
Universität Leipzig

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

# Verteiltes Graph-Layouting mit Gradoop

Masterarbeit

vorgelegt von:
Daniel Baumgarten

Matrikelnummer:
3710159

Betreuer:
M. Sc. Kevin Gomez
Prüfer:
Prof. Dr. Erhard Rahm
Dr. Johannes Zschache

# Inhaltsverzeichnis

# 1 Einführung

Viele Datensätze, die für Forschung und Wirtschaft relevant sind, liegen in der Form von Graphen vor. Einige Beispiele, für Daten die typischerweise als Graph vorliegen, sind beispielsweise Straßenkarten, soziale Netzwerke, Kaufempfehlungsdatenbanken, Wissensdatenbanken aber auch weniger alltägliche Dinge wie etwa der logische Aufbau elektrischer Schaltungen.

Der erste Schritt bei der Analyse eines Datensatzes ist es in der Regel, sich einen groben Überblick über die Daten zu verschaffen. Dazu kommen, neben einigen statistischen Kennzahlen, vor allem Visualisierungen zum Einsatz. Das menschliche Gehirn ist sehr gut darin, Muster in Bildern zu identifizieren, weshalb sich mit einer guten Visualisierung und der Betrachtung durch einen Menschen oft wertvolle Informationen über den Datensatz in Erfahrung bringen lassen.

Allerdings gestaltet sich das Visualisieren von Graphen stellenweise schwierig. Nur die wenigsten Graph-Datensätze (wie etwa Straßenkarten) enthalten detaillierte Informationen dazu, wie die einzelnen Elemente des Graphen für eine Visualisierung räumlich anzuordnen sind. Das Generieren einer solchen räumlichen Anordnung für die Elemente eines Graphen wird als Layouting bezeichnet. Dabei werden jedem Knoten des Graphen Koordinaten im Layoutbereich zugewiesen. Bei der Zuweisung der Koordinaten muss darauf geachtet werden, dass die entstehende Anordnung die dem Graphen zugrundeliegende Struktur möglichst akkurat wiedergibt, so dass eine Zeichnung des erstellten Layouts einem Betrachter ermöglicht, einfach relevante Informationen über den Graphen zu gewinnen. Beispielsweise sollten bei der Visualisierung des Freundes-Graphen eines sozialen Netzwerkes die Knoten von Freunden möglichst dicht beieinander gezeichnet werden, so dass Freundesgruppen direkt als Clique erkennbar sind. Allerdings muss dabei auch darauf geachtet werden, jegliche Überlappungen, sofern möglich, zu vermeiden. Überlappt ein Knoten einen Anderen geht der Untere visuell praktisch verloren. Kreuzen sich Kanten zu häufig, fällt es sehr schwer deren Verlauf zu folgen.

Force-directed Layouting Algorithmen sind die bekanntesten Techniken zum Layouten von Graphen, da sie leicht verständlich und meist simpel zu implementieren sind. Dabei werden Graphen als physikalisches System betrachtet, bei dem Knoten aufeinander Anziehungs- und Abstoßungskräfte ausüben. Durch iteratives Anwenden der ausgeübten Kräfte auf die Positionen der Knoten lassen sich Anordnungen für Graphen finden, die deren interne Struktur widerspiegeln.

Leider sind diese Verfahren prinzipbedingt sehr rechenintensiv, speziell dann, wenn sie für Graphen mit vielen Knoten durchgeführt werden sollen. In anderen Bereichen der Datenanalyse begegnete man den in letzter Zeit stark gestiegenen Datenmengen und dem dadurch stark steigenden Rechenaufwand mit Techniken, um die nötigen Berechnungen auf einer Vielzahl von Rechner parallel durchzuführen. Beispielsweise ermöglicht die Software Gradoop es, Analysen von

Graphen mit Hilfe von Apache Flink über eine große Anzahl von Rechnern verteilt durchzuführen. Allerdings unterstützt Gradoop derzeit nicht das Layouten und Zeichnen der analysierten Graphen.

Ziel dieser Arbeit ist es, zu Gradoop die Funktion hinzuzufügen, die zu analysierenden Graphen zu layouten und zu zeichnen. Dies soll, genau wie die restlichen Analysefunktionen von Gradoop, verteilt über mehrere Rechner geschehen, um so trotz der inhärent aufwändigen Berechnung von Graph-Layouts auch für große Graphen in annehmbaren Zeiträumen nützliche Visualisierungen zu erhalten.

Der erste Schritt dabei war es, verschiedene bestehende Algorithmen für Force-directed Layouting in Flink für die Nutzung mit Gradoop zu implementieren. Unter diesen bestehenden Algorithmen sind sowohl allgemeine, grundlegende Algorithmen wie der Fruchterman-Reingold-Algorithmus (Kapitel 3.1), die als Grundlage für viele fortschrittlichere Algorithmen dienen, als auch Algorithmen, die speziell für die Ausführung in verteilten Umgebungen wie Flink entwickelt wurden (Kapitel 3.3, 3.2). Speziell beim Fruchterman-Reingold-Algorithmus stellt die Implementierung des, ursprünglich für zentralisierte Berechnungen ausgelegten, Algorithmus mit dem verteilten Ansatz von Flink eine besondere Herausforderung dar.

Der nächste Schritt nach der Fertigstellung der Implementierung der existierenden Algorithmen war das Entwerfen und Implementieren eigener Algorithmen, deren Ziel es ist die bestehenden Algorithmen hinsichtlich Laufzeit, Skalierbarkeit und Qualität der Ergebnisse zu verbessern. Einige der Ideen dahinter sind beispielsweise das Reduzieren des Berechnungsaufwandes durch Sampling von Knoten (Kapitel 4.2.1), die Generierung übersichtlicherer Layouts und das Beschleunigung der Berechnung durch geschicktes Gruppieren von Knoten (Kapitel 4.2.2) oder das Kombinieren existierender Algorithmen, um deren individuelle Vorteile zu vereinen (Kapitel 4.2.3).

Alle Implementierungen sollen hinsichtlich ihrer Laufzeiten und der Skalierbarkeit bezüglich steigender Graph-Größen und stärkerer Parallelisierung durch Einsatz von mehr Rechnern evaluiert und miteinander verglichen werden. Außerdem werden objektive Qualitätsmetriken benötigt, um die errechneten Layouts bezüglich ihrer Qualität vergleichen zu können. Auch diese Metriken werden im Rahmen dieser Arbeit in Flink implementiert. Da das letztendliche Ziel eines Graph-Layoutings meist das Erzeugen eines Bildes des Graphen ist, wird außerdem eine Möglichkeit benötigt, mittels Gradoop aus den errechneten Layouts fertige Bilder zu erzeugen.

Ausdrücklich nicht Ziel dieser Arbeit ist es, sämtliche bestehenden Graph-Layouting Algorithmen bezüglich Laufzeiten, Skalierbarkeit und Qualität zu übertreffen. Vielmehr geht es um die Erforschung der grundsätzlichen Machbarkeit von verteiltem Graph-Layouting mit Flink/Gradoop und Bewertung der Leistungsfähigkeit der verschiedenen Implementierungen.

# 2 Hintergrund

Dieses Kapitel erklärt einige Begriffe und Technologien, die für das Verständnis der restlichen Arbeit wichtig sind.

## 2.1 Theoretische Grundlagen

Im Folgenden werden zunächst einige grundlegende Begriffe definiert.

- **Graph**: Ein geordnetes Paar $G = (V, E)$. $V$ ist dabei eine Menge an Knoten und $E$ eine Menge an Kanten. Für die Menge der Kanten gilt: $E \subseteq [V]^2$. Die Kanten sind also zweielementige Untermengen von V [1]. Bei gerichteten Graphen sind die einzelnen Kanten keine Mengen, sondern geordnete Paare und die Reihenfolge der Elemente des Paares gibt die Richtung der Kante an. Beim Zeichnen eines Graphen werden Knoten üblicherweise als Punkte (oder Kreise) und Kanten als Linien (bei gerichteten Graphen Pfeile) dargestellt (siehe Abbildung 1).
- **Graph-Layout**: Ein Graph-Layout ist eine Funktion $l : V \to \mathbb{R}^d : x \mapsto l(x)$, die jedem Knoten eines Graphen eine Position in einem d-Dimensionalen Layout-Bereich zuweist.
- **Layouting Algorithmus**: Ein Layoutingalgorithmus ist eine Funktion $f : G \to L : x \mapsto f(x)$, die einem Graphen ein Layout zuordnet.
- **Dichte eines Graphen**: Die Dichte D eines ungerichteten Graphen ist definiert als $D = \frac{2*|E|}{|V|*(|V|-1)}$ und gibt Auskunft darüber wie nah die Anzahl der Kanten des Graphen an der maximal möglichen Anzahl an Kanten des Graphen liegt. Eine Dichte von 0 bedeutet, der Graph besitzt keine Kanten und eine Dichte von 1 bedeutet, dass jeder Knoten mit jedem anderen Knoten direkt über eine Kante verbunden ist.
- **Grad eines Knoten**: Der Grad (zumeist engl. Degree) eines Knoten gibt die Anzahl an Kanten an, die ein Knoten besitzt [1]. In gerichteten Graphen wird dabei zwischen In-Degree (Anzahl der eingehenden Kanten) und Out-Degree (Anzahl der ausgehenden Knoten) unterschieden.
- **Graphtheoretischer Abstand zweier Knoten**: Anzahl an Kanten im kürzesten Pfad zwischen beiden Knoten
- **Exzentrizität eines Knoten**: Größter Abstand zwischen dem fraglichen Knoten und allen anderen Knoten im Graphen [1]
- **Durchmesser eines Graphen**: Kleinste Exzentrizität aller Knoten im Graphen [1]
- **k-Nachbarschaft eines Knotens**: Menge aller Knoten mit einem graphtheoretischen Abstand von <=k vom fraglichen Knoten [2]

- **Cluster/Community**: Eine Menge an Knoten eines Graphen, deren Mitglieder untereinander dicht verbunden sind, aber mit Knoten außerhalb der Menge nur wenig verbunden sind. [3]
- **Clustering**: Unterteilen der Knoten des Graphen in Partitionen, so dass die Mitglieder einer Partition untereinander stärker verbunden sind, als die Partitionen untereinander. [3]
- **Sampling**: Auswahl einer zufälligen Teilmenge der Knoten oder Kanten des Graphen. Die Sampling-Rate $s \in \mathbb{R}, 0 \leq s \leq 1$ gibt dabei an, mit welcher Wahrscheinlichkeit ein Knoten (oder eine Kante) in die Teilmenge aufgenommen wird.

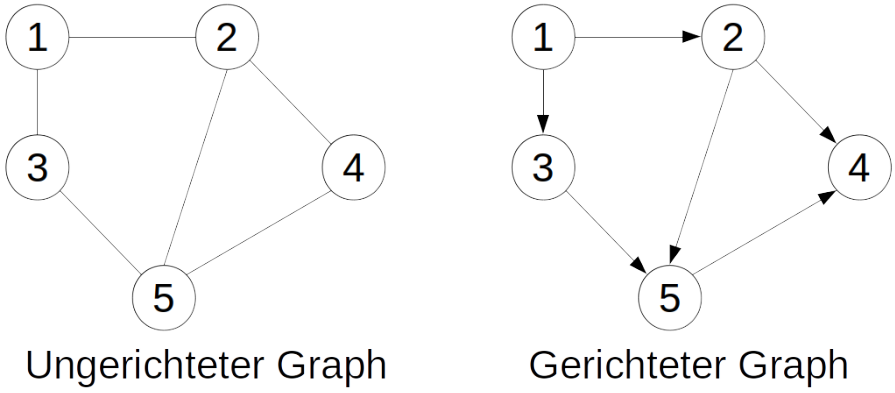


Abbildung 1: Visuelle Darstellung von Graphen

## 2.2 Apache Flink

Apache Flink [4] ist ein System für verteilte Berechnungen auf Stream- und Batchdaten. Ziel von Flink ist es, durch massive horizontale Skalierung enorm große Datenmengen möglichst in Echtzeit zu verarbeiten, ohne dafür auf spezialisierte (und damit teure) Spezialhardware zurückgreifen zu müssen.

Sämtliche Berechnungen mittels Flink werden als Datenfluss mit Quelle, diversen Transformationen und Senke angegeben. Ein Programmierer benutzt Java, Scala oder Python [5], um seinen Datenfluss zu definieren. Er gibt dabei eine oder mehrere Datenquellen an, aus denen die zu verarbeitenden Daten geladen werden sollen. Flink unterstützt dabei eine Vielzahl möglicher Quellen, wie beispielsweise verschiedene Datenbanken, HDFS oder simple Textdateien. Die geladenen Daten werden als sogenanntes DataSet repräsentiert. Anschließend werden Transformationen angegeben, die auf das DataSet angewendet werden sollen, um dieses in ein anderes DataSet zu transformieren. Dabei stehen viele

vorgefertigte Transformationen, wie z.B.map, Filter oder Join [6], zur Verfügung. Die größte Rolle spielen dabei allerdings vom Programmierer selbst erstellte Funktionen (genannt user defined functions, kurz UDF) die, z.B. als Teil einer Map-Operation, auf die Daten angewendet werden. Während die Benutzung eigener Funktionen bei anderen Datenflussframeworks (wie z.B. Tensorflow) eine absolute Ausnahme ist und teils massive Performance-Einbußen oder andere Probleme mit sich bringt, sind eigene Funktionen in Flink als zentrales Element vorgesehen. Abbildung 2 zeigt ein simples Beispiel für ein Flink-Programm, mit dem die Häufigkeit von Worten in einem Text ermittelt wird.

Wichtig ist hierbei, dass die tatsächlichen Operationen nicht sofort ausgeführt werden. Ruft der Programmierer beispielsweise die Map-Funktion in Java auf, wird diese nicht tatsächlich auf die Daten angewendet, sondern nur ihre Verwendung im logischen Ausführungsplan vermerkt. Erst wenn der Datenfluss vollständig beschrieben ist, folgt der nächste Schritt. Der logische Ausführungsplan wird an einen bestimmten Knoten im Rechencluster gesendet. Dieser erstellt daraus unter Einbeziehung der verfügbaren Knoten, der vom Nutzer eingestellten Parallelität und diverser Heuristiken einen physischen Ausführungsplan, welcher in kleine Teilaufgaben zerlegt auf die verfügbaren Rechenknoten verteilt wird.

