Zeichnungen niederschlägt kann exemplarisch im letzten Abschnitt dieses Kapitels betrachtet werden. Eine Implementierung des reduzierten Quadtrees ist nicht erfolgt, denn auch wenn dieselbe Laufzeitklasse wie bei FRLayout++WSPD nicht garantiert werden kann, so deuten die Ergebnisse doch darauf hin, dass in der praktischen Anwendung diese eingehalten wird. Auch ist die Varianz der Laufzeit nicht größer als bei FRLayout++WSPD. Neukonstruktionsfunktionen auch für den Quadtree einzuführen und deren Wirkung zu prüfen, könnte ebenfalls einen Test wert sein.

4.7 Vergleich von FRLayout++ und FRLayout++WSPD mit KKLayout

Zuletzt soll noch der Algorithmus von Kamada und Kawai mit dem neuen Algorithmus verglichen werden. Als Implementierung dieses Algorithmus verwende ich die Klasse *KKLayout* aus dem JUNG-Framework. Diese Klasse habe ich nicht verändert. Es ist daher möglich, dass sie auf ähnlich langsamen Strukturen wie FRLayout basiert oder qualitativ noch bessere Ergebnisse liefern könnte.

Die Statistiken zu einem dreifachen Durchlauf der Rome-Graphen sind in Abb. 4.10 zu sehen. Alle drei Algorithmen benötigen ähnlich viele Kantenkreuzungen. Die Kurve von KKLayout verläuft hier in etwa in der Mitte zwischen den beiden anderen. In allen anderen Qualitätsmerkmalen liefert KKLayout teils deutlich bessere Ergebnisse als FRLayout++ und FRLayout++WSPD. Das deckt sich auch mit dem Eindruck beim direkten Betrachten von Zeichnungen dieser Algorithmen. Die Zeichnungen sehen meist ansprechender aus, wobei besonders die gleichmäßige Kantenlänge eine Stärke dieses Algorithmus ist.

Diese vergleichsweise guten Ergebnisse werden jedoch auch mit einer hohen Laufzeit bezahlt. Verglichen mit einer Zeit im niedrigen zweistelligen Millisekundenbereich, die FRLayout++ und Derivate zum Zeichnen eines Rome-Graphen benötigen, erscheint eine Laufzeit im Bereich um eine Sekunde geradezu astronomisch hoch zu liegen. Die Kurven dieser sind in der Grafik der CPU-Zeiten so dicht an die x-Achse geschmiegt, dass sie kaum erkennbar sind. Ob sich der seltsame Verlauf der Kurve zur CPU-Zeit von KKLayout, welche für Graphen ab etwa 40 Knoten wieder sinkt, mit verschiedenen benötigten Iterationsanzahlen oder anders erklären lässt, kann ich nicht sagen. Es mag sein, dass sich durch eine bessere Implementierung die Laufzeit noch etwas senken lässt. Dass sie damit in den Bereich von FRLayout++ oder ähnlichem kommt, darf als unwahrscheinlich gelten. Zudem liegt die Laufzeitklasse von KKLayout höher als die von FRLayout++, sodass sich diese Methode schon aufgrund der Laufzeit kaum für große Graphen eignet.

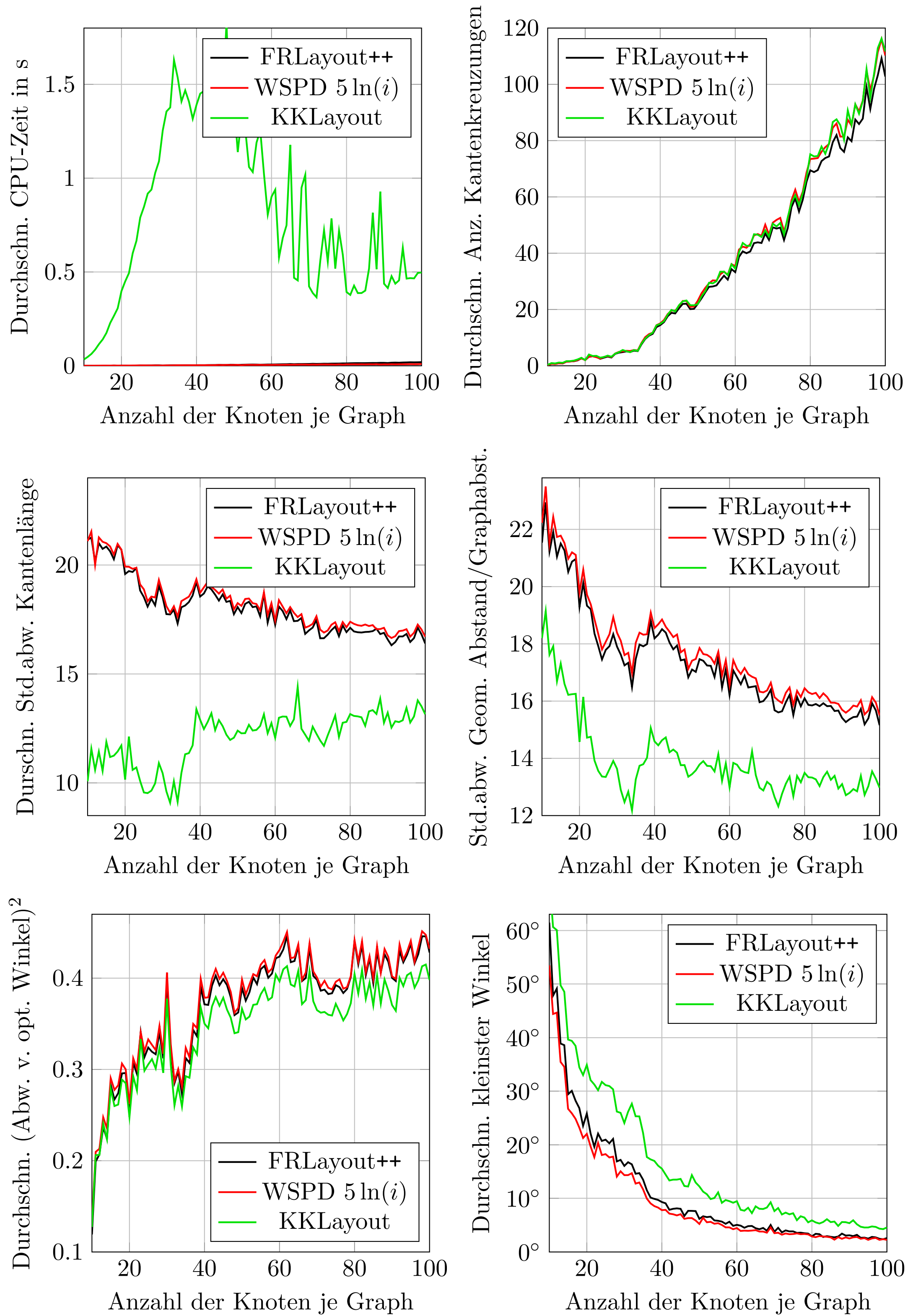


Abb. 4.10: 3-facher Durchlauf der Rome-Graphen mit FRLayout++, FRLayout++WSPD mit ln-Neukonstruktionsfunktion und $s = 1,0$ und KKLayout.