Anschließend folgt die eigentliche Berechnung. Die dafür eingeteilten Knoten lesen die Daten aus der Quelle und schicken sie an den Knoten weiter, der für den nächsten Berechnungsschritt zuständig ist. Dieser führt seine Berechnungen aus und schickt seine Ergebnisse wiederum weiter, bis die fertig verarbeiteten Daten schließlich an ihrem vorgesehenen Ziel ankommen. Eine Besonderheit dabei ist, dass DataSets "lazy" sind. Ihr Inhalt wird erst dann bestimmt, sobald dieser tatsächlich benötigt wird. Dadurch beginnt die Ausführung eines Flink-Programms dann, wenn der Inhalt des Ergebnis-DataSets abgefragt wird (z.B. um diesen in eine Datei zu schreiben). Sobald dies der Fall ist, werden die Berechnungen ausgeführt, die nötig sind, um den Inhalt des DataSet zu erzeugen. Falls zum Erzeugen der Inhalt von anderen DataSets als Eingabe benötigt wird, werden zuvor auch deren Inhalte erzeugt. Dies setzt sich fort, bis letztendlich die Datenquelle erreicht wird, deren Inhalte dann eingelesen und als Ausgangswerte für die Berechnungen benutzt werden.

Die Details der verteilten Ausführung bleiben dem Programmierer verborgen, so dass sich die Benutzung von Flink (verglichen mit ähnlichen Frameworks) äußerst einfach gestaltet. Allerdings können nicht alle Operationen gleich performant verteilt ausgeführt werden. Einige Operationen, wie beispielsweise die Map-Operation können völlig problemlos auf allen beteiligten Rechnern komplett parallel durchgeführt werden, ohne dass dabei Kommunikationsaufwand zwischen den Rechnern anfällt. Andere Operationen, wie beispielsweise die Join-Operation, sind komplexer und können zwar über mehrere Rechner verteilt werden, müssen dafür aber große Mengen an Daten übertragen. Einige wenige Operationen schließlich, etwa bestimmte Aggregationen über den kompletten Datensatz,

müssen (zumindest in Teilen) auf einem einzelnen Rechner durchgeführt werden. Beim Arbeiten mit Flink sollte also immer darauf geachtet werden, möglichst Operationen zu verwenden, die sich gut parallel ausführen lassen.

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// get input data
DataSet<String> text = env.readTextFile( filePath: "input.txt");

// create the output DataSet
DataSet<Tuple2<String, Integer>> counts =
  // use flatMap to split the text into words
  text.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>> () {

    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
      // normalize and split the line
      String[] tokens = value.toLowerCase().split( s: "\\W+");

      // emit the pairs
      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(new Tuple2<>(token, 1));
        }
      }
    }
  }) //group by word and sum the count for each word
    .groupBy( ...fields: 0).sum( field: 1);

// write output to file
counts.writeAsText( filePath: "output.txt");

// run all output operations
env.execute();
```

Abbildung 2: Beispiel: WordCount mit Flink

## 2.3 Think Like a Vertex und Gelly

Think-Like-a-Vertex, auch genannt Vertex-Centric-Paradigm, ist ein Ansatz, um möglichst effizient (und evtl. verteilt) Berechnungen auf sehr großen Graphen auszuführen. Die erste veröffentlichte Implementierung eines Think-Like-a-Vertex-Frameworks war Google-Pregel, welches auf dem Bulk-Asynchronous-Programming-Modell [7] basiert. Die grundlegende Idee dabei ist es, Berechnungen aus der Perspektive eines einzelnen Knotens zu betrachten, anstatt den gesamten Graphen. Jeder Knoten des Graphen hat eine eindeutige Id und jeder Knoten und jede Kante können einen Wert zugewiesen bekommen. Die eigentliche Berechnung verläuft in Iterationen, genannt Super-Steps. In jedem einzelnen Super-Step kann jeder Knoten seine unmittelbare Nachbarschaft betrachten, also z.B. die Ids seiner direkten Nachbarn lesen. Des Weiteren kann er an alle (oder nur einzelne) seiner Nachbarn Nachrichten verschicken und die Nachrichten lesen, die an ihn von Nachbarn während der vorherigen Iteration geschickt werden.

Basierend auf diesen Nachrichten verändern Knoten ihren eigenen Wert und/oder verschicken neue Nachrichten. Die Berechnung endet nach einer festen Anzahl an Super-Steps, sobald keine Nachrichten mehr verschickt werden oder bei Erreichen eines, vom Nutzer festgelegten, Konvergenzkriteriums. Dieser Ansatz ist flexibel genug, um damit eine Vielzahl gängiger Graphen-Algorithmen umzusetzen (z.B. Pagerank) und dennoch aufgrund seiner hohen Lokalität so skalierbar, dass er auch auf sehr große Graphen und verteilte Berechnungssysteme anwendbar ist. Um die gute Skalierbarkeit beizubehalten, müssen mit diesem Ansatz umgesetzte Algorithmen allerdings einige Eigenschaften erfüllen. [2, Kap. 4.1]

- Jeder Knoten tauscht nur Nachrichten mit seinen direkten Nachbarn aus.
- Jeder Knoten speichert nur eine kleine Menge an Daten.
- Der Kommunikationsaufwand pro Iteration ist gering (z.B. linear in der Anzahl an Kanten im Graphen).

Neben dem grundlegenden Vertex-Centric Paragdigma gibt es noch leichte Abwandlungen davon, wie etwa Scatter-Gather oder Gather-Sum-Apply, welche zusätzliche Einschränkungen mit sich bringen (z.B. indem Nachrichtenempfang und Senden in getrennten Phasen geschehen), aber dadurch in bestimmten Fällen performanter sind. Eine bekanntes Think-Like-a-Vertex Framework ist beispielsweise Apache Giraph [8].

Gelly [9] ist Flink's API für Graph-Verarbeitung. Es bietet die Möglichkeit, Graphen aus Flink-Datensätzen zu erstellen und auf diesen graphspezifische Operationen auszuführen. Das Datenmodell entspricht im Wesentlichen dem von Pregel und Giraph. Jeder Knoten und jede Kante im Graph können einen Wert erhalten. Jedem Knoten ist darüber hinaus eine eindeutige Id zugeordnet. Neben simplen Operationen auf Knoten und Kanten sind vor allem die iterativen Graph-Operationen interessant. Gelly unterstützt dabei Algorithmen auf Basis von Vertex-Centric-Iterations, Scatter-Gather und Sum-Gather-Apply. Sämtliche verfügbaren Graph-Operationen sind dabei hinter den Kulissen aus Flink-Operationen zusammengesetzt.

## 2.4 Gradoop

Gradoop ist ein Framework zur skalierbaren Graph-Analyse auf Basis von Apache Flink, das an der Universität Leipzig entwickelt wird. Es bietet ein Datenmodell zum Repräsentieren von Graphen und Operatoren, die auf diese Graphen angewendet werden können. Durch Aneinanderreihung verschiedener Operatoren können komplexe Analyseworkflows deklarativ beschrieben werden [10].

Als Datenmodell kommt bei Gradoop das Extended Property Graph Model (EPGM) zum Einsatz. Dies ist eine Erweiterung der Property Graph Models (PGM) [11]. Im Property Graph Model ist ein Graph ein gerichteter Graph, dessen Knoten und Kanten mit je einem Label versehen werden können, welches Auskunft darüber gibt, welche Art von Objekt der Knoten (oder die Kante) repräsentiert. Beispielsweise könnte ein Knoten eine Person repräsentieren und

eine Kante eine Freundschaft zwischen zwei Personen. Außerdem besitzt jeder Knoten und jede Kante eine Menge an Schlüssel-Wert-Paaren, die sogenannten Properties, in denen beliebige Werte gespeichert werden können. Letztendlich besitzt jeder Knoten und jede Kante noch eine eindeutige Id.
Das EPGM erweitert nun das PGM dahingehend, dass mehrere einzelne Graphen (genannt logische Graphen) betrachtet werden, die sich Knoten und Kanten teilen können. Eine EPGM-Datenbank besteht aus einer Menge an Knoten, einer Menge von Kanten und einer Menge von logischen Graphen. Ein logischer Graph wiederum besteht aus einer Teilmenge der Knotenmenge und einer Teilmenge der Kantenmenge der Datenbank. Logische Graphen besitzen außerdem ebenfalls eine Id, ein Label und Properties [12].

Gradoop stellt eine Vielzahl von Operationen bereit, die auf einzelnen Graphen oder Gruppen von Graphen (genannt Graph Collections) angewendet werden. Je nach Operator ist das Ergebnis der Operation ebenfalls ein Graph, eine Graph Collection oder in seltenen Fällen ein einzelner skalarer Wert. Mittels der Graph Analytical Language (GRALA) können komplexe Analyseworkflows deklarativ durch die Verkettung von Operatoren in Java definiert werden [13]. Abbildung 3 zeigt einen simplen Workflow, der für einen Graphen, dessen Knoten (unter anderem) Personen repräsentieren, das Durchschnittsalter der Personen berechnet. Neben der Definition des Workflows mittels Java ist es mit BIGGR [14] auch möglich, Workflows mit Hilfe einer GUI graphisch zu definieren.

Ein Analyseworkflow besteht in der Regel aus dem Einlesen der Eingabedaten, einer Reihe von Operationen und dem anschließenden Abspeichern der Ergebnisse. Gradoop liefert eine große Menge an Graph-Operatoren. Darunter sind einerseits vergleichsweise simple Operatoren zum Auswählen, Modifizieren oder Aggregieren von Knoten und Kanten. Andererseits gibt es auch einige Operatoren für komplexe Analysen, wie beispielsweise das Finden von Mustern in Graphen [15] und das Gruppieren von Graph-Elementen [16]. Außerdem bringt Gradoop direkt Implementierungen gängiger Analysealgorithmen, wie beispielsweise PageRank oder Label Propagation mit [13].

Gradoop unterstützt eine Vielzahl von Ein- und Ausgabeformaten für Graphen. Neben einfachen dateibasierten Formaten - wie json, csv und dot - kann Gradoop auch direkt mit Apache HBase und Apache Accumulo arbeiten. Außerdem ist es mögliche, eigene Datenquellen und Datensenken zu implementieren, um weitere Formate zu unterstützen.

In Gradoop werden Graphen intern durch Flink-DataSets repräsentiert und jede Operation ist intern durch eine Reihe von Transformationen auf den DataSets des Graphen umgesetzt [13]. Dadurch übernimmt Gradoop viele Vorteile von Flink, wie die automatische verteilte Ausführung von Berechnungen. Die Umsetzung des Datenmodells und der Operationen in Flink erfolgt transparent für den Nutzer und wird normalerweise durch das Arbeiten mit GRALA vor dem Nutzer verborgen. Allerdings ist es trotzdem möglich, direkt auf das zugrundeliegende Datenmodell zuzugreifen, um beispielsweise mittels FLink eigene Operatoren für Graphen zu implementieren.

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
GradoopFlinkConfig cfg = GradoopFlinkConfig.createConfig(env);

// load input graph
CSVDataSource source = new CSVDataSource( csvPath: "hdfs://input_dataset", cfg);
LogicalGraph input = source.getLogicalGraph();

LogicalGraph result = input
  // only consider vertices of type Person
  .vertexInducedSubgraph(new ByLabel<>("Person"))
  // calculate the average of the age property
  .aggregate(new AverageVertexProperty( propertyKey: "age",  aggregatePropertyKey: "ageavg"));

// store output graph
CSVDataSink sink = new CSVDataSink( csvPath: "hdfs://results", cfg);
result.writeTo(sink);

// run computation
env.execute();
```

Abbildung 3: Beispiel: Altersdurchschnitt mit Gradoop

## 2.5 Force-directed Layout

Einige der flexibelsten Algorithmen zum Berechnen von Graph-Layouts gehören zur Kategorie der Force-Directed Algorithmen [17]. Solche Algorithmen berechnen Layouts ausschließlich auf Grundlage der Struktur eines Graphen und benutzen dabei meist die folgenden zwei grundlegenden Regeln:

1. Knoten, die durch eine Kante verbunden sind, sollten dicht beieinander gezeichnet werden.

2. Knoten sollten nicht zu nah beieinander gezeichnet werden.

Grundsätzlich werden diese beiden Regeln durch Anziehungskräfte zwischen durch Kanten verbundenen Knoten und Abstoßungskräfte zwischen sämtlichen Knoten modelliert, die über mehrere Iterationen auf den Graphen angewendet werden, bis ein Kräftegleichgewicht zu Stande kommt.
Einer der bekanntesten Ansätze dieser Art ist das Fruchterman-Reingold-Model [18], bei dem sich Knoten verhalten wie gleich-geladene elektrische Partikel und Kanten sich wie mechanische Federn verhalten.

Aufgrund ihrer einfachen Verständlichkeit, der leichten Implementierung in Software und den optisch ansprechenden Ergebnissen werden Force-Directed Layout Algorithmen häufig eingesetzt [17], [19].

Abbildung 4 zeigt ein zufälliges und ein mittels Force-directed Layouting erstelltes Layout des selben Graphen.

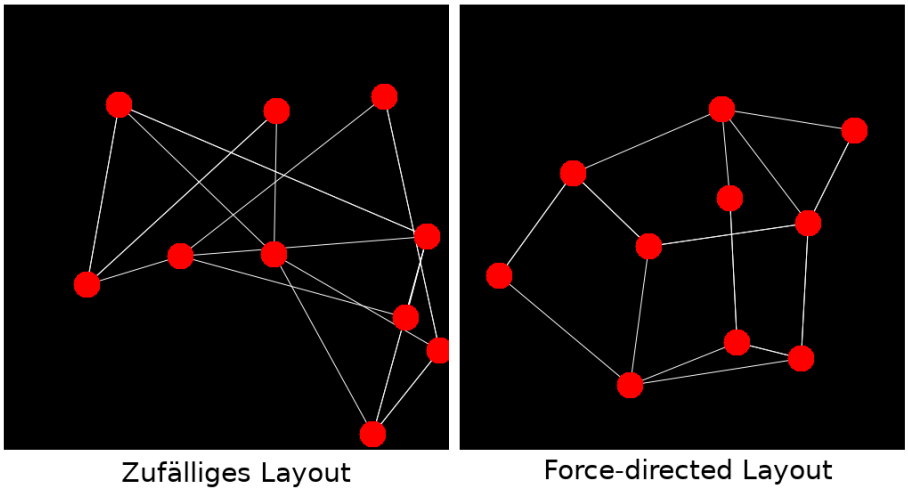


Abbildung 4: Zufälliges und Force-directed Layout

Das Force-directed Layout ist optisch erheblich ansprechender als das zufällige Layout und gibt die Struktur des Graphen gut wieder, was unter anderem an der geringen Anzahl an Kantenkreuzungen zu erkennen ist.

# 3 Related Work

Am Anfang des Force-directed Graph-Layouting standen zentralisierte Algorithmen, wie der von Fruchterman und Reingold [18]. Da allerdings die Abstoßungskräfte zwischen allen Knoten des Graphen berechnet werden müssen, sind solche Algorithmen sehr rechenintensiv und lassen sich daher nur für relativ kleine Graphen sinnvoll durchführen.

Um die Berechnung auch für größere Graphen zu ermöglichen, wurde anschließend versucht, die Berechnungen zu parallelisieren. So entwickelten beispielsweise Chae et al. [20] eine Methode, die Knoten gleichmäßig auf verschiedene Displays zu verteilen, wobei dann jeweils ein Prozessor für die Berechnung der Positionen der Knoten eines Displays zuständig ist. Damit konnten Visualisierungen für Graphen mit einigen Tausend Knoten berechnet werden.

Tikhonova und Ma [21] entwickelten einen parallelen Force-directed Algorithmus, mit dem es ihnen gelang, mithilfe von 32 Prozessoren einen Graphen mit 260.385 Knoten in etwa 40 Minuten zu layouten.

In jüngerer Vergangenheit wurde versucht, Graph-Visualisierung mit Hilfe der aufkommenden Big-Data-Frameworks umzusetzen. So implementierten Hinge und Auber [22] einen Force-directed Algorithmus auf Basis von Apache Spark, einem Open-Source Map-Reduce Framework, welches auf iterative Berechnungen ausgelegt ist.
Arleo et al. [2] entwickelten einen verteilten Graph-Layouting Algorithmus auf Basis des Think-Like-a-Vertex Paradigmas unter Verwendung des Open-Source Graph-Verarbeitungsframeworks Giraph. In ihren Experimenten konnten sie damit einen Graphen mit einer Million Knoten zeichnen und benötigten dafür nur 8 Minuten und Cloud-Rechenleistung im Wert von einem Dollar.

Kurz darauf entwickelten Arleo et al. [23] eine verbesserte Version ihres Algorithmus, indem sie diesen um einen Multi-Level-Ansatz erweiterten. Dabei wird der ursprüngliche Graph zuerst iterativ reduziert, indem benachbarte Knoten mittels des Solar-Merger-Algorithmus zusammengefasst werden. Anschließend findet das Force-directed Layouting auf dem reduzierten Graphen statt, was aufgrund der reduzierten Knoten-Anzahl erheblich performanter ist. Letztendlich wird die anfängliche Reduzierung rückgängig gemacht und das Layout das erweiterten Graphen basierend auf dem Layout des vorangehenden reduzierten Graphen berechnet.

## 3.1 Fruchterman-Reingold-Algorithmus

Der Fruchterman-Reingold-Algorithmus [18] ist ein vergleichsweise simpler und äußerst bekannter Algorithmus für Force-directed-Layouting und bildet darüber hinaus die Grundlage für viele weitere Algorithmen. Damit liefert der Algorithmus

eine Vergleichsbasis in Sachen Performance und Layout-Qualität für folgende Algorithmen. Weiterhin kann sein Code zu großen Teilen für andere Algorithmen wiederverwendet werden.

Der Algorithmus ist im Wesentlichen die direkte Umsetzung der in Kapitel 2.5 genannten Prinzipien. Alle Knoten stoßen sich gegenseitig voneinander ab und durch eine Kante verbundene Knoten ziehen einander an.

Die Formeln für die Anziehungskräfte ($f_a$) und die Abstoßungskräfte ($f_r$) lauten:

$$f_a(v1, v2) = \frac{d(v1, v2)^2}{k} \tag{1}$$

$$f_r(v1, v2) = \frac{-k^2}{d(v1, v2)} \tag{2}$$

Dabei steht $d()$ für die Entfernung zwischen den Knoten und $k$ ist ein frei wählbarer Parameter, der angibt, in welchem Abstand Anziehungs- und Abstoßungs-Kräfte zwischen zwei verbundenen Knoten gleich groß sind. Es ist klar erkennbar, dass die Anziehungskräft, zwischen verbundenen Knoten, mit zunehmender Entfernung schnell größer wird, die Abstoßungskraft jedoch abnimmt. Bei abnehmenden Entfernungen hingegen, wird die Abstoßungskraft sehr groß und die Anziehungskraft geringer.

Um ein geeignetes $k$ zu berechnen, wird folgende Formel verwendet:

$$k = \sqrt{\frac{layoutBreite * layoutHöhe}{anzahlKnoten}} \tag{3}$$

Bei diesem Algorithmus handelt es sich es im Kern um eine diskrete Physik-Simulation. Als solche muss sie mit einer vorgegebenen Schrittweite arbeiten, die angibt, wie viel Zeit zwischen den einzelnen Simulationsschritten liegt. In diesem konkreten Fall stellt sich also die Frage, wie weit ein Knoten in einer Iteration bewegt werden soll, wenn auf ihn eine Kraft einer bestimmten Größe einwirkt.

Zu große Schrittweiten führen zu großen Fehlern in der Simulation. So könnte hier beispielsweise ein Knoten über sein eigentliches Ziel hinausschießen und sich am Ende weiter davon entfernt befinden als zuvor. Zu kleine Schrittweiten führen zu unnötig langen Laufzeiten.

Eine relativ übliche Herangehensweise ist das sogenannte Simulated Annealing. Dabei beginnt man anfangs mit einer großen Schrittweite und reduziert die Schrittweite stückweise über die Laufzeit des Algorithmus. So besteht am Anfang genug Bewegungsfreiheit, um lokale Minima zu überwinden und gegen Ende des Algorithmus werden nur noch kleinere Anpassungen durchgeführt. Das genaue Verfahren zur stückweisen Reduzierung (der sogenannte Abkühl-Schedule) hat hierbei starke Auswirkungen auf die Qualität des Ergebnisses und ist oft abhängig von der spezifischen Aufgabenstellung.

Fruchterman und Reingold schlagen vor, Simulated Annealing zu benutzen und zwar so, dass die Verschiebung eines Knotens in einer Iteration auf die aktuelle Schrittweite beschränkt wird. Ist die Verschiebung kleiner als die aktuelle Schrittweite, bleibt sie unverändert.
Leider wird von den Autoren kein spezifischer Schedule angegeben. Die Berechnung der Position für die nächste Iteration geschieht wie folgt:

```
beschränkt(x) = ( |x| < schrittWeite ) ? x : x / |x| * schrittWeite
position(v1,t+1) = position(v1,t) + beschränkt( kräfte(v1,t) )
```

Beim naiven Ansatz zur Berechnung der Abstoßungskräfte müssten diese zwischen allen Knoten des Graphen berechnet werden. Das würde zu einer quadratischen Komplexität ( $O(|V|^2)$ ) führen und ist damit für die meisten Graphen mit praxisüblicher Größe völlig ungeeignet.

Um dies zu vermeiden, haben Fruchterman und Reingold sich eine Optimierungsstrategie überlegt. Die Grundannahme dabei ist, dass die Abstoßungskräfte von weit entfernten Knoten so gering sind, dass sie vernachlässigt werden können. Daher werden Abstoßungskräfte nur zwischen Knoten berechnet, die weniger als 2k voneinander entfernt sind.

Wie dies umgesetzt wird, ist in Abbildung 5 zu sehen. Der Layoutbereich wird in Zellen unterteilt, die jeweils 2k hoch und breit sind. Für die Abstoßungskräfte eines Knotens werden nun nur noch Knoten in der selben oder in einer direkt benachbarten Zelle betrachtet. Für den Knoten v sind also nur noch die Knoten q und s relevant. Knoten r liegt nicht in einer benachbarten Zelle und kann daher ignoriert werden. Innerhalb der betrachteten Zellen befinden sich noch Knoten, die mehr als 2k vom aktuellen Knoten entfernt sind. Diese müssen noch nachträglich entfernt werden (im Beispiel Knoten s). Andernfalls würde das Gitter sichtbare Verzerrungen im Layout verursachen.

Im Optimalfall (die Knoten sind gleichmäßig über den Layoutbereich verteilt) führt dieser Ansatz zu einer linearen Komplexität. Sollten die Knoten aber ungünstig verteilt sein (sehr nah beieinander liegen), bleibt es bei der quadratischen Komplexität. [24]

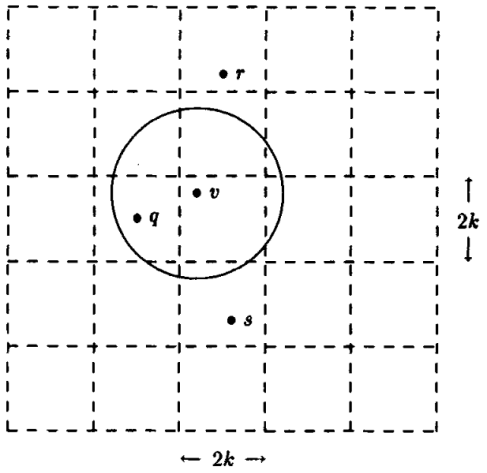


Abbildung 5: Rastereinteilung des Layoutbereiches [18]

## 3.2 Centroid basierter Layouting-Algorithmus in Spark

In "Distributed Graph Layout with Spark" [22] entwickeln die Autoren David Auber und Antoine Hinge einen Force-directed Layouting-Algorithmus unter Verwendung von Apache Spark. Da sich Apache Spark und Apache Flink stark ähneln, eignet sich dieser Algorithmus auch gut zur Umsetzung in Flink. Darüber hinaus verspricht der Ansatz eine gute Skalierbarkeit, was zusätzlich für eine Implementierung und Erprobung im Rahmen dieser Arbeit sprach.

Die Grundidee von Auber und Hinge ist es, nur noch einen Teil der Abstoßungskräfte zu berechnen. Anstatt Abstoßungskräfte für jeden Knoten mit jedem anderen zu berechnen, werden Abstoßungskräfte nur zwischen Knoten und speziellen Punkten, die eine Ansammlung von Knoten repräsentieren, berechnet. Diese Repräsentanten werden über ein k-means Clustering [25] erzeugt. Die bei dem Clustering ermittelten Clusterzentren repräsentieren jeweils alle Knoten in ihrem Cluster.

Der genaue Ablauf ist wie folgt: Anfangs werden eine bestimmte Anzahl an Knoten zufällig ausgewählt. Deren Positionen bilden die initialen Cluster-Zentren. In jeder einzelnen Iteration wird nun jeder Knoten dem ihm am nächsten liegenden Cluster-Zentrum zugeordnet und die Position des jeweiligen Clusterzentrums wird auf den Durchschnitt der Positionen der ihm zugeordneten Knoten gesetzt. Für jeden Knoten werden nun die Abstoßungen mit allen Cluster-Zentren berechnet. Zusätzlich wird auch noch eine Abstoßung vom Cluster-Mittelpunkt (dem Durchschnitt der Position aller Knoten) aus berechnet. Die Berechnung der Anziehungskräfte und die Anwendung der Kräfte erfolgen exakt wie beim Fruchterman-Reingold-Layouter.

Um eine gute Balance zwischen Laufzeit und Qualität zu bekommen, ist es wichtig, dass die Anzahl der Cluster-Zentren zur Anzahl von Knoten im Graphen passt. Das erreichen Auber und Hinge [22], indem sie sicherstellen, dass jedem Cluster-Zentrum immer zwischen 0.25% und 5% der Knoten des Graphen zugeordnet sind. Cluster-Zentren, denen mehr als 5% der Knoten zugeordnet sind, werden in zwei separate Cluster-Zentren aufgeteilt. Cluster-Zentren mit weniger als 0.25% der Knoten werden entfernt.

## 3.3 GiLa

GiLa ist ein von Arleo et al. [2] vorgestellter Layouting-Algorithmus auf Basis des Think-like-a-Vertex Paradigmas (Kapitel 2.3) und funktioniert im Prinzip ähnlich wie der Fruchterman-Reingold-Algorithmus. Es berechnet ebenfalls ein Graph-Layout über Anziehungs- und Abstoßungskräfte und benutzt dafür die gleichen Formeln. Der entscheidende Unterschied ist, das Abstoßungskräfte nur zwischen einigen wenigen Knoten berechnet werden. Während sich beim Fruchterman-Reingold-Algorithmus alle Knoten abstoßen, die nah genug beieinander liegen, gehen die GiLa-Autoren davon aus, dass die graphentheoretische Distanz zwischen Knoten eine ausreichend gute Näherung der Entfernung zwischen Knoten ist. Bei GiLa stoßen sich also Knoten ab, die über weniger als n Kanten miteinander verbunden sind.

Zu Beginn des Algorithmus werden alle Knoten mit einem Grad von 1 (sie haben lediglich eine eingehende/ausgehende Kante) aus dem Graphen entfernt. Durch diese Reduzierung der Knoten soll das Layouting beschleunigt werden. Die entfernten Knoten werden am Ende des Algorithmus wieder eingefügt.

Die verbleibenden Knoten werden nun auf alle beteiligten Rechner-Knoten verteilt. Durch die Benutzung des Spinner-Algorithmus [26] wird sichergestellt, dass sich Knoten, die durch eine Kante verbunden sind, möglichst auf dem selben Rechner-Knoten befinden. Da Nachrichten nur zwischen graphentheoretisch nah bei einander liegenden Knoten ausgetauscht werden, lässt sich so recht viel Netzwerkverkehr zwischen den Rechner-Knoten einsparen.

Gemäß dem Think-like-a-Vertex Paradigma können nur Knoten miteinander kommunizieren, die direkte Nachbarn voneinander sind. Um zwecks Kräfteberechnung von alle Knoten zu finden, die weniger als n Kanten entfernt sind, wird kontrolliertes Flooding eingesetzt [2]. Jeder Knoten sendet eine Nachricht mit seiner aktuellen Position und einer TTL (Time to live. Maximale Anzahl an Weiterleitungen für diese Nachricht) von n an all seine Nachbarn. Diese prüfen, ob sie diese Nachricht bereits empfangen haben (in einer vorherigen Iteration oder von einem gemeinsamen Nachbarn) und wenn nicht, benutzen sie die empfangene Position, um die aktuell auf sie einwirkenden Kräfte zu berechnen, reduzieren die TTL um eins und leiten die Nachricht dann (sofern die TTL dann nicht Null ist) an all ihre Nachbarn weiter.

Das Algorithmus endet, sobald sich in einer Iteration mehr als 15% der Knoten um weniger als einen festgelegten Wert bewegen.

GiLa verwendet ebenfalls Simulated Annealing zur Regulierung der Schrittweite und verwendet dafür die folgende Formel:

$$schrittweite(iteration) = \sqrt{\frac{knotenAnzahl}{layoutBreite/layoutHöhe}} * k * 0.93^{iteration} \quad (4)$$

# 4 Implementierung

Neben dem bloßen Implementieren und Evaluieren verschiedener Algorithmen ist es eines der Hauptziele dieser Arbeit, ein praxistaugliches Feature zum Layouten und Visualisieren von Graphen für Gradoop zu entwickeln. Daher wurde während der gesamten Entwicklung Wert darauf gelegt, die Standards des Gradoop-Projektes bezüglich Struktur, Codequalität und Dokumentation einzuhalten, um am Ende dieser Arbeit eine problemlose Integration in den Hauptentwicklungszweig von Gradoop zu ermöglichen. Außerdem wurde von Anfang an eine gute Testabdeckung für den geschriebenen Code angestrebt. Nebenbei erwies sich dies während der Entwicklung, insbesondere bei der Umsetzung von Optimierungen, als äußerst hilfreich, da auftretende Fehler so schnell entdeckt und beseitigt werden konnten.