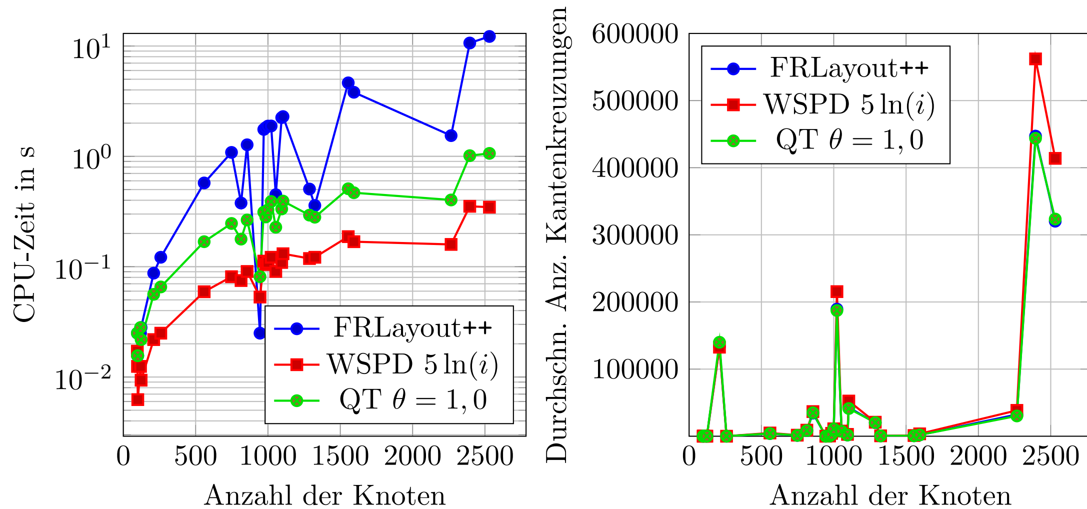


Abb. 4.11: 5-facher Durchlauf von 29 Hachul-Graphen mit max. 2531 Knoten mit FRLayout++, FRLayout++WSPD mit ln-Neukonstruktionsfkt. und $s = 1,0$ und FRLayout++Quadtree mit $\theta = 1,0$.

4.8 Vergleich bei größeren Graphen

Um insbesondere den erwarteten Laufzeitvorteil bei etwas größeren Graphen als den Rome-Graphen mit maximal 100 Knoten zu sehen, wurden die kleineren Graphen aus der Sammlung „Hachul-Graphen“ [Hac] gezeichnet. Einige Graphen aus dieser Sammlung erscheinen in der Diplomarbeit von Gronemann [Gro09].

Diese Sammlung enthält sehr große Graphen. Da die vorgestellten Algorithmen solch große Graphen nicht schön zeichnen können und auch die Laufzeit sehr hoch liegt, wurden nur 29 der kleinsten Graphen dieser Sammlung getestet. Der größte getestete Graph hat 2531 Knoten. Die Ergebnisse sind in Abb. 4.11 zu sehen.

Diese Ergebnisse sind jedoch sehr mit Vorsicht zu genießen. Die Kurven werden nur von sehr wenigen Datenpunkten aufgespannt, die eingezeichneten Kurven sind daher sehr schwankend und wenig aussagekräftig. Die Graphen sind, anders als die Rome-Graphen, keineswegs ein repräsentativer Testdatensatz, sondern eher Graphen mit speziellen Formen, wie Gitter oder eine Schneeflocke. Im folgenden Abschnitt sind Zeichnungen zu fünf dieser Graphen abgebildet. Die Skala zur CPU-Zeit ist logarithmisch. Viel mehr als, dass FRLayout++ deutlich mehr Zeit benötigt, kann man aus den Grafiken nicht ableiten. FRLayout++WSPD ist zudem noch etwas schneller als FRLayout++Quadtree, hat bei den zwei größten Graphen jedoch auch etwa 100.000 Kantenkreuzungen mehr produziert. Bei dieser Größenordnung an Kantenkreuzungen für einen Graphen mit 2000-3000 Knoten und 6000-8000 Kanten erkennt man, dass diese Graphzeichenalgorithmen in dieser unveränderten Form bereits für mittelgroße Graphen kaum noch geeignet sind. Als Teil eines Multilevelalgorithmus könnten sie auch für größere Graphen eingesetzt werden.

Eine CPU-Zeit von 187,2 Sekunden (gut drei Minuten), die FRLayout++ zum Zeichnen

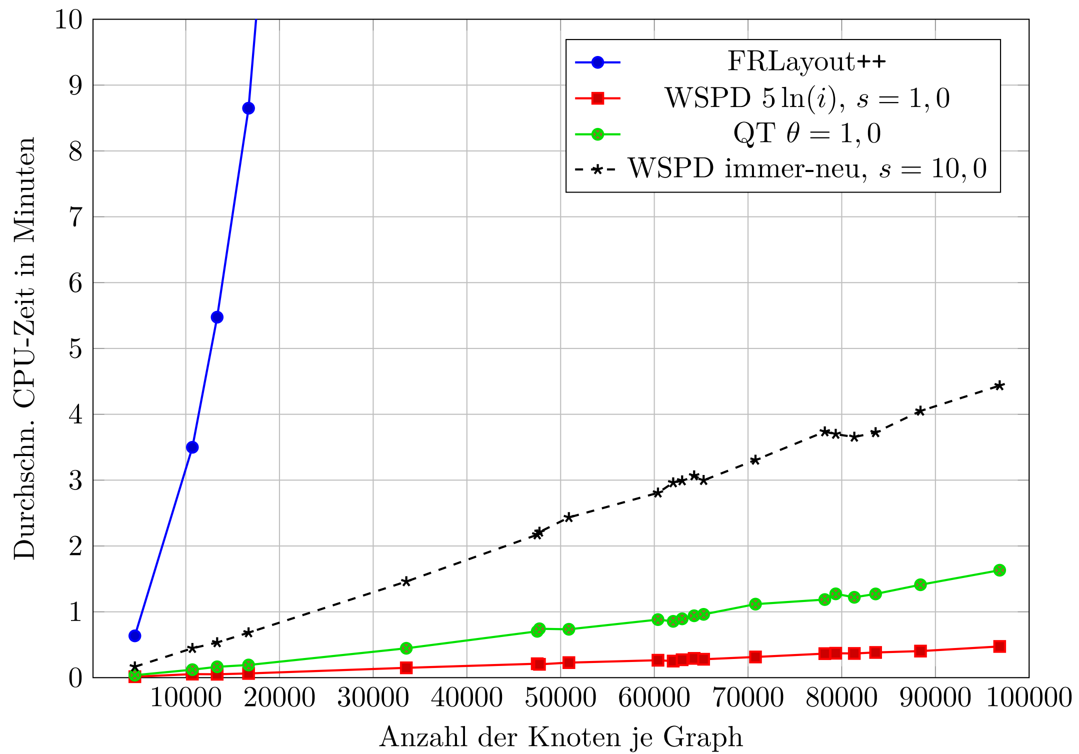


Abb. 4.12: Laufzeitvergleich für 20 zusammenhängende, zufällige Graphen und bis zu 96.856 Knoten.

eines zufälligen Graphen mit 10.000 Knoten (und 2,5 mal so vielen Kanten) benötigt hat, lässt zudem gegenüber 4,6 Sekunden (FRLayout++WSPD mit 5ln-Neukonstruktionsfunktion und $s = 1,0$) und 6,5 Sekunden (FRLayout++Quadtree mit $\theta = 1,0$) erahnen, was für ein zeitliches Einsparungspotenzial in den Algorithmen dieser Art vorliegt. In einem weiteren Testlauf wurden 20 zufällige Graphen mit bis zu 100.000 Knoten gezeichnet und die CPU-Zeit gemessen. Diese zufälligen Graphen wurden mit dem EppsteinPowerLawGenerator [EW02] aus JUNG erzeugt, wobei nur die größte Zusammenhangskomponente betrachtet wurde. Jeder dieser Graphen hat etwa 2,5 mal mehr Kanten als Knoten. Das relativ klare Ergebnis ist in Abb. 4.12 zu sehen. Bei Graphen dieser Größenordnung sind auch die langsameren Varianten von FRLayout++WSPD wie die mit $f_{\text{Neukonstruktion}}^{\text{immer-neu}}$ und $s = 10,0$ deutlich schneller.

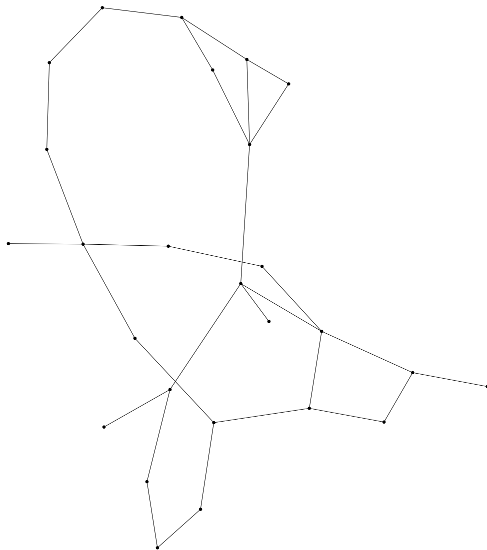
4.9 Abbildungen verschiedener Zeichnungen

Nach den bloßen statistischen Werten wird zur Bewertung der Qualität der gelieferten Zeichnungen in diesem Abschnitt noch ein Überblick über Zeichnungen einiger Beispielgraphen gegeben.

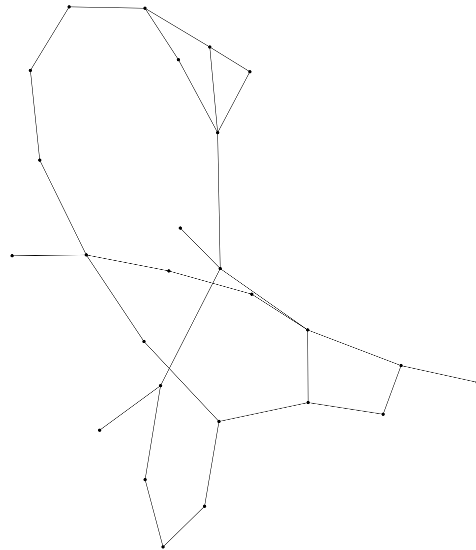
In Abb. 4.13, Abb. 4.14 und Abb. 4.15 sind Zeichnungen von Rome-Graphen unterschiedlicher Größe zu sehen. In Abb. 4.14 sind an den Zeichenalgorithmen nur die s-Werte variiert.

Die darauffolgenden fünf Abbildungen sind aus den Hachul-Graphen. Nur KKLayout kann in Abb. 4.16 jeden „Arm“ der Schneeflocke separat zeichnen. Ähnlich in Abb. 4.17 und Abb. 4.18. Bei den größten abgebildeten Zeichnungen Abb. 4.19 und Abb. 4.20, die verglichen mit dem, was heute möglich ist, noch nicht sehr groß sind, schafft es KKLayout dagegen nicht eine ansprechende Zeichnung zurückzugeben. Das könnte daran liegen, dass die Iterationen begrenzt sind und viel mehr Iterationen für eine gute Zeichnung nötig wären. Die auf FRLayout++ basierenden Zeichnungen lassen die Struktur bei diesen speziellen Beispielgraphen dagegen noch einigermaßen erkennen. Die Beobachtung der Winkel an den Blättern des Baumes in der Zeichnung von FRLayout++Quadtree in Abb. 4.19 deckt sich mit der Erwartung einer schlechteren Winkelauflösungen dieses Algorithmus.

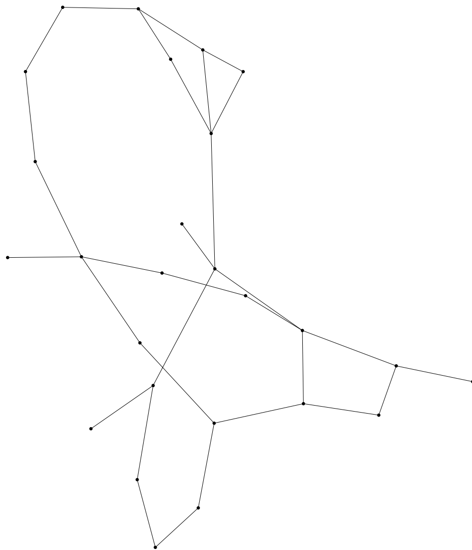
In allen Abbildungen bekamen alle vier Algorithmen dieselben zufälligen Knotenstartpunkte zugewiesen.