Außerdem wurde großer Wert auf einfache Verwendbarkeit der Layouting-Funktionalität gelegt. Alle implementierten Layouting-Algorithmen teilen ein gemeinsames Interface und sind daher beliebig austauschbar. Des weiteren bieten alle Algorithmen diverse Konfigurationsmöglichkeiten, um sie einfach den jeweiligen Bedürfnissen des Nutzers anpassen zu können. Dennoch sind die meisten dieser Konfigurationsmöglichkeiten rein optional und mit passenden Standartwerten versehen, so dass ein Nutzer, der einfach ein Bild von seinem Graphen generieren möchte, sich nicht mit komplexen Details beschäftigen muss. Die einzigen zwingend notwendigen Angaben für den grundlegenden Fruchterman-Reingold-Algorithmus sind die grobe Anzahl an Knoten im zu layoutenden Graphen und die Anzahl der zu absolvierenden Iterationen.

Die grobe Anzahl an Knoten wird zur Bestimmung verschiedener Standardparameter benötigt und muss daher noch vor der Flink-Ausführung bekannt sein. Eine automatische Bestimmung zur Laufzeit wäre zwar wahrscheinlich möglich, würde die Implementierung aber drastisch verkomplizieren und schwer wartbar machen, weshalb ganz bewusst darauf verzichtet wurde. In der Regel sollte ein Benutzer die grobe Anzahl an Knoten in dem von ihm analysierten Graphen kennen, zumal die an den Algorithmus übergebene Anzahl nicht 100% korrekt sein muss. Im Zweifelsfall ist es aber dennoch möglich die Anzahl der Knoten automatisch von Flink bestimmen zu lassen und die ermittelte Anzahl an den Layouting-Algorithmus zu übergeben. Der einzige Nachteil dabei ist, dass dafür eine separate Flink-Ausführung nötig ist.

Ein Beispiel für die Nutzung der Knoten-Anzahl zu Bestimmung von Standardwerten ist die Auswahl der Größe des Layouting-Bereiches. Die Formel zur Berechnung von $k$ für den Fruchterman-Reingold-Algorithmus (Formel 3) kann auch benutzt werden, um aus einem festgelegten k (in dieser Implementierung gilt als Standard k=100) und der Anzahl der Knoten, die Größe des Layoutbereichs zu bestimmen. Auf diese Weise braucht der Nutzer nur die (sowieso bekannte) Anzahl an Knoten an den Algorithmus zu übergeben und muss sich nicht um weitere Details kümmern. Natürlich wäre es auch möglich, die Größe festzulegen

und k dynamisch zu bestimmen, allerdings ist es schwierig, eine gute Größe für alle Graphen zu wählen, da diese sich in ihrer Größe stark unterscheiden können. Ist der Bereich zu groß, würden sich Graphen mit mehreren Zusammenhangskomponenten zu stark ausbreiten. Ist der Bereich zu klein, würde k bei großen Graphen so klein werden, dass Probleme mit der Gleitkommagenauigkeit auftreten können. Daher scheint dieser Ansatz zielführend, zumal der Nutzer die Standartwerte jederzeit durch eigene ersetzen kann.

Auf eine Funktion zur automatischen Bestimmung der benötigten Iterationen wurde bewusst verzichtet. Jede Möglichkeit, diese umzusetzen (wie etwa ein automatischer Abbruch sobald die Summe der aktuell errechneten Kräfte einen bestimmten Grenzwert unterschreitet) würde es erfordern, einen speziellen Wert (wie etwa den genannten Grenzwert) festzulegen. Diese willkürliche Festlegung würde dazu führen, dass die benötigten Laufzeiten zur Errechnung eines Layouts nur schlecht nachvollziehbar sind und außerdem einen Vergleich verschiedener Algorithmen oder auch mit den Ergebnissen anderer Studien deutlich erschweren. Beispielsweise unterscheiden sich die Laufzeiten, je nach genauer Wahl des Abbruchkriteriums, erheblich. Außerdem ist es für bestimmte Graphen möglich, dass das Erreichen eines festen Abbruchkriteriums so viel Zeit in Anspruch nimmt, dass der Nutzer in absehbarer Zeit kein Ergebnis erhält und dabei auch keine Möglichkeit, hat die Berechnung vorzeitig abzubrechen. Abschließend erleichtert eine von Anfang an bekannte Anzahl von Iterationen das Finden eines geeigneten Simulated Annealing Schedules. Ohne von vorneherein bekannte Iterationsanzahl wäre etwa der in Kapitel 4.1.2 beschriebene Schedule nicht nutzbar. Für die Zukunft ist es allerdings nicht ausgeschlossen, eine optionale Funktion zur automatischen Bestimmung der benötigten Iterationen in Gradoop zu integrieren. Diese sollte aber immer optional bleiben.

Natürlich benötigen einige Algorithmen prinzipbedingt noch weitere Eingaben des Nutzers. So muss etwa dem Sampling-Layouter eine Samplingrate vorgegeben werden. Die passende Wahl der Parameter kann dem Nutzer allerdings durch gute Dokumentation erleichtert werden.

Da es sich beim Layouting von großen Graphen um ein rechenintensives Problem handelt und sich daher ein hoher Rechenaufwand nicht vermeiden lässt, wurde großer Wert darauf gelegt, die Implementierung der Algorithmen so effizient wie möglich zu gestalten. Speziell bei den Stellen, welche beim Layouting stellenweise mehrere Milliarden mal ausgeführt werden (etwa die Funktionen zum Berechnen der zwischen Knoten auftretenden Kräfte), können bereits kleinere Verbesserungen am Ende zu deutlich geringeren Laufzeiten führen. Leider sind, bedingt durch die verwendeten Technologien, nicht allzu drastische Optimierungen möglich. Bei der Verwendung von Java etwa sind kaum low-level Optimierungen möglich und man ist auf die Optimierungen des Java-Compilers angewiesen. Noch drastischer ist die Situation bei Flink. Flink nimmt den Entwicklern viel Arbeit ab und kümmert sich automatisch um viele Dinge, wie z.B. das Erstellen und Optimieren von Ausführungsplänen und die verteilte Ausführung dieser. Dies ist zwar sehr bequem, gibt Entwicklern aber dafür kaum Möglichkeiten,

händisch optimierend einzugreifen. Auch ist nie ausgeschlossen, dass händische Optimierungsversuche durch den internen Flink-Optimizer zunichte gemacht werden oder gar gegenteilige Effekte haben.

Allerdings gibt es durchaus einige Techniken, die sich verwenden lassen, um Flink-Programme zu beschleunigen. Für die Implementierung der Algorithmen wurden die in [27] aufgeführten Ratschläge weitestgehend berücksichtigt. So bauen die intern verwendete Datentypen auf den Flink-Tupel-Klassen auf, was zu einer merklichen Geschwindigkeitssteigerung geführt hat. Außerdem wird an besonders häufig ausgeführten Code-Stellen, wie den Kräfteberechnungsfunktionen, so gut wie möglich auf das Allozieren von Java-Objekten verzichtet, um den damit verbundenen Overhead zu vermeiden und den Java-Garbage-Collector zu entlasten. Stellenweise findet in kritischen Bereichen des Codes zur Ausführungszeit keine einzige Objektallozierung mehr statt. Ermöglicht wird dies unter anderem durch eine Eigenheit von mittels Flink ausgeführten Java-Funktionen. Die Eingabe- und Ausgabeobjekte der Funktionen werden von Flink vor der Ausführung der Funktion deserialisiert und nach der Ausführung wieder serialisiert. Dadurch ist es möglich, innerhalb der Funktion Ein- und Ausgabeobjekte in nachfolgenden Aufrufen der Funktion wieder zu verwenden, ohne die Ausgaben vorheriger Aufrufe nachträglich zu verändern.

```
Thing outputthing = new Thing();
public Thing map(int input){
    outputthing.value = input;
    return outputthing;
}
```

Betrachten wir beispielsweise die obenstehende Map-Funktion. Wird diese in Java ohne Flink auf jedes Element der Liste [1,2,3] angewendet, wäre die Ausgabe die Liste [Thing(3),Thing(3),Thing(3)]. Dies ist darin begründet, dass die Funktion immer nur einen Zeiger auf ein und das selbe Objekt zurück gibt und nachfolgende Aufrufe den Wert, auf den diese Zeiger zeigen verändert. Wird diese Funktion aber mittels Flink ausgeführt, ist das Ergebnis die Liste [Thing(1),Thing(2),Thing(3)], da jeder Rückgabewert direkt serialisiert wird und damit durch nachfolgende Funktionsaufrufe nicht mehr verändert wird.

Auf diese Weise können in Flink Objekte sehr effizient wiederverwendet und viele Objekt-Allokationen eingespart werden. Allerdings ist dieses Verhalten eher ungewohnt und kann die Lesbarkeit von Code verschlechtern. Außerdem müssen besondere Maßnahmen ergriffen werden, falls eine solche Funktion (etwa im Rahmen von Tests) ohne Flink ausgeführt wird, da es sonst zu unerwarteten Ergebnissen (falschen Ausgaben) kommt.

## 4.1 Implementierung existierender Layouting-Algorithmen

### 4.1.1 Random-Layouter

Um für die folgenden Implementierungen einen Vergleichswert zu haben wurde zuerst einmal der einfachst mögliche Layouting-Algorithmus implementiert. Bei diesem werden jedem Knoten zufällige Koordinaten innerhalb eines vorgegebenen Bereiches zugewiesen. Dies lässt sich in Flink mit einem einzigen Map-Task äußerst performant implementieren. Die dabei entstehenden Layouts sind für die meisten praktischen Zwecke nicht verwendbar, da sie die interne Struktur des Graphen in keinster Weise wiedergeben, wodurch alle Graphen nahezu gleich aussehen. Allerdings starten die Meisten Layouting-Algorithmen mit einem zufälligen Layout, wofür dieser Layouter wiederverwendet werden kann. Außerdem hilft er dabei, mit einer Eigenheit von Performance-Messungen in Flink umzugehen. In Flink kann lediglich die Laufzeit eines kompletten Programms (Daten laden, verarbeiten, speichern) gemessen werden. Dadurch verfälschen die Operationen zum Laden und Speichern (die je nach Datenformat und Größe des Datensatzes durchaus relevant sein können) die Laufzeitmessungen. Die einfachste Lösung dafür ist es, die Laufzeit des Random-Layouters (bei dem nahezu 100% der Laufzeit auf Laden und Speichern entfallen) von der Laufzeit des untersuchten Algorithmus abzuziehen. Dadurch erhält man die reine Laufzeit der Layouting-Operation.

### 4.1.2 Fruchterman-Reingold-Layouter

In Abbildung 6 ist Der Datenfluss der Flink-Implementierung des Fruchterman-Reingold-Algorithmus (siehe Kapitel 3.1) schematisch dargestellt. Der Algorithmus erhält als Eingabe die Knoten und Kanten des Graphen im Gradoop-Format (Vertices und Edges). Zu diesem Zeitpunkt wurden den Knoten bereits initiale Koordinaten zugewiesen (im einfachsten Fall durch den Random-Layouter). Nun werden die Knoten und Kanten zuerst über jeweils einen Map-Task in ein internes Format (LVertices und LEdges) konvertiert. Das interne Format enthält nur die Informationen aus den Knoten und Kanten die für das Layouting benötigt werden. Das reduziert die Datenmengen, die in den folgenden Schritten zwischen den einzelnen Rechenknoten des Flink-Clusters übertragen werden müssen und erhöht dadurch die Ausführungsgeschwindigkeit.

Um die Anziehungskräfte zwischen verbundenen Knoten zu berechnen werden die Knoten zwei mal mit den Kanten gejoint (einmal für die Start- und einmal für die Zielknoten), um Paare aus Start- und Zielknoten für jede existierende Kante zu erhalten (LVertex,LVertex). Zwischen diesen Paaren werden nun die auftretenden Anziehungskräfte errechnet (A-Forces).

Die Abstoßungskräfte zwischen Knoten können (ohne die Kanten) direkt aus den Knoten berechnet werden. Die Implementierung der Abstoßungsberechnung ist komplexer und wird daher in Abbildung 7 gesondert dargestellt. Aufgrund

der Tatsachen, dass die Zell-Nachbarschaften sich überlappen und die Nachbarschaftsbeziehung nicht transitiv ist (Zelle b ist Nachbar von Zelle a und von Zelle c, aber a und c sind keine Nachbarn von einander) gestaltete sich die Umsetzung dieser Idee in Flink schwierig.

Umgesetzt wird dies letztendlich, indem als Erstes jeder Knoten mit der ID der Zelle versehen wird, in der er sich befindet (Assign Cellid). Diese Id setzt sich aus X und Y Koordinate der Zelle im Raster zusammen, weshalb sich die IDs der Nachbarzellen leicht berechnen lassen. Anschließend wird das Knoten-Dataset mehrfach mit sich selbst gejoint. Dabei dient die Zell-Id als Schlüssel. Zuerst werden die Knoten mit allen Knoten gejoint, welche die selbe Zell-Id haben, also in der selben Zelle liegen. Anschließend werden die Knoten mit allen Knoten gejoint, deren Zell-Id die ID der rechten Nachbarzelle ist. Das Ganze wird nun mit allen weiteren möglichen Nachbarrichtungen wiederholt. Letztendlich wären also 9 Joins nötig, um jeden Knoten mit allen Knoten in seinen Nachbarzellen (und der eigenen Zelle) zusammen zu bringen.

Praktischerweise lassen sich hier Symmetrieeffekte ausnutzen. Angenommen, Zelle b ist ist der rechte Nachbar von Zelle a, dann ist es egal, ob Zelle a mit ihrem rechten Nachbar oder Zelle b mit ihrem linken Nachbarn gejoint wird. Abgesehen von der Reihenfolge der Knoten im Join-Tupel ist das Ergebnis identisch. Berechnet man bei einem solchen Join also direkt die Kräfte für beide beteilige Knoten gleichzeitig, lässt sich der dazu symmetrische Join einsparen. Somit sind anstatt 9 nur noch 5 Joins nötig, was die Laufzeit des Algorithmus spürbar senkt.

In Flink sind an sich ausschließlich Equi-Joins möglich. Für die Joins mit Nachbarzellen wird daher ein CustomKeyExtractor verwendet. Soll beispielsweise mit der rechten Nachbarzelle gejoint werden, extrahiert der Extractor für die linke Seite des Joins nicht die tatsächlich gespeicherte Id, sondern rechnet die gespeicherte Id in die Id der rechten Nachbarzelle um (der X-Koordinaten-Teil der Id wird um 1 erhöht). Die Ids der rechten Seite des Joins werden unverändert extrahiert. Wenn nun die extrahierten Ids übereinstimmen, bedeutet das, dass der eine Knoten in der gesuchten Nachbarzelle des anderen Knotens liegt.