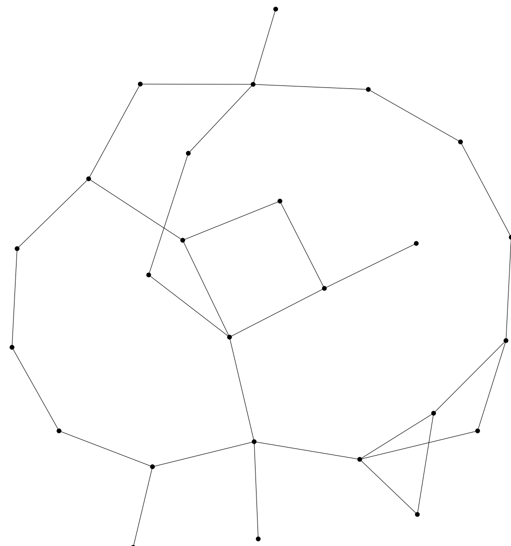
(a) FRLayout++



(b) FRLayout++WSPD mit Neukonstruktion in jeder Iteration und $s = 1,0$

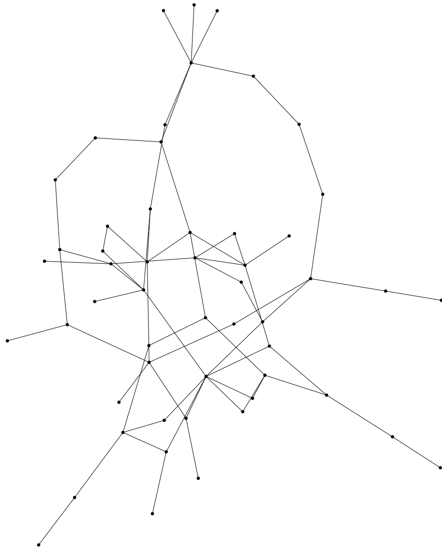


(c) FRLayout++WSPD mit Neukonstruktionsfunktion $5 \ln(i)$ und $s = 1,0$

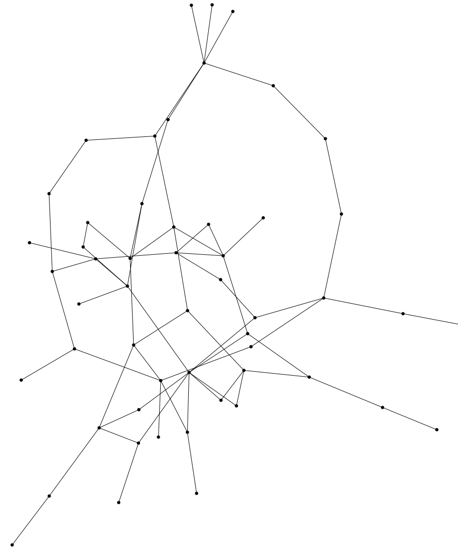


(d) KKLayout

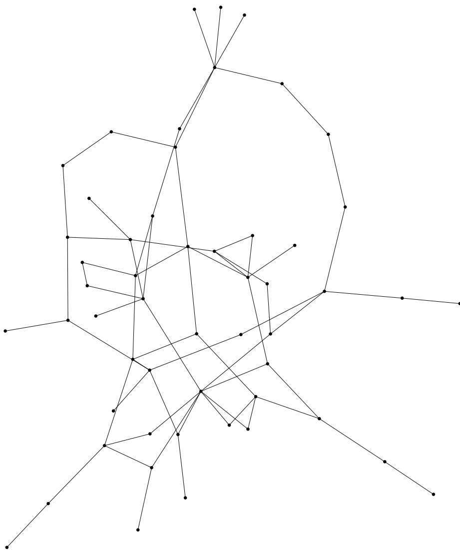
Abb. 4.13: Zeichnungen des Rome-Graphen graficon26nodi/grafo173.26 (26 Knoten) durch verschiedene Algorithmen.



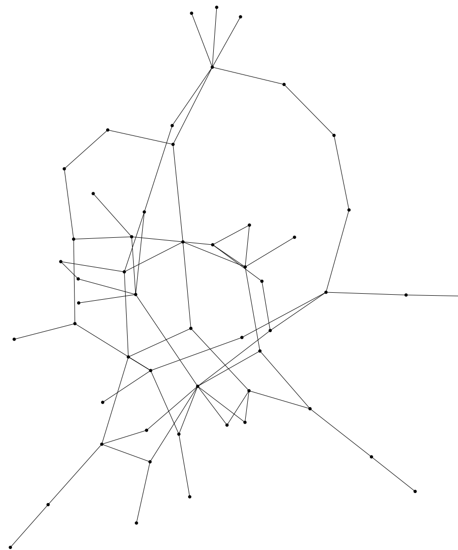
(a) FRLayout++WSPD mit Neukonstruktionsfunktion $5\ln(i)$ und $s = 0,001$



(b) FRLayout++WSPD mit Neukonstruktionsfunktion $5\ln(i)$ und $s = 0,01$

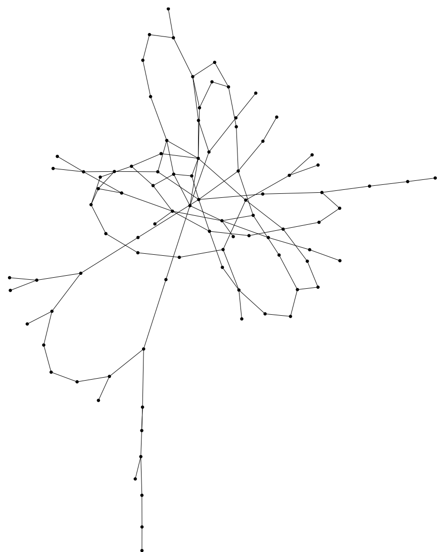


(c) FRLayout++WSPD mit Neukonstruktionsfunktion $5\ln(i)$ und $s = 1,0$

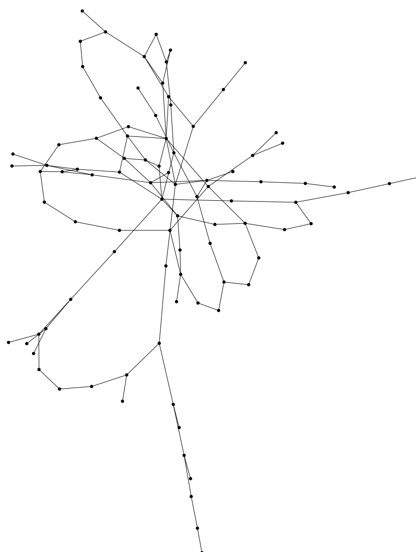


(d) FRLayout++WSPD mit Neukonstruktionsfunktion $5\ln(i)$ und $s = 100,0$

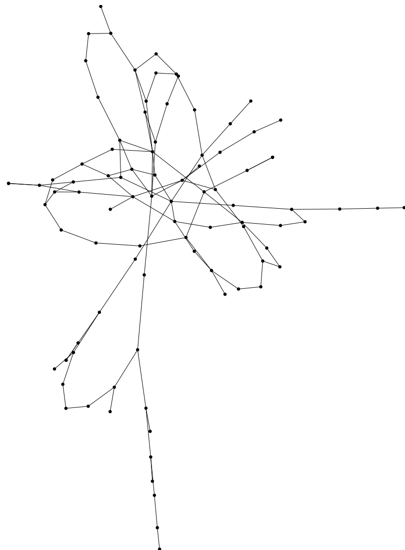
Abb. 4.14: Zeichnung des Rome-Graphen grafo2097.53 (53 Knoten) durch FRLayout++WSPD mit derselben Neukonstruktionsfunktion und verschiedenen s-Werten.