Wird beispielsweise das DataSet der Knoten, über die in Tabelle 1 aufgeführten Keys mit sich selbst gejoint, dann ist das Ergebnis die Menge `{{K1,K3},{K1,K8},{K4,K1},{K6,K3}}`. In dieser ist jedem Knoten, jeder ander Knoten, in der von ihm aus rechts liegenden Zelle zugeordnet.

Tabelle 1: Extrahierte Join-Keys für Join mit rechter Nachbarzelle

| Knoten | Zell-Id | Join-Key links | Join-Key rechts |
|---|---|---|---|
| K1 | {X:1,Y:1} | {X:2,Y:1} | {X:1,Y:1} |
| K2 | {X:3,Y:2} | {X:4,Y:2} | {X:3,Y:2} |
| K3 | {X:2,Y:1} | {X:3,Y:1} | {X:2,Y:1} |
| K4 | {X:0,Y:1} | {X:1,Y:1} | {X:0,Y:1} |
| K5 | {X:5,Y:5} | {X:6,Y:5} | {X:5,Y:5} |
| K6 | {X:1,Y:1} | {X:2,Y:1} | {X:1,Y:1} |
| K7 | {X:7,Y:1} | {X:8,Y:1} | {X:7,Y:1} |
| K8 | {X:2,Y:1} | {X:3,Y:1} | {X:2,Y:1} |

Nachdem durch die Joins Tupel sich potentiell abstoßender Knoten erzeugt wurden, müssen nun die Tupel herausgefiltert werden, deren Knoten mehr als 2k auseinander liegen (Filter Max Distance). Abschließend werden aus jedem solchen Tupel zwei Kräfte errechnet (eine für jeden Knoten im Tupel) und die errechneten Kräfte in einem DataSet zusammengeführt (Union).

Das Ergebnis der Kräfteberechnung sind zwei Datasets (R-Forces enthält die errechneten Abstoßungskräfte, A-Forces die Anziehungskräfte), deren Elemente aus je einem Kraftvektor (der Richtung und Stärke der errechneten Kraft angibt) und einer GradoopId (die angibt für welchen Knoten die Kraft errechnet wurde) bestehen. Diese beiden Datasets werden nun zu einem einzelnen Dataset zusammengeführt (Forces). Anschießend werden die Kräfte nach GradoopId gruppiert und aufsummiert. Dadurch entsteht ein Dataset (Sum-Forces), das für jeden Knoten exakt einen Vektor mit der Summe aller auf den Knoten einwirkenden Kräfte enthält.

Nun werden die errechneten Kräfte an die zugehörigen Knoten gejoint (LVertex, Force). Da jetzt jedem Knoten die auf in einwirkende Kraft zugeordnet ist, kann aus der aktuellen Position des Knotens und der Kraft die neue Position des Knotens errechnet werden.

Da für den Fruchterman-Reingold-Algorithmus kein Simulated Annealing Abkühl-Schedule vorgegeben ist, dieser aber für die Berechnung der neuen Knotenposition aus alter Position und einwirkender Kraft benötigt wird, musste für diese Implementierung selbst ein Schedule gewählt werden. Dieser lautet wie folgt:

$$schrittweite(iteration) = startSchrittweite * basis^{iteration} \tag{5}$$

*StartSchrittweite* ist dabei festgelegt als:

$$startSchrittweite = \frac{\sqrt{layoutBreite * layoutHöhe}}{2} \tag{6}$$

*Basis* wird mittels folgender Formel so festgelegt, dass die Schrittweite für die letzte Iteration 10% von k (siehe Kraftberechnung) beträgt.

$$basis = (\frac{endSchrittweite}{startSchrittweite})^{\frac{1}{maxIterationen-1}} \tag{7}$$

Dadurch, dass Start- und Endschrittweite festgelegt sind und sich mit der Anzahl der Iterationen lediglich ändert wie viele Zwischenschritte es gibt, ist sichergestellt, dass unabhängig von der Iterationsanzahl immer ein sinnvoller Wertebereich genutzt wird. Bei einem ähnlichen Schedule mit fester Basis und flexibler Endschrittweite würde sich die Schrittweite bei wenigen Iterationen nicht ausreichend schnell verkleinern und bei vielen Iterationen irgendwann so klein werden, dass kaum noch Veränderungen im Layout auftreten. Der hier gezeigte Schedule ist nicht optimal und kann wahrscheinlich noch verbessert werden. In den Versuchen lieferte er dennoch gute Ergebnisse mit relativ wenigen Iterationen.

Falls die vorgegebene Anzahl an Iterationen noch nicht erreicht wurde, wird die nächste Iteration begonnen und die Knoten mit den neuen Positionen als Eingabe für die Kräfteberechnung genutzt. Sobald alle nötigen Iterationen absolviert sind, werden die Knoten (die derzeit im LVertex Format vorliegen) mit den Original-Knoten (im EPGM-Format) gejoint. So kann die errechnete Position für jeden Knoten als Property im EPGM-Knoten eingetragen werden. Somit wurde jedem Knoten des Eingabe-Graphen eine Position zugewiesen und das Layouting ist beendet.

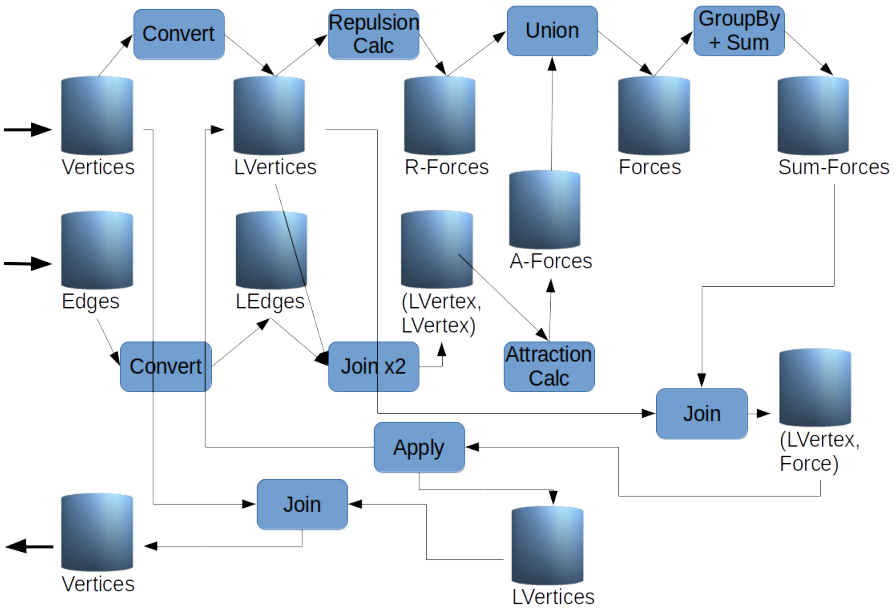


Abbildung 6: Datenfluss des FR-Layouter

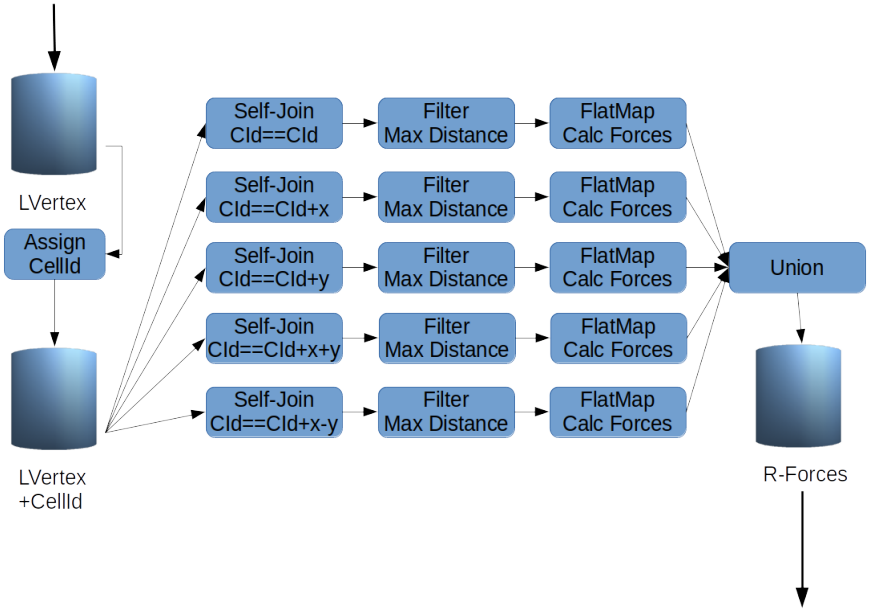


Abbildung 7: Abstoßungsberechnung in Flink

### 4.1.3 Centroid-Layouter

Derzeit ist es mit Flink nicht möglich, gleichzeitig über mehrere DataSets zu iterieren. Für die Implementierung, des ursprünglich in Spark entwickelten Zentroid basierten Layouting-Algorithmus (siehe Kapitel 3.2), ist es allerdings nötig, parallel über zwei DataSets (eins für die Knoten und eins für die aktuellen Zentroide) zu iterieren. Als Workaround wurde ein Marker-Interface (GraphElement) geschaffen, welches von den Klassen für die Zentroide und für die Knoten "implementiert" wird. Damit können Zentroide und Knoten zusammen in einem DataSet vom Typ GraphElement gespeichert werden. Am Beginn jeder Iteration wird dieses gemischte Dataset in je ein DataSet für Knoten und für Zentroide aufgeteilt und am Ende der Iteration werden die beiden DataSets wieder zusammengeführt. Das erzeugt ein wenig Overhead, der sich insgesamt aber nicht dramatisch auswirken sollte.

Der grundlegende Ablauf der Implementierung unterscheidet sich in den meisten Punkten nicht vom Fruchterman-Reingold-Layouter. Hinzu kommt lediglich die Berechnung der Cluster-Zentren. Dafür werden zu Beginn des Algorithmus einige Knoten zufällig ausgewählt. Dabei wird die Auswahlrate mit der folgenden Formel bestimmt:

$$auswahlRate = \frac{1}{(0.05 + 0.0025)/2} * anzahlKnotenImGraph \tag{8}$$

Damit werden statistisch so viele initiale Cluster-Zentren gewählt, dass jedes von ihnen im Schnitt 2.624% der Knoten des Graphen enthält und damit genau mittig im zuvor genannten erlaubten Bereich liegt.

Das Aktualisieren der Cluster-Zentren-Positionen erfolgt über einen Map-Task über alle Knoten, der als BroadcastSet das DataSet mit den aktuellen Zentroiden enthält. Die Map-Funktion iteriert für jeden Knoten über alle Zentren und gibt dann ein Tupel aus, welches die Id des Zentrums und die Position des Knotens beinhaltet. Diese Tupel werden anschließend nach Id gruppiert und für jede Gruppe der Durchschnitt der Positionen berechnet, welcher dann die neue Position des Zentrums ist. Im Rahmen dieser Berechnung wird auch für jedes Zentroid gespeichert, wie viele Knoten zu ihm gehören. Diese Information wird anschließend von einem FlatMap-Task über die Zentren verwendet um (wie in Kapitel 3.2 beschrieben) bei Bedarf Zentren aufzuteilen oder zu entfernen.

Die Berechnung der Abstoßungskräfte erfolgt mittels eines Map-Tasks über die Knoten. Dieser Map-Task erhält als BroadcastSet das DataSet mit den aktuellen Zentroiden. Die Map-Funktion errechnet damit nun die Summe der Abstoßungskräfte, die von den Zentroiden aus auf den aktuellen Knoten einwirken und gibt diese aus.

Die Berechnung der Anziehungskräfte und das Anwenden der Kräfte erfolgt wieder analog zum Fruchterman-Reingold-Layouter.

Da das DataSet mit den Cluster-Zentren maximal 400 Einträge enthält (durch die 0.25% Grenze), ist es unproblematisch, in den Map-Funktionen jeden Knoten mit jedem Zentroid zu vergleichen. Und da die Cluster-Zentren als BroadcastSet übergeben werden, fällt für all diese Vergleiche kaum Netzwerkverkehr an.

Interessant ist, dass Auber und Hinge behaupten, die Formeln des Fruchterman-Reingold-Algorithmus zu verwenden, aber die Formeln in ihrem Paper diesen nicht entsprechen [22]. Die in dieser Arbeit entwickelte Implementierung verwendet dagegen tatsächlich die Formeln des Fruchterman-Reingold-Algorithmus.

### 4.1.4 GiLa-Layouter

Für die Implementierung von GiLa (siehe Kapitel 3.3) in Flink wurde Flink's Think-like-a-Vertex Schnittstelle Gelly (siehe Kapitel 2.3) verwendet. Da Giraph und Gelly große Ähnlichkeit zueinander haben, können große Teile des Algorithmus (z.B. die Verbreitung von Positionen mittels Flooding) in Gelly genau so implementiert werden wie in Giraph. Es existieren allerdings durchaus einige Abweichungen zwischen der originalen GiLa-Implementierung und der Implementierung in Gelly.

Während Kanten in Giraph ungerichtet sind, sind Kanten in Gelly (zumindest in der Kombination mit Gradoop) gerichtet und Knoten können Nachrichten nur in Richtung der Kanten verschicken. Da der GiLa-Algorithmus von ungerichteten Graphen ausgeht, müssen für Gelly für alle Kanten des Eingabegraphen zusätzliche Kanten in Gegenrichtung eingefügt werden, wodurch die Größe des Datensatzes und damit der nötige Rechenaufwand steigt.

Um eine bessere Vergleichbarkeit mit den anderen Layout-Algorithmen zu erreichen, arbeitet die Gelly-Implementierung (im Gegensatz zum Original) nicht mit einer automatischen Abbruchbedingung, sondern läuft (wie die anderen Algorithmen auch) immer für eine vom Nutzer vorgegebene Anzahl an Iterationen. Da die genaue Anzahl der durchzuführenden Iterationen so von Anfang an bekannt ist, wird der Simulated Annealing Schedule des Fruchterman-Reingold-Layouters verwendet.

Da Gelly keine Möglichkeit bietet auf die Verteilung der Knoten über die Rechner-Knoten Einfluss zu nehmen, entfällt dieser Schritt bei der Gelly-Implementierung. Das wird sich voraussichtlich deutlich auf die Performance der Implementierung auswirken, da die Reduzierung des Kommunikationsaufwandes durch geschicktes Partitionieren der Knoten ein wesentlicher Aspekt bei GiLa ist.

Der Gesamtablauf ist nun wie folgt: Der Eingabegraph bekommt zuerst durch den Random-Layouter ein initiales Layout. Danach werden alle Knoten mit Grad 1 vorübergehend aus dem Graphen entfernt. Anschließend wird der gerichtete Eingabegraph in einen (temporären) ungerichteten Graphen konvertiert. Der letztendliche Ausgabegraph enthält diese zusätzlichen Kanten nicht mehr. Um die im Gradoop-Format vorliegenden Knoten und Kanten mittels Gelly verarbei-

ten zu können, müssen diese zuerst in ein Gelly-geeignetes Format konvertiert werden. Nun folgt das eigentlichen Layouting. Knoten versenden Nachrichten mit ihrer Position, ermitteln die auf sie einwirkenden Kräfte aus den empfangenen Nachrichten und wenden diese Kräfte auf ihre Position an. Nachdem die erforderliche Anzahl an Iterationen absolviert wurde, werden die im Gelly-Format vorliegenden Knoten über ihre Id mit den ursprünglichen Knoten gejoint und die errechnete Position wird in diesen als Property hinterlegt. Zum Schluss werden nun die anfangs entfernten Knoten wieder eingefügt und deren Position aus der Position ihres (einzigen) Nachbars errechnet.

## 4.2 Eigene Algorithmen

### 4.2.1 Sampling-Layouter

Der Sampling-Layouter entstand bei dem Versuch, den Fruchterman-Reingold-Layouter möglichst einfach zu optimieren. Der größte Teil der Rechenzeit beim Fruchterman-Reingold-Layouter entfällt auf das Berechnen der Abstoßungskräfte zwischen zahlreichen Knotenpaaren. Die Überlegung ist nun, dass nicht alle diese Abstoßungen wirklich nötig sind, um ein ausreichend gutes Layout zu erhalten. Bis auf die Abstoßungsberechnung ist der Sampling-Layouter identisch zum Fruchterman-Reingold-Layouter. Die Abstoßungskräfte werden nun aber nicht mehr zwischen allen Knoten unter 2k Abstand zueinander berechnet. Stattdessen werden Abstoßungen zwischen allen Knoten des Graphen und einer (in jeder Iteration neu) zufällig gewählten Teilmenge der Knoten in 2k Abstand berechnet. Zusätzlich werden die errechneten Abstoßungskräfte mit dem Kehrwert der Samplingrate für die Teilmenge multipliziert. Damit wird die Abstoßungskraft der Konten berücksichtigt, die nicht Teil der Teilmenge geworden sind. Ist beispielsweise die Samplingrate 0.1, so wird etwa jeder zehnte Knoten in die Teilmenge übernommen. Damit muss die Abstoßungskraft 10 mal so groß sein wie eigentlich errechnet, da ein Knoten nun stellvertretend für insgesamt 10 Knoten steht.

Je kleiner diese Teilmenge (die Samplingrate) ist, desto weniger rechenaufwändig ist die Berechnung. Allerdings führen kleine Teilmengen auch zu einer zunehmend schlechter werdenden Layoutqualität, da die Teilmenge den Originalgraphen nur noch unzureichend genau repräsentiert. Es gilt also eine Abwägung zwischen Qualität und Laufzeit zu treffen.

Problematisch ist, dass bei dieser Art der Abstoßungsberechnung die im Fruchterman-Reingold-Layouter auftretenden Symmetrien bei den Zell-Nachbar-Joins nicht mehr auftreten. Dadurch müssen anstatt nur 5, die vollen 9 Joins durchgeführt werden (Siehe Kapitel 4.1.2), was zu einer verschlechterten Performance führt.

### 4.2.2 VertexFusing-Layouter

Die grundlegende Idee bei fast jedem Versuch, einen Force-directed Layouting Algorithmus zu beschleunigen, ist es, die Anzahl der zu berechnenden Kräfte zu reduzieren. Oft wird dies versucht, indem Kräfte nur zwischen wenigen ausgewählten Knoten berechnet werden. Eine andere Möglichkeit besteht darin, die Größe und Komplexität des ursprünglichen Graphen zu reduzieren und das Layout auf einer vereinfachten Version des Graphen durchzuführen. So gehen beispielsweise Multilevelalgorithmen wie Multi-GiLa [23] vor. Die Schwierigkeit dabei besteht darin, eine Methode zu finden, mit der sich der Graph möglichst stark und ohne großen Rechenaufwand vereinfachen lässt, ohne dabei die zugrundeliegende Struktur des Graphen zu zerstören. Dieses Kapitel beschreibt eine Idee für einen solchen Vereinfachungsalgorithmus, seine Anwendung im Layouting und die Umsetzung in Flink.

#### 4.2.2.1 Zusammenfassen ähnlicher Knoten

Die zugrundeliegende Überlegung ist, dass zwei Knoten ähnlich zueinander sind, wenn sie nahe beieinander liegen und auf sie ähnliche Kräfte einwirken. Denn wenn dies der Fall ist, dann werden sie in der nächsten Iteration wieder nahe beieinander liegen und wieder ähnliche Kräfte erfahren. Darüber hinaus üben nahe beieinander liegende Knoten ähnliche Kräfte auf ihre Umgebung aus. Da sich die beiden Knoten im zukünftigen Verlauf des Layoutings immer nah beieinander befinden, ist die Überlegung naheliegend, beide Knoten als einen einzelnen Knoten zu betrachten, der sich in der Mitte zwischen beiden Knoten befindet. Dieser Superknoten übt die gleichen Kräfte auf seine Umgebung aus, wie die zwei Knoten die durch ihn ersetzt werden. Dafür erbt er die Kanten der beiden ursprünglichen Knoten, wodurch er ihre Anziehungskräfte übernimmt. Der Superknoten befindet sich ungefähr an der selben Stelle wie die zwei ursprünglichen Knoten und übt damit ähnliche Abstoßungskräfte auf seine Umgebung aus wie diese. Allerdings steht der Superknoten nun stellvertretend für zwei Knoten, weshalb seine Abstoßungskraft verdoppelt werden muss. Die Formel für die Abstoßungskräfte zwischen zwei Knoten muss nun also berücksichtigen, wie viele Unterknoten ein Superknoten repräsentiert. Diese Anzahl wird als Masse des Superknotens bezeichnet (ein "normaler" Knoten hat eine Masse von 1). Die neue Abstoßungsformel lautet dadurch:

$$f_a(v1, v2) = \frac{-k^2}{d(v1, v2)} * masse(v1) * masse(v2) \tag{9}$$

Die Masse der beiden beteiligten Knoten hat einen ähnlichen Einfluss wie die Masse in der Physik Einfluss auf die zwischen Körpern wirkende Gravitation hat. Eine Verdopplung der Masse eines Körpers führt zur Verdopplung der Gravitationskraft zwischen den Körpern. Eine Verdopplung der Masse beider Körper führt zu einer Vervierfachung der Kraft usw. [28].

Ein Superknoten aus zwei Knoten übt eine doppelt so starke Abstoßungskraft aus wie jeder seiner Einzelknoten es tun würde. Da die Abstoßungskräfte aber symmetrisch wirken, erfährt er auch eine doppelt so große Abstoßungskraft von jedem anderen Knoten. Damit erfährt er also doppelt so starke Abstoßungskräfte wie jeder seiner Unterknoten erfahren würde, was bei normaler Anwendung der Kraft dazu führen würde, dass der Superknoten sich doppelt so weit bewegen würde wie jeder seiner Unterknoten. Gleiches gilt für die Anziehungskräfte. Der Superknoten erbt alle Kanten der Unterknoten und erfährt dadurch die gesammelten Anziehungskräfte beider Unterknoten. Da der Superknoten aber ja eigentlich das Verhalten der Unterknoten widerspiegeln soll, darf das nicht passieren. Bisher wurde bei der Kraftanwendung die Massenträgheit nicht beachtet. (Dies war auch nicht nötig, da beim normalen Fruchterman-Reingold-Algorithmus alle Knoten dieselbe Masse aufweisen.) Um einen Körper mit der doppelten Masse gleich stark zu beschleunigen, benötigt es doppelt so viel Kraft. Auch hier ist die Analogie zur Gravitation bemerkenswert. Wenn ein Gegenstand auf die Erde fällt, ziehen sich der Gegenstand und die Erde zwar gegenseitig gleich stark an (auf sie wirkt die gleiche Kraft ein), aber da die Erde erheblich schwerer ist, wird die Erde erheblich weniger stark beschleunigt als der Gegenstand [28]. Im Falle dieses Algorithmus muss also bei der Berechnung der neuen Position eines Superknotens die auf ihn einwirkende Kraft durch die Masse des Superknotens dividiert werden. Dadurch befindet sich der Superknoten in der nächsten Iteration an der Position, an der sich auch seine Einzelknoten befinden würden. Die Berechnung der neuen Position aus vorheriger Position und einwirkenden Kräften geschieht nun also wie folgt:

```
beschränkt(x) = ( |x| < schrittWeite ) ? x : x / |x| * schrittWeite
kräfteNeu(v1,t) = kräfte(v1,t) / masse(v1)
position(v1,t+1) = position(v1,t) + beschränkt(kräfteNeu(v1,t))
```

Da im Zuge der Zusammenfassung von Knoten zu Superknoten auch Kanten zu Superkanten zusammengefasst werden (siehe Kapitel 4.2.2.3), muss auch die Berechnung der Anziehungskräfte minimal angepasst werden. Eine Superkante die n Kanten zusammenfasst, muss auch die n-fache Anziehungskraft ausüben wie eine einfache Kante.

Dieses Zusammenfassen von ähnlichen Knoten kann nun iterativ durchgeführt werden, so dass bestehende Superknoten weitere Knoten aufnehmen oder mit anderen Superknoten zusammengefasst werden. (Beim Zusammenfassen zweier Superknoten addieren sich deren Massen.) Nach ausreichend vielen Schritten sind alle Knoten mit ausreichender Ähnlichkeit zusammengefasst und die verbleibenden (Super-)Knoten sind untereinander nicht ähnlich genug, um ihrerseits zusammengefasst zu werden. Der resultierende Graph ist nun eine Vereinfachung des ursprünglichen Graphen, der dennoch dessen ursprüngliche Struktur widerspiegelt.

Zu beachten ist dabei noch, dass nur Knoten, die durch eine Kante verbunden sind, für die Zusammenfassung in Frage kommen. Das reduziert die Laufzeit erheblich, da nicht mehr $|V|^2$ Vergleiche zwischen Knoten durchgeführt werden

müssen, sondern nur noch |E|. Da sich verbundene Knoten durch die zwischen ihnen herrschende Anziehungskraft einander annähern, befinden sie sich zumeist nahe beieinander und kommen so als ähnliche Knoten in Frage. Knoten die nicht direkt verbunden sind, befinden sich meist zu weit von einander entfernt, um für eine Zusammenfassung überhaupt in Frage zu kommen. Aufgrund dieser Tatsachen und der Tatsache, dass die Zusammenfassung iterativ erfolgt und so auch Knoten zusammengefasst werden können, die ursprünglich nicht durch eine Kante verbunden waren, ist die Beschränkung auf Vergleiche zwischen verbundenen Knoten weitgehend unproblematisch.

#### 4.2.2.2 Ähnlichkeitsfunktion

Wie bereits erklärt, sind zwei Knoten ähnlich zueinander, wenn sie nahe beieinander liegen und auf sie ähnliche Kräfte einwirken. Allerdings braucht es nun noch ein Ähnlichkeitsmaß für Kräfte und Positionen. Da Kräfte als Vektoren repräsentiert werden, bietet sich auf den ersten Blick das Kosinus-Maß als Ähnlichkeitsfunktion für die Kräfte an. Da dieses aber lediglich die Richtungen zweier Vektoren vergleicht und deren Betrag ignoriert, kommt es hier nicht in Frage. Stattdessen lautet das verwendete Ähnlichkeitsmaß für Kräfte:

$$sim(f1, f2) = 1 - \frac{|f1 - f2|}{|f1| + |f2|} \tag{10}$$

Mit diesem Ähnlichkeitsmaß ergibt sich eine Ähnlichkeit von 0 für direkt entgegen gesetzte Kräfte, von 1 für identische Kräfte und einem Wert dazwischen für alles andere. Anzumerken ist dabei, dass Kräfte welche die gleiche Richtung, aber unterschiedliche Beträge haben, keine Ähnlichkeit von 1 erzielen (im Gegensatz zum Kosinus-Maß).

Die Ähnlichkeit zwischen zwei Positionen wird am Verhältnis vom Abstand zu k gemessen. k ist dabei der Abstand, den zwei über eine Kante verbundene Knoten haben (da sich in einer Entfernung von k Anziehungs- und Abstoßungskräfte zwischen diesen Knoten aufheben). Dieses Verhältnis wird anschließend auf Werte zwischen 0 und 1 begrenzt.

$$sim(pos1, pos2) = min(1, max(0, 1 - \frac{d(pos1, pos2) - k}{k})) \tag{11}$$

Das führt dazu, dass Positionen mit einem Abstand von k oder weniger eine Ähnlichkeit von 1 bekommen und ab 2k Abstand eine Ähnlichkeit von 0.

Die Gesamtähnlichkeit ist nun das Produkt der Positions- und Kräfteähnlichkeit. Zwei Knoten gelten als ähnlich (und somit als zusammenfassbar), wenn ihre Gesamtähnlichkeit einen vom Nutzer vorgegebenen Schwellwert überschreitet. Durch die Wahl dieses Schwellwertes lässt sich variieren, wie stark der Graph zusammengefasst werden soll. Hohe Schwellwerte führen zu einer geringen und niedrige Schwellwerte zu einer aggressiven Zusammenfassung.

Das hier vorgestellte Ähnlichkeitsmaß ist nicht das einzig denkbare. Speziell das Ähnlichkeitsmaß für Positionen oder die Gewichtung der Einzelähnlichkeiten ist durchaus diskutabel. Daher bietet die Implementierung des Algorithmus dem Nutzer die Möglichkeit, eine eigene Ähnlichkeitsfunktion zu benutzen.

#### 4.2.2.3 Gruppieren via Ähnlichkeitsfunktion

Gradoop bietet zwar die Möglichkeit, die Knoten von Graphen zu gruppieren um vereinfachte Graphen zu generieren [16], allerdings ist es dabei nur möglich, Knoten nach Eigenschaften (zum Beispiel dem Wert einer EPGM-Property) zu gruppieren. Das Gruppieren über eine Ähnlichkeitsfunktion hingegen ist derzeit in Gradoop nicht möglich und noch dazu komplexer als das Gruppieren nach Eigenschaften. Dafür muss vor dem eigentlichen Gruppierungsschritt zuerst bestimmt werden, welche Knoten zu einer gemeinsamen Gruppe gehören und daher am Ende einen Superknoten bilden sollen.

Die grundlegende Idee ist es, jeden Knoten mit dem Knoten zusammen zu fassen, der ihm laut Ähnlichkeitsfunktion am ähnlichsten ist. Bei der Umsetzung dieser Idee treten allerdings diverse Probleme auf. Eine Ursache für diese Probleme ist die Tatsache, dass jeder Knoten nur einem einzelnen Superknoten zugeordnet werden kann. Eine weitere Ursache ist, dass die Eigenschaft “am Ähnlichsten zu” nicht unbedingt symmetrisch ist.