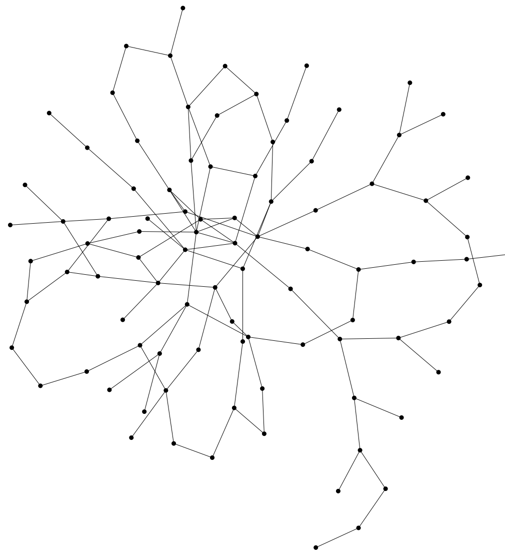
(a) FRLayout++



(b) FRLayout++WSPD mit Neukonstruktion in jeder Iteration und $s = 1,0$

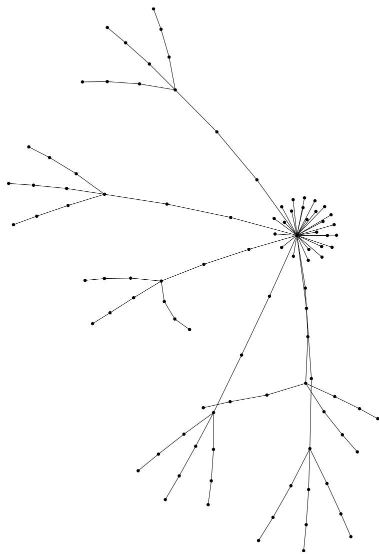


(c) FRLayout++Quadtree mit $\theta = 1,0$

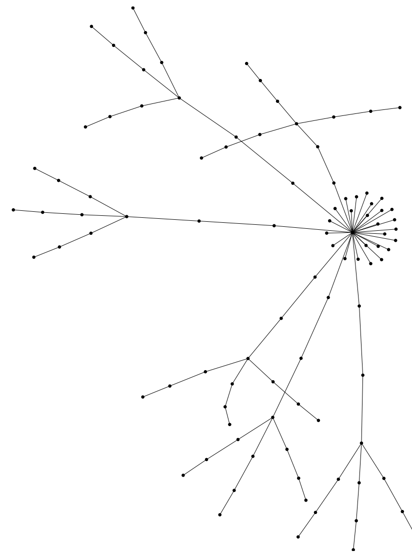


(d) KKLayout

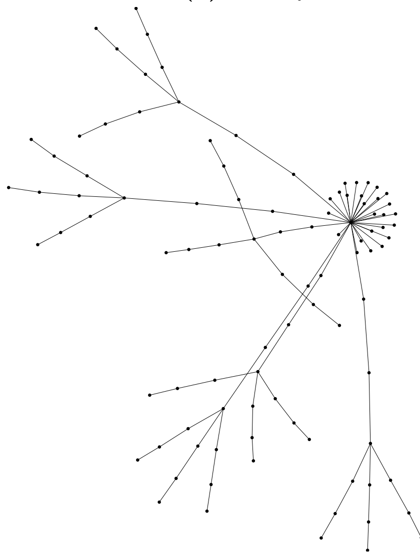
Abb. 4.15: Zeichnungen des Rome-Graphen grafo8573.92 (92 Knoten) durch verschiedene Algorithmen.



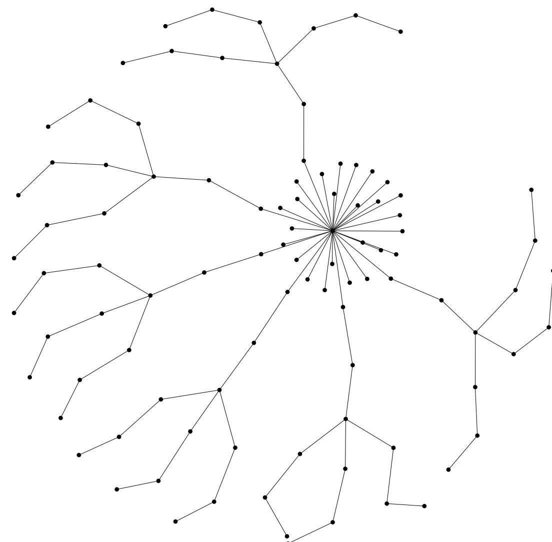
(a) FRLayout++



(b) FRLayout++WSPD mit Neukonstruktion in jeder Iteration und $s = 1,0$

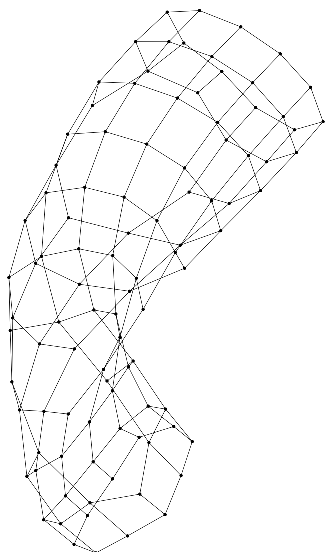


(c) FRLayout++WSPD mit Neukonstruktionsfunktion $5\ln(i)$ und $s = 1,0$

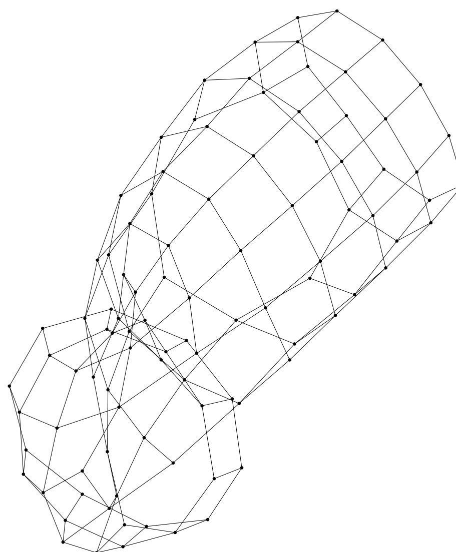


(d) KKLayout

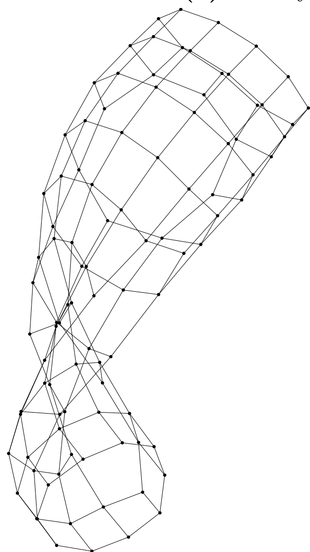
Abb. 4.16: Zeichnung des Hachul-Graphen snowflake_A (98 Knoten) durch verschiedene Algorithmen.