Beispiel (siehe Abbildung 8): Es werden 3 Knoten betrachtet a,b und c. Diese bekommen als Werte die Zahlen a=1, b=3 und c=4 zugeordnet. Die Ähnlichkeit zwischen den Knoten wird definiert als Kehrwert des Betrags der Differenz ihrer Werte. Es gilt also: $sim(a, b) = 1/(|1 - 3|) = 0.5$, $sim(b, c) = 1$, $sim(a, c) = 0.33$. Damit ist der ähnlichste Knoten zu a der Knoten b. Aber der ähnlichste Knoten zu b ist Knoten c. Als Schwellwert ab dem zwei Knoten ähnlich genug sind, um zusammen gefasst werden zu können, wird nun 0.45 festgelegt. Damit müsste Knoten a mit Knoten b (dem ihm am ähnlichsten Knoten) zusammengefasst werden und Knoten b mit Knoten c. Wird Knoten b nun aber mit Knoten c zusammengefasst, steht b nicht mehr zur Verfügung, um mit Knoten a zusammen gefasst werden zu können. Und da der Superknoten bc einen Wert von 3.5 ((3 + 4)/2) erhalten würde und damit die Ähnlichkeit $sim(a, bc) = 0.4$ zu gering wäre, um a und bc nachträglich zusammen zu fassen, würde Knoten a ohne Partner verbleiben. Würde man allerdings zuerst Knoten a und Knoten b zusammenfassen hätte der Knoten ab einen Wert von 2 und damit wäre $sim(ab, c) = 0.5$ und somit groß genug um eine Zusammenfassung von ab und c zu erlauben. Letztendlich wären also alle Knoten a,b und c zu einem Superknoten zusammengefasst.

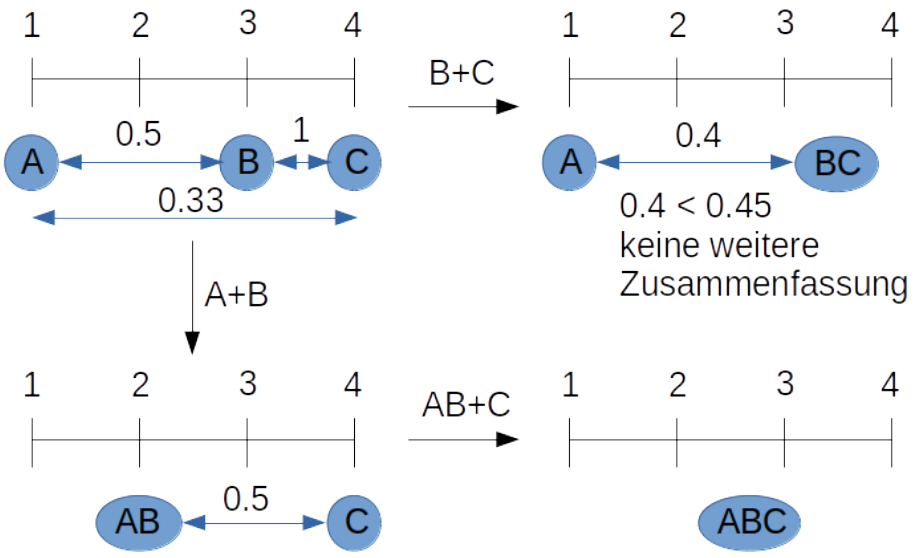


Abbildung 8: Beispiel: Gruppieren von Knoten

Wie im Beispiel gezeigt, hat die Reihenfolge, in der Knoten aufgrund ihrer Ähnlichkeit zusammen gefasst werden, erheblichen Einfluss auf das Endergebnis. Das Finden einer optimalen Zusammenfassungsreihenfolge, also einer Reihenfolge, bei der die Knoten bestmöglich zusammen gefasst werden, ist ein äußerst komplexes Optimierungsproblem, dessen Lösung weit mehr Rechenaufwand benötigen würde, als dadurch beim Layouting eingespart wird. Glücklicherweise ist eine optimale Lösung hier nicht erforderlich, weshalb stattdessen die folgende Näherungslösung verwendet wird.

Zuerst werden die Knoten des Graphen zufällig in zwei etwa gleich große Kategorien aufgeteilt. Die eine Kategorie wird Spenderknoten und die Andere Empfängerknoten genannt. Nun gilt die Regel, dass Zusammenfassungen nur zwischen einem Spenderknoten und einem Empfängerknoten möglich sind, nicht aber zwischen zwei Knoten der gleichen Kategorie. Es wird nun also für jeden Spenderknoten derjenige Empfängerknoten gesucht, zu dem er die größte Ähnlichkeit hat und der Spenderknoten wird dem Empfängerknoten zugeordnet, zumindest sofern die errechnete Ähnlichkeit den gewählten Schwellwert übersteigt. Einem Empfängerknoten können dabei beliebig viele Spenderknoten zugeordnet werden. Da die Empfängerknoten selbst keinem anderen Knoten zugeordnet werden, stehen sie für die gesamte Dauer der Berechnung unverändert für Zuordnungen zur Verfügung und die Reihenfolge der Zuordnungen spielt keine Rolle mehr. Nachdem alle Zuordnungen erfolgt sind (wobei es durchaus auch Spenderknoten geben kann, die aufgrund des gesetzten Ähnlichkeitsschwellwertes keinen

geeigneten Zuordnungspartner gefunden haben), bildet jeder Empfängerknoten mit den ihm zugeordneten Spenderknoten einen Superknoten, dessen Wert dem Durchschnitt aller beteiligten Knoten entspricht.

Dieser Ansatz hat natürlich einen offensichtlichen Mangel. Wenn zwei Knoten der selben Kategorie (Spenderknoten/Empfängerknoten) zugeteilt wurden, können diese nicht gemeinsam einen Superknoten bilden, ganz egal, wie ähnlich sie zueinander sind. Um diese Beschränkung zu überwinden, muss der beschriebene Prozess mehrfach hintereinander ausgeführt werden, wobei bei jeder Durchführung die Zuordnung zu den Kategorien neu vorgenommen wird. Wenn dadurch die Knoten nicht mehr der selben Gruppe angehören und ihre Ähnlichkeit immer noch hoch genug ist können diese nun gemeinsam einen Superknoten bilden. Zu beachten ist dabei jedoch, dass die Knoten zu dem Zeitpunkt bereits in einem (oder zwei verschiedenen) Superknoten enthalten sein können. Das ist aber nicht weiter problematisch, da nun einfach die bestehenden Superknoten zu einem neuen Superknoten kombiniert werden (sofern sie der Anforderung an die Mindestähnlichkeit entsprechen). So werden Stück für Stück die Knoten und Superknoten immer weiter zusammengefasst, bis die verbleibenden (Super-)Knoten so unähnlich zueinander werden, dass keine weitere Zusammenfassung möglich ist. Damit ist das Gruppieren beendet.

#### 4.2.2.4 Umsetzung der Gruppierung in Flink

Mit dem vorgestellten Verfahren ist nun eine vergleichsweise performante Implementierung der Gruppierungsoperation in Flink möglich. Da die Implementierung (wie bereits mehrfach erwähnt) eigene Datentypen für die Graph-Elemente verwendet, aber der in Gradoop vorhandene Gruppierungsoperator nur mit den Gradoop-eigenen Datentypen arbeiten kann, wäre für die Benutzung dieses Operators die wiederholte Umwandlung zwischen beiden Datenformaten nötig, was unnötigen Overhead verursachen würde. Daher wird stattdessen eine eigene, auf die internen Datentypen ausgelegte Gruppierungsfunktion implementiert.

Abbildung 9 zeigt schematisch wie die Implementierung in Flink umgesetzt ist. Im ersten Schritt wird jeder Knoten mit einer Wahrscheinlichkeit von 50% entweder den Spenderknoten oder den Empfängerknoten zugeordnet (Assign Randomly). Da wie in Kapitel 4.2.2.1 beschrieben nur die durch eine Kante verbundenen Knoten auf Ähnlichkeit überprüft werden müssen, werden als nächstes die Kanten des Graphen mit ihren Start- und Zielknoten gejoint, um so Paare aus miteinander verbundenen Knoten zu erhalten (LVertex,LVertex). Nun werden von diesen Paaren all diejenigen verworfen, bei denen beide Knoten Spenderknoten oder beide Knoten Empfängerknoten sind, da diese, wie bereits erwähnt, für eine Verschmelzung von vorneherein nicht in Frage kommen. Für jedes der verbleibenden Paare wird nun die Ähnlichkeit der beiden Knoten anhand der gegebenen Ähnlichkeitsfunktion berechnet und als Ergebnis wird ein Tupel vom Format-(Spenderknoten, Empfängerknoten, Ähnlichkeit) ausgegeben. Tupel, bei denen die errechnete Ähnlichkeit unter dem festgesetzten Schwellwert

liegt, werden an dieser Stelle direkt verworfen. Da jeder Spenderknoten nur dem Empfängerknoten mit der größten Ähnlichkeit zugeordnet werden soll, werden diese Tupel nun nach Spenderknoten gruppiert (Group By Donor Id) und die entstehenden Gruppen so reduziert, dass in jeder Gruppe nur das Tupel mit dem höchsten Ähnlichkeitswert übrig bleibt. Das Ergebnis ist ein DataSet (LVertex,LVertex), das für jeden zu verschmelzenden Spenderknoten ein Tupel mit dem passenden Empfängerknoten enthält. Dieses DataSet wird im Folgenden als Verschmelzungskandidaten bezeichnet.

Das DataSet der Verschmelzungskandidaten wird nun nach Empfängerknoten gruppiert (Group By Acceptor). So enthält jede Gruppe alle Knoten, die zusammen mit einem bestimmten Empfängerknoten einen Superknoten bilden sollen. Die Knoten jeder Gruppe werden nun zu einem einzelnen Knoten reduziert (Reduce By Avg). Die so entstehenden Knoten sind die Superknoten. Jeder Superknoten enthält eine Liste der IDs aller Knoten, die Teil des Superknotens geworden sind. Die Anzahl der enthaltenen Knoten wird als Masse des Superknotens bezeichnet. Dabei muss beachtet werden, dass diese Knoten bereits selbst Superknoten sein könnten. In diesem Fall übernimmt der neue Superknoten alle Unterknoten aller vorherigen Superknoten. Der Wert des neuen Superknotens (in diesem Fall die Position im Layout) ist der nach den Massen der Teilknoten gewichtete Durchschnitt der Werte aller Teilknoten. Der neue Superknoten erbt die ID des Empfängerknotens, aus dem er erzeugt wurde. Bei der Erzeugung des Superknotens darf nicht vergessen werden, dass der Empfängerknoten genauso ein Unterknoten des neuen Superknotens, ist wie die Spenderknoten und ebenso in der Berechnung des neuen Wertes berücksichtigt werden muss.

Es gibt mit großer Wahrscheinlichkeit immer einige Spenderknoten die keinen geeigneten Empfänger gefunden haben, so, wie es wahrscheinlich auch Empfängerknoten gibt, denen kein einziger Spenderknoten zugeordnet wurde. Diese Knoten sind dadurch nicht Teil eines der im vorherigen Schritt erzeugten Superknoten, sollten aber dennoch Teil des vereinfachten Graphen sein. Daher müssen diese aus den ursprünglichen Knoten ermittelt werden. In Flink ist es möglich, alle Elemente eines DataSets zu finden, die nicht Teil eines anderen DataSets sind, indem man einen LeftOuterJoin zwischen beiden DataSets ausführt und alle Elemente verwirft, die einen Join-Partner gefunden haben. Übrig bleiben dann nur die Elemente des linken DataSets (z.B. die ursprünglichen Knoten), die keinen Partner mit der selben Id im rechten DataSet (z.B. die Empfängerknoten in den Verschmelzungskandidaten) haben. Führt man diesen Join nun zwischen den ursprünglichen Knoten und den Spenderknoten im DataSet der Verschmelzungskandidaten aus und führt anschließend noch einen weiteren Join mit dem Ergebnis und den Empfängerknoten, denen mindestens ein Spenderknoten zugewiesen wurde, durch, dann erhält man alle Knoten, die nicht Teil eines Superknotens sind. Diese Knoten bilden jetzt gemeinsam mit den Superknoten die Knoten des vereinfachten Graphen.

Nachdem die Knoten des vereinfachten Graphen bestimmt wurden, fehlen nun noch die Kanten. Da diverse Knoten zusammengefasst wurden (und damit nicht

mehr unter ihrer ursprünglichen Id auffindbar sind), referenzieren viele Kanten nun nicht mehr existierende Knoten als ihre Start- und Zielknoten. Daher müssen die Knotenreferenzen in den Kanten angepasst werden. Dafür werden die Kanten mit dem DataSet der Verschmelzungskandidaten gejoint (Join Fix IDs). Dieses enthält für jeden verschwundenen Knoten die ID des Empfängerknotens, mit dem dieser verschmolzen ist (und damit die Id des neuen Superknotens). Durch den Join der Kanten mit den Verschmelzungskandidaten über Start-/ZielknotenId der Kante und Id des Spenderknotens kann die in der Kante gespeicherte Id des Spenderknotens durch die Id des Superknotens ersetzt werden. Damit zeigt jede Kante wieder auf einen existierenden Knoten.

Durch die Verschmelzung von Knoten und das Anpassen der Kanten kann es nun vorkommen, dass mehrere Kanten mit dem selben Start- und Zielpunkt existieren. Diese doppelten Kanten werden nun zu jeweils einer Superkante kombiniert. Jede Superkante enthält die Ids aller Kanten, die sie ersetzt. Die Anzahl der ersetzten Kanten wird (analog zu den Superknoten) als Masse der Kante bezeichnet und später für die Kraftberechnung beim Layouting benötigt.

Damit ist ein Durchlauf der Knotengruppierung abgeschlossen und die ermittelten Superkanten, einfachen Kanten, Superknoten und nicht-zugeordneten Knoten bilden nun den vereinfachten Graphen. Durch wiederholte Anwendung der Knotengruppierung kann der Graph nun immer weiter vereinfacht werden, bis es irgendwann keine Knoten mehr gibt, die ähnlich genug sind, um zusammengefasst zu werden.