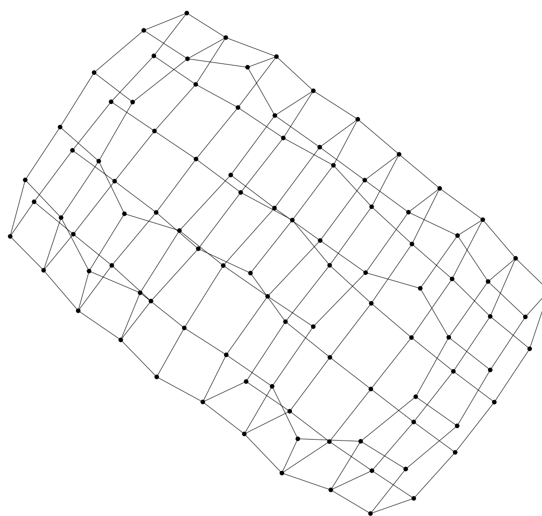
(a) FRLayout++



(b) FRLayout++WSPD mit Neukonstruktionsfunktion $5\ln(i)$ und $s = 1,0$

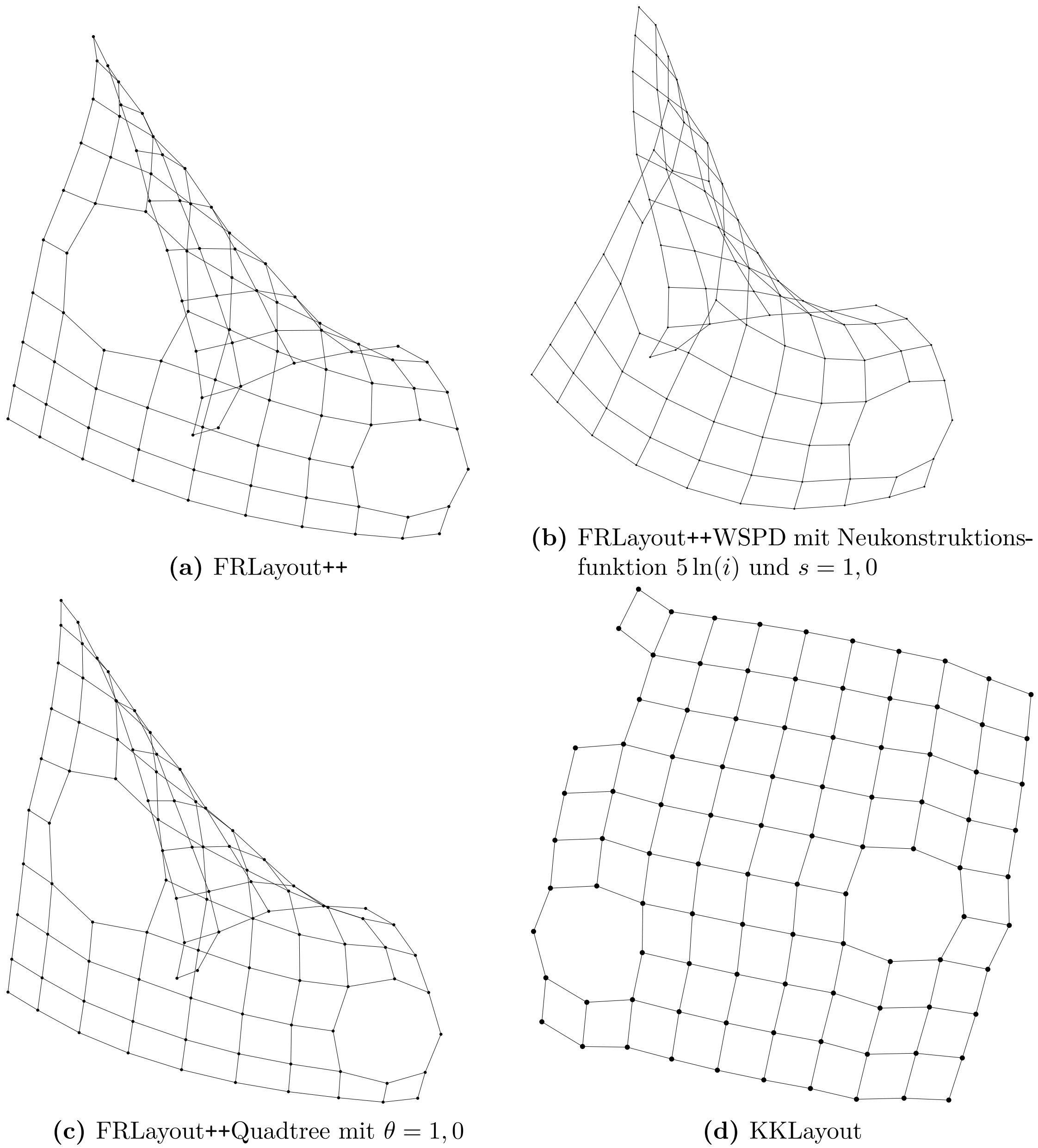


(c) FRLayout++Quadtree mit $\theta = 1,0$



(d) KKLayout

Abb. 4.17: Zeichnung des Hachul-Graphen cylinder_rnd_010_010 (97 Knoten) durch verschiedene Algorithmen.



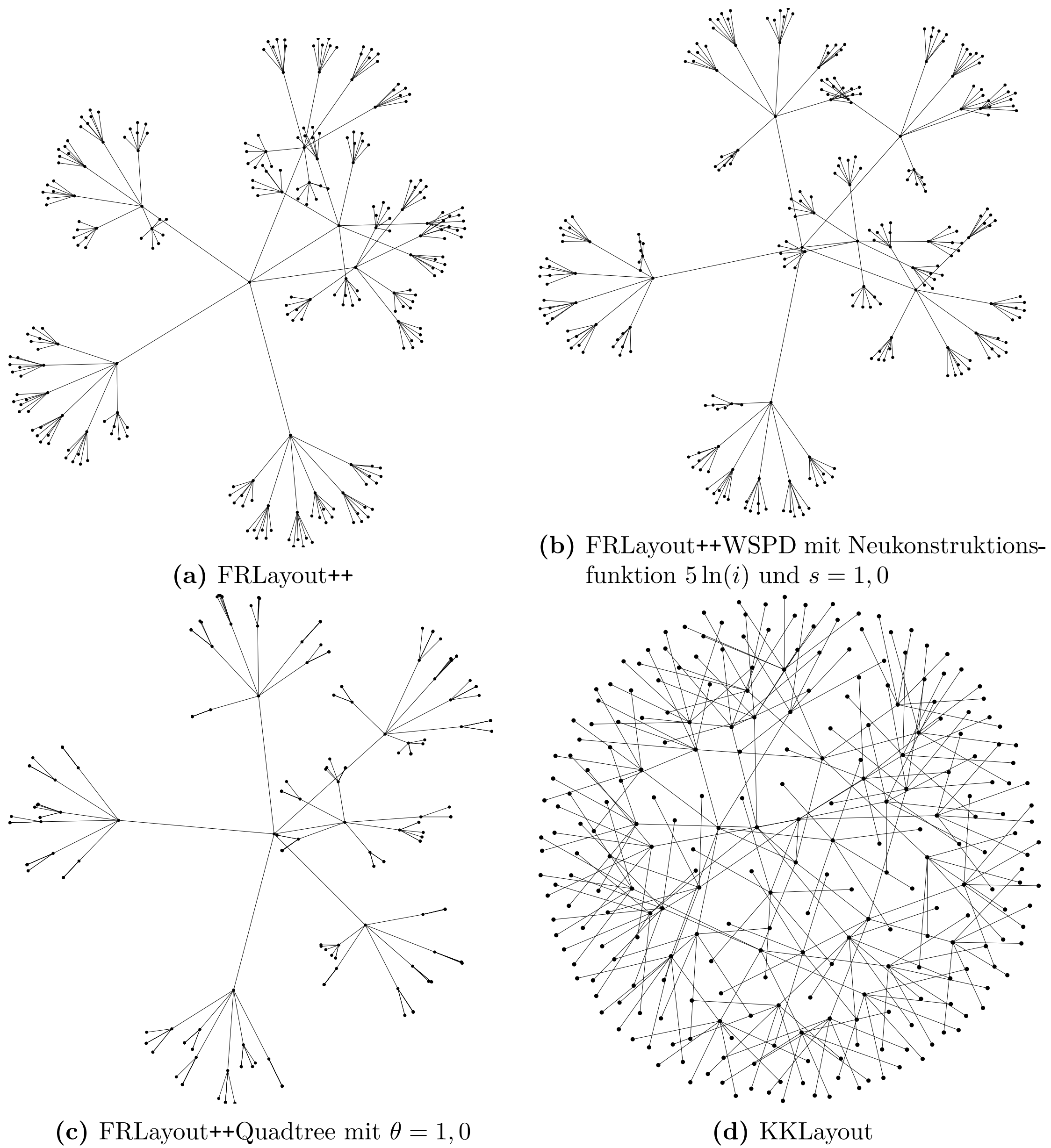
(a) FRLayout++

(b) FRLayout++WSPD mit Neukonstruktionsfunktion $5\ln(i)$ und $s = 1,0$

(c) FRLayout++Quadtree mit $\theta = 1,0$

(d) KKLayout

Abb. 4.18: Zeichnung des Hachul-Graphen grid_rnd_010 (97 Knoten) durch verschiedene Algorithmen.



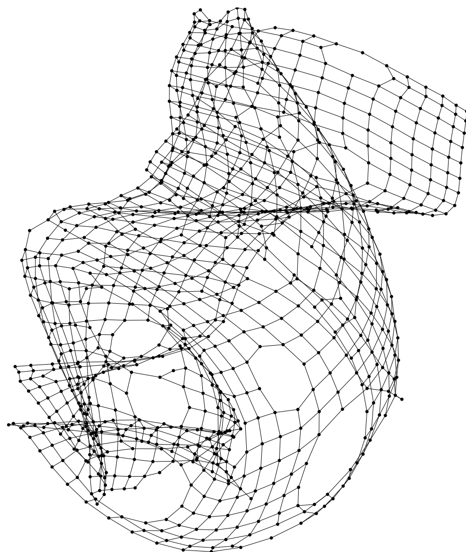
(a) FRLayout++

(b) FRLayout++WSPD mit Neukonstruktionsfunktion $5 \ln(i)$ und $s = 1,0$

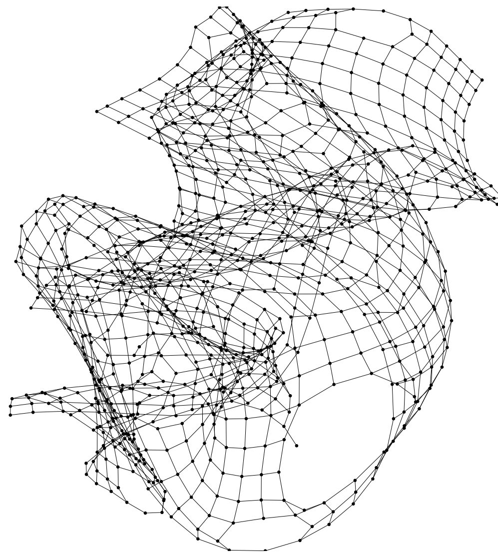
(c) FRLayout++Quadtree mit $\theta = 1,0$

(d) KKLayout

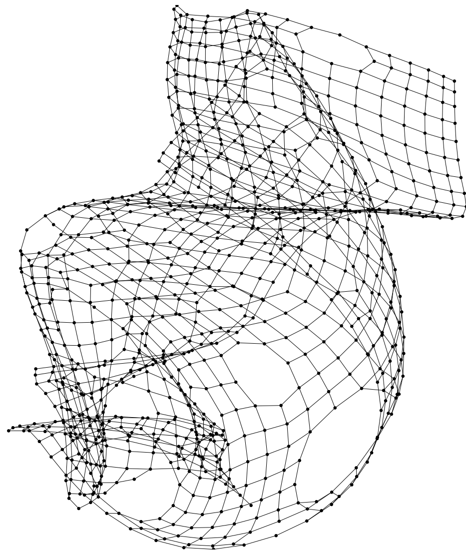
Abb. 4.19: Zeichnung des Hachul-Graphen tree_06_03 (259 Knoten) durch verschiedene Algorithmen. KKLayout benötigt 796 Kreuzung, während die anderen in dieser Reihenfolge nur auf 46, 64 und 11 Kreuzungen kommen.



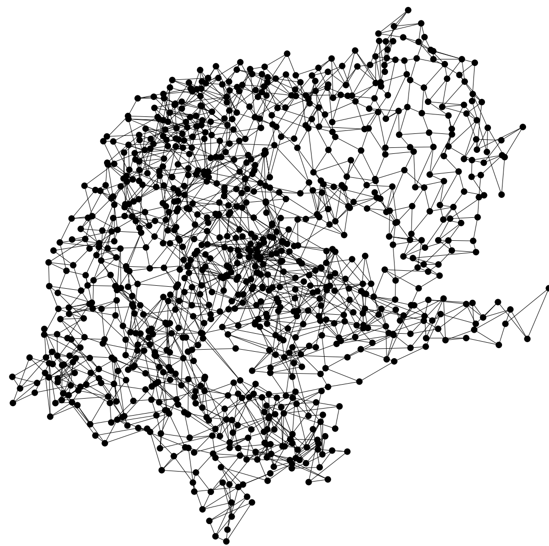
(a) FRLayout++



(b) FRLayout++WSPD mit Neukonstruktionsfunktion $5\ln(i)$ und $s = 1,0$



(c) FRLayout++Quadtree mit $\theta = 1,0$



(d) KKLayout

Abb. 4.20: Zeichnung des Hachul-Graphen grid_rnd_032 (985 Knoten) durch verschiedene Algorithmen. KKLayout benötigt 6269 Kreuzung, während die anderen in dieser Reihenfolge nur auf 1582, 2611 und 1674 Kreuzungen kommen.

5 Fazit und Ausblick

In dieser Bachelorarbeit wurde eine neue Möglichkeit vorgestellt, einen klassischen kräftebasierten Graphzeichenalgorithmus zu beschleunigen. Die angepassten Algorithmen zeichnet aus, dass sie

- die Laufzeitklasse des Algorithmus für einen Graphen $G = (V, E)$ von $O(|V|^2)$ auf $O(|V| \log |V| + |E|)$ verbessern. Dies wurde theoretisch gezeigt. Die statistischen Ergebnisse entsprechen den Erwartungen einer besseren Laufzeitklasse.

- qualitativ annähernd so gut Ergebnisse wie der zugrunde liegende Graphzeichenalgorithmus liefern. Dies wurde empirisch mit einer statistischen Auswertung der relativ umfangreichen Rome-Graphensammlung gezeigt.

Es bleibt jedoch festzuhalten, dass sich der Einsatz für Graphen mit zweistelliger Knotenzahl kaum lohnt. Die Zeitersparnis hält sich hier stark in Grenzen und das bessere Ergebnis liefert in den meisten Fällen immer noch der herkömmliche Algorithmus. Selbst wenn der qualitative Unterschied nur gering ist, so würde man an heutigen Rechnern wohl in den meisten Anwendungsfällen einige Bruchteilsekunden mehr in Kauf nehmen, wenn die zurückgegebene Zeichnung dafür im Mittel 1 oder 2 Kantenkreuzungen weniger enthält. Interessant wird dieser Ansatz besonders für größere Graphen, bei denen die Laufzeitklasse von $O(|V|^2)$ zum Problem wird.

Da das herkömmliche Zeichnen des kompletten Graphen, wie es hier geschehen ist, für solch große Graphen aufgrund der vielen lokalen Minima schlechte Ergebnisse mit vielen Kreuzungen liefert, könnte dieses Verfahren mit WSPD in einem Multilevel Graphzeichenalgorithmus zur Anwendung kommen. Als Teil eines solchen muss es sich mit vorhandenen Verfahren wie dem von Walshaw, der Fast Multipole Method von Hachul und Jünger oder MDS-Verfahren in Qualität und Laufzeit messen lassen.

Zu untersuchen wäre auch, ob sich die Verwendung von größeren s-Werten für die WSPD dann lohnt. Der vermeintliche Vorteil der WSPD, dass ein s-facher Mindestabstand garantiert wird, wird für gegen 0 gehende s-Werte, die beim Test der Rome-Graphen in der Nutzen-Kosten-Abwägung besonders gut abgeschnitten haben, kaum ausgenutzt. Außerdem könnten andere Ansätze für Neukonstruktionsfunktionen ausprobiert werden oder Möglichkeiten des Updates der Datenstrukturen untersucht werden.

Literaturverzeichnis

[APG94] Srinivas Aluru, Gurpur M. Prabhu und John Gustafson: Truly distribution-independent algorithms for the N-body problem. In: *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Seiten 420–428. IEEE Computer Society Press, 1994.

[BGKM11] Gereon Bartel, Carsten Gutwenger, Karsten Klein und Petra Mutzel: An experimental evaluation of multilevel layout methods. In: *Proc. Int. Symp. Graph Drawing (GD'10)*, Band 6502 der Reihe *LNCS*, Seiten 80–91. Springer, 2011.

[BH86] Josh Barnes und Piet Hut: A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[BHR96] Franz J. Brandenburg, Michael Himsolt und Christoph Rohrer: An experimental comparison of force-directed and randomized graph drawing algorithms. In: Franz J. Brandenburg (Herausgeber): *GD 1995*, Band 1027, Seiten 76–87. 1996. `http://dx.doi.org/10.1007/BFb0021792`.

[BP07] Ulrik Brandes und Christian Pich: Eigensolver methods for progressive multidimensional scaling of large data. In: *Proc. Int. Symp. Graph Drawing (GD'06)*, Band 4372 der Reihe *LNCS*, Seiten 42–53. Springer, 2007.

[Cal95] Paul B. Callahan: *Dealing with higher dimensions: the well-separated pair decomposition and its applications.* Dissertation, Johns Hopkins University Baltimore, Maryland, 1995.

[CK95] Paul B. Callahan und S. Rao Kosaraju: A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)*, 42(1):67–90, 1995.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to algorithms.* The MIT Press, 3. Auflage, 2009.

[dCvO08] Mark de Berg, Otfried Cheong, Marc van Kreveld und Mark Overmars: *Computational Geometry: Algorithms and Applications.* Springer, 3. Auflage, 2008.

[DETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia und Ioannis G. Tollis: *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.

[DH96] Ron Davidson und David Harel: Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG)*, 15(4):301–331, 1996.

[Ead84] Peter Eades: A heuristics for graph drawing. *Congressus numerantium*, 42:146–160, 1984.

[EL73] Kurt Endl und Wolfgang Luh: *Analysis II: Eine integrierte Darstellung.* Akademische Verlagsgesellschaft Frankfurt am Main, 1973.

[EW02] David Eppstein und Joseph Yannkae Wang: A steady state model for graph power laws. In: *2nd Int. Workshop Web Dynamics*, 2002.

[FB74] Raphael A. Finkel und Jon Louis Bentley: Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[FLM95] Arne Frick, Andreas Ludwig und Heiko Mehldau: A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In: *Proc. Int. Symp. Graph Drawing (GD'94)*, Band 894 der Reihe *LNCS*, Seiten 388–403. Springer, 1995.

[FR91] Thomas M.J. Fruchterman und Edward M. Reingold: Graph drawing by force-directed placement. *Softw., Pract. Exper.*, 21(11):1129–1164, 1991.

[GGK04] Pawel Gajer, Michael T. Goodrich und Stephen G. Kobourov: A multidimensional approach to force-directed layouts of large graphs. *Computational Geometry: Theory and Applications*, 29(1):3–18, 2004.

[GHK13] Emden R Gansner, Yifan Hu und Shankar Krishnan: Coast: A convex optimization approach to stress-based embedding. In: *Proc. Int. Symp. Graph Drawing (GD'13)*, Band 8242 der Reihe *LNCS*, Seiten 268–279. Springer, 2013.

[GKN04] Emden R Gansner, Yehuda Koren und Stephen North: Graph drawing by stress majorization. In: *Proc. Int. Symp. Graph Drawing (GD'04)*, Band 3383 der Reihe *LNCS*, Seiten 239–250. Springer, 2004.

[GR87] Leslie Greengard und Vladimir Rokhlin: A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.

[Gro09] Martin Gronemann: Engineering the fast-multipole-multilevel method for multicore and SIMD architectures. *Diplomarbeit, Technische Universität Dortmund*, 2009.

[Hac] Hachul-Graphen. `http://www.informatik.uni-koeln.de/public/gronemann/hachul_graphs.zip`.

[Har11] Sariel Har-Peled: *Geometric approximation algorithms.* American Mathematical Society Providence, 2011.

[HJ04] Stefan Hachul und Michael Jünger: Drawing large graphs with a potential-field-based multilevel algorithm. In: *Proc. Int. Symp. Graph Drawing (GD'04)*, Band 3383 der Reihe *LNCS*, Seiten 285–295. Springer, 2004.

[HK02] Yehuda Harel und David Koren: A fast multi-scale method for drawing large graphs. *Journal of Graph Algorithms and Applications*, 6(3):179–202, 2002.

[Hu05] Yifan Hu: Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[JUN] Java Universal Network/Graph Framework (JUNG). `http://jung.sourceforge.net/`.

[KK89] Tomihisa Kamada und Satoru Kawai: An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[KN12] Sven Oliver Krumke und Hartmut Noltemeier: *Graphentheoretische Konzepte und Algorithmen.* Springer, 2012.

[Kob12] Stephen G Kobourov: Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*, 2012.

[Mey99] Kenneth R Meyer: *Periodic solutions of the N-body problem.* Springer, 1999.

[NR04] Takao Nishizeki und Md Saidur Rahman: *Planar graph drawing.* World Scientific, 2004.

[NS07] Giri Narasimhan und Michiel Smid: *Geometric spanner networks.* Cambridge University Press, 2007.

[QE01] Aaron Quigley und Peter Eades: FADE: Graph Drawing, Clustering, and Visual Abstraction. In: *Proc. Int. Symp. Graph Drawing (GD'00)*, Band 1984 der Reihe *LNCS*, Seiten 197–210. Springer-Verlag, 2001.

[Rom] Rome-Graphen. `www.graphdrawing.org/download/rome-graphml.tgz`.

[Sal91] John K. Salmon: *Parallel hierarchical N-body methods.* Dissertation, California Institute of Technology, 1991.

[Sam90] Hanan Samet: *The design and analysis of spatial data structures.* Addison-Wesley Reading, MA, 1990.

[ST02] Vin D. Silva und Joshua B. Tenenbaum: Global versus local methods in nonlinear dimensionality reduction. In: *Advances in neural information processing systems*, Seiten 705–712, 2002.

[STT81] Kozo Sugiyama, Shojiro Tagawa und Mitsuhiko Toda: Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[Tun98] Daniel Tunkelang: Jiggle: Java interactive graph layout environment. In: *Proc. Int. Symp. Graph Drawing (GD'98)*, Band 1547 der Reihe *LNCS*, Seiten 412–422. Springer, 1998.

[Tut63] William T Tutte: How to draw a graph. *Proc. London Math. Soc*, 13(3):743–768, 1963.

[Wal03] Chris Walshaw: A multilevel algorithm for force-directed graph-drawing. *Journal of Graph Algorithms and Applications*, 7(3):253–285, 2003.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 9. November 2015

. . . .
Johannes Zink