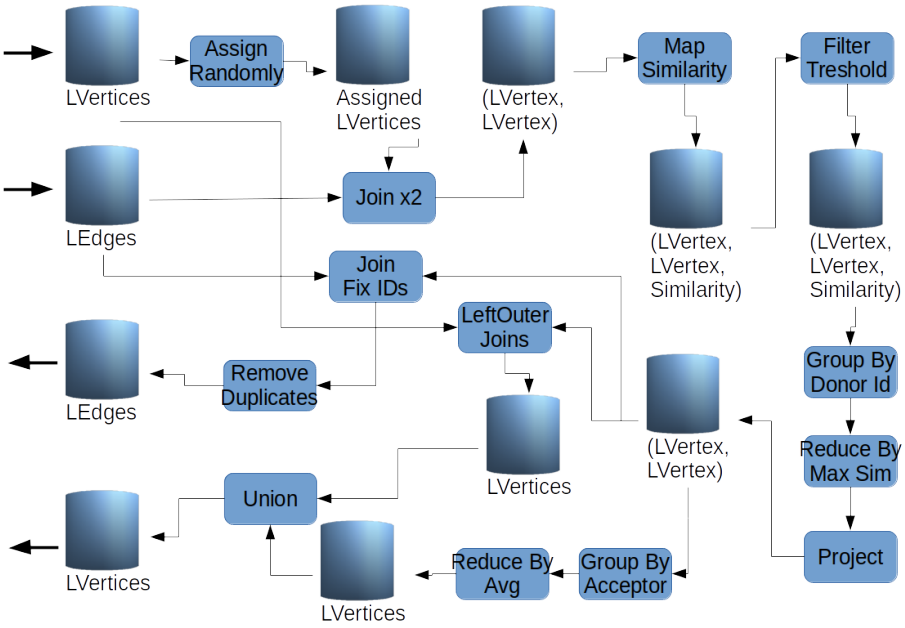


Abbildung 9: Schematischer Ablauf der Gruppierung

#### 4.2.2.5 Layouting-Algorithmus

Im Optimalfall würde die Vereinfachung des Graphen durch Gruppierung von Knoten vollständig vor Beginn des eigentlichen Layoutings durchgeführt werden, da die Reduzierung der Knoten im Graphen das Layouting deutlich beschleunigt. Unpraktischerweise zeigen sich die Ähnlichkeiten zwischen den Knoten, die ja über deren Positionen und erfahrenen Kräfte berechnet werden, erst mit Fortschreiten des Layouting-Prozesses. Während am Anfang des Layoutings alle Knoten noch zufällig angeordnet sind, gewinnt der Graph durch das Layouting an Struktur und zusammengehörige Knoten wandern stückweise dichter zueinander, bis am Ende alle zusammengehörenden Knoten nah beieinander und weit weg von fremden Knoten liegen. Um eine optimale Gruppierung der Knoten zu erreichen, müsste das Gruppieren also nach Abschluss des Layoutings durchgeführt werden. Da aber die ursprüngliche Motivation hinter dem Gruppieren der Knoten war, die Anzahl der Knoten zu reduzieren und so das Layouting zu beschleunigen, wäre es völlig nutzlos das Gruppieren erst nach Abschluss des Layoutings durchzuführen.

Die Lösung dieses Problems besteht darin, Layouting und Gruppierung gleichzeitig auszuführen. Nach jeder Layouting-Iteration wird eine Iteration des Gruppierungsalgorithmus durchgeführt. Dabei werden dann alle Knoten zu Superknoten verschmolzen, deren Bewegungen in der letzten Layouting-Iteration dazu geführt haben, dass eine Verschmelzung nun gerechtfertigt ist. So startet das Layouting zwar mit allen Knoten, jedoch reduziert sich nach jeder Layouting-Iteration die

Anzahl der Knoten, die in der nächsten Iteration noch weiter verarbeitet werden müssen. Das Layouting beschleunigt sich mit jeder absolvierten Iteration. Glücklicherweise ist das Fehlen an Beschleunigung in den ersten Iterationen aufgrund des folgenden Effektes nicht weiter problematisch: Zu Beginn des Layoutings sind alle Knoten zufällig und gleichmäßig über den kompletten Layoutbereich verteilt, wodurch sich nur wenige Knoten zueinander in einem Abstand von weniger als 2k befinden. Da nur solche Knoten vom Fruchterman-Reingold-Algorithmus für die Abstoßungsberechnung herangezogen werden, ist die Abstoßungsberechnung zu dem Zeitpunkt nicht sonderlich aufwändig. Mit fortschreiten des Layoutings wandern die Knoten allerdings, aufgrund der zwischen ihnen wirkenden Anziehungskräfte, dichter zueinander, wodurch sich mehr Knoten in weniger als 2k Abstand zueinander befinden und der Rechenaufwand für die Berechnung der Abstoßungskräfte steigt. Das Gruppieren von Knoten im Laufe des Layouting könnte also im Prinzip dazu führen, dass der Algorithmus genau dort beschleunigt wird, wo er normalerweise langsamer werden würde.

Die entscheidende Frage ist nun, ob und wie sehr der Geschwindigkeitsgewinn durch die Vereinfachung der Kräfteberechnung den Aufwand für die Gruppierung der Knoten übersteigt.

Abgesehen von der zusätzlichen Gruppierung verläuft das Layouting exakt wie beim normalen Fruchterman-Reingold-Layouter. Nachdem das Layout für den vereinfachten Graphen fertiggestellt ist, gibt es drei verschiedene Möglichkeiten, wie genau das endgültige Ergebnis des Layoutings aussehen soll.

1. Der vereinfachte Graph wird ausgegeben. Dies ist die einfachste Möglichkeit, da keine weiteren Schritte erforderlich sind. Je nach Anwendungsfall könnte es aber problematisch sein, dass der Ausgabegraph nicht mehr aus den selben (und gleich vielen) Knoten und Kanten besteht wie der Eingabegraph. Dies kann aber durchaus auch einen Vorteil darstellen. Zeichnungen von Graphen mit mehreren Zehntausend Knoten sind oft sehr unübersichtlich und unnötig kompliziert, ohne dabei wirklich mehr Informationen zu liefern. Der Erkenntnisgewinn aus dem Bild des vereinfachten Graphen kann durchaus größer sein als der aus dem Bild des vollständigen Graphen, speziell dann, wenn beim Zeichnen des Graphen die Eigenheiten des vereinfachten Graphen betrachtet werden. So sollte beispielsweise die Größe von Superknoten und Superkanten in der Zeichnung widerspiegeln, wie viele Unterknoten und Unterkanten sie in sich vereinen.

2. Der Ausgabegraph besteht aus den Knoten und Kanten des ursprünglichen Eingabegraphen und alle Knoten bekommen die Position des Superknotens, zu dem sie im vereinfachten Graphen gehören, zugeordnet. Dabei kann die zugeordnete Position auch ein wenig zufällig variiert werden, um zu verhindern, dass sich zu viele Knoten gegenseitig verdecken. Wenn mit dem erzeugten Layout noch weiter gearbeitet werden soll (z.B. im Rahmen einer Clusteranalyse), bietet sich diese Lösung an, allerdings sinkt damit die Aussagekraft erzeugter Bilder.

3. Es wird vorgegangen wie bei Variante 2, aber anschließend werden noch einige Layouting-Iterationen auf dem de-gruppierten Graphen ausgeführt. In der Theorie würde dadurch das beim Degruppieren entstandene Layout verbessert werden. Da diese Iterationen aber auf dem vollständigen Graphen ausgeführt werden würden, wären sie entsprechend rechenaufwändig. Außerdem müssen dabei zusätzliche Details beachtet werden, wie die Tatsache, dass der Simulated Annealing Schedule für diese Iterationen angepasst werden muss (nicht wieder am Anfang beginnen darf), da diese zusätzlichen Iterationen sonst das ursprünglich errechnete Layout wieder zerstören würden.

Welche der vorgeschlagenen Herangehensweisen die Beste ist hängt vom spezifischen Anwendungsfall ab. Daher sollte die Auswahl dem Nutzer überlassen werden.

### 4.2.3 CombiLayouter

Im Rahmen der Evaluierung des Centroid-Layouters (Kapitel 5.3.4) fielen einige Eigenschaften besonders auf. Zum einen erreicht der Centroid-Layouter prinzipbedingt auch bei großen Graphen eine gute Laufzeit und Skalierbarkeit. Zum anderen sind die damit errechneten Layouts nach den Qualitätsmetriken relativ gut und geben die globale Struktur des Graphen erkennbar wieder. Da die Abstoßungskraftberechnung aber nicht zwischen allen nah beieinander liegenden Knoten berechnet wird, kommt es zu Überlappungen von Knoten und Kanten, speziell in den Bereichen, in denen sich kein Zentroid befindet. Durch diese, vor allem in den Außenbereichen des Graphen auftretenden, Überlappungen sind feine Details, wie etwa einzelne Verzweigungen oder kleine Cluster, im Layout nicht mehr erkennbar. Das fällt besonders bei Graphen mit vielen kleinen Zusammenhangskomponenten auf. Alle Zusammenhangskomponenten, die zu klein sind, um ein eigenes Zentroid zugeordnet zu bekommen. kollabieren in einen einzelnen Punkt und da sie aufeinander ebenfalls keine Abstoßungskräfte ausüben, werden sie alle vollständig an die Ränder des Layouting-Bereiches gedrängt. Das so errechnete Layout ist für diese Graphen beinahe vollständig unbrauchbar (siehe Abbildung 22).

Beim Fruchterman-Reingold-Layouter hingegen stoßen sich alle nah beieinander liegenden Knoten gegenseitig ab, weshalb es kaum zu Überlappungen zwischen Knoten kommt und im resultierenden Layout auch kleine Details aus wenigen oder gar einzelnen Knoten erkennbar sind. Allerdings macht genau diese Detailliertheit den Algorithmus in den Anfangsphasen des Layoutings, in denen eigentlich nur grobe Veränderungen am Layout durchgeführt werden, unnötig langsam.

Zusammenfassend lässt sich also sagen:

- Der Centroid-Layouter errechnet schnell ein grobes Layout, dem es aber an Details fehlt.

- Der Fruchterman-Reingold-Layouter errechnet sehr detaillierte Layouts, hat aber in der Anfangsphase (bei der es noch nicht auf Präzision ankommt) Performance-Probleme.

Die naheliegende Lösung ist es nun, beide Algorithmen so zu kombinieren, dass sie gegenseitig ihre Schwächen ausgleichen. Um dies zu erreichen werden erst einige Iterationen des Centroid-Layouters durchgeführt. Das Ergebnis ist ein grobes Layout, bei dem durch Überlappungen viele Details verloren gehen. Auf dieses Zwischenlayout wird nun der Fruchterman-Reingold-Layouter angewendet. Da sich nun alle benachbarten Knoten gegenseitig abstoßen, werden sich vorher überlappende Knoten auseinander bewegen, wodurch die von ihnen gebildeten Details sichtbar werden. Da das Zwischenlayout bereits die grobe Struktur des Graphen widerspiegelt, sind die einzelnen Regionen des Graphen bereits gut über den Layoutbereich verteilt. Diese Verteilung sorgt dafür, dass weniger Knoten in weniger als 2k Abstand zueinander liegen, was die Performance des Fruchterman-Reingold-Layouters verbessert.

Zu beachten ist hierbei, dass der Fruchterman-Reingold-Layouter hierfür ein wenig angepasst werden muss. Zum einen darf nicht (wie sonst) am Anfang ein zufälliges Startlayout erzeugt werden, sondern des bereits bestehende benutzt werden. Zum anderen muss der Simulated Annealing Schedule in etwa bei der Schrittweite anfangen, bei der der Centroid-Layouter aufgehört hat, da sonst das bestehende Layout wieder zerstört wird.

Besonders praktisch bei diesem Ansatz ist, dass sich über das Verhältnis der Iterationen von Centroid-Layouter zu Fruchterman-Reingold-Layouter nahezu stufenlos zwischen Geschwindigkeit und Layoutqualität wählen lässt.

## 4.3 Zusätzliche Implementierungen

### 4.3.1 Qualitätsmetrik: Kanten-Kreuzungen

Um die verschiedenen Algorithmen objektiv vergleichen zu können, müssen neben den Laufzeiten und der Skalierbarkeit auch die Qualität der errechneten Layouts beurteilt werden. Obwohl eine manuelle Beurteilung durch einen Menschen im Prinzip möglich und meist in gewissem Rahmen auch praktisch sinnvoll ist, ist sie dennoch stark subjektiv. Daher werden Methoden benötigt, die Qualität eines Graph-Layouts objektiv zu bewerten.

Die durchschnittliche Anzahl an Kantenkreuzungen im Layout (engl. Cross-Edges) ist eine einfach verständliche und dennoch effektive Methode, die Qualität eines Layouts zu beurteilen und findet in vielen Arbeiten zum Layouten von Graphen Anwendung [2], [24]. Dabei wird gezählt, wie häufig Kanten-Kreuzungen im Layout auftreten und die ermittelte Anzahl durch die Anzahl der Kanten im Graph geteilt. Je weniger Kreuzungen auftreten, desto besser gibt das Layout die Struktur der Graphen wieder und desto besser ist das Layout. Allerdings

gibt es durchaus auch Graphen, die nicht zwei dimensional gezeichnet werden können, ohne das sich dabei Kanten kreuzen. Im Allgemeinen ist es auch gar nicht nötig, Kanten-Kreuzungen komplett zu vermeiden, da einige wenige Kreuzungen, speziell bei großen Graphen, kaum auffallen und so die Qualität nur wenig beeinflussen.

Ein praktisches Problem mit dieser Qualitätsmetrik ist ihre aufwändige Berechnung. Im einfachsten Fall muss jede Kante des Graphen mit jeder anderen Kante auf Kreuzung überprüft werden, was zu einer Komplexität von $O(|E|^2)$ führt. Selbst ein unoptimierter Force-directed Layout Algorithmus hat nur eine Komplexität von $O(|V|^2 + |E|)$ und da Graphen praktisch immer deutlich mehr Kanten als Knoten enthalten, ist die naive Berechnung der Kanten-Kreuzungen aufwändiger als die Berechnung des Layouts selbst.

Um diesem Problem zu begegnen, wurde im Rahmen dieser Arbeit versucht, die Komplexität mit einer ähnlichen Methode wie beim Fruchterman-Reingold-Algorithmus zu reduzieren. Der Layoutbereich wird durch ein gleichmäßiges Gitter in einzelne Zellen aufgeteilt. Alle Kanten werden in exakt so viele Unterkanten zerteilt, wie sie Zellen durchlaufen. Dabei befindet sich jede einzelne Unterkante immer innerhalb einer einzigen Zelle. Nun müssen nur noch die Unterkanten der einzelnen Zellen mit allen anderen Unterkanten in der Zelle auf Kreuzungen überprüft werden. Die Gesamtzahl an Kreuzungen in dem Layout ist dann die Summe der Kreuzungen innerhalb aller Zellen. Da die Zerlegung der Kanten in Unterkanten und das Zählen der Kantenkreuzungen der einzelnen Zellen jeweils parallel erfolgen könnten, müsste dieser Ansatz gut horizontal skalierbar sein. Die Anzahl an Vergleichen zwischen Unterkanten, die mit diesem Verfahren nötig sind, beträgt:

$$anzahlVergleiche = |E|^2 * \frac{geschnitteneZellenProKante^2}{|zellen|} \tag{12}$$

Es fallen also $geschnitteneZellenProKante^2/|zellen|$ mal so viele Kantenvergleiche an, wie bei dem naiven Verfahren. Damit würden weniger Kantenvergleiche als bei der naiven Variante anfallen, sobald jede Kante durchschnittlich weniger als $\sqrt{|Zellen|}$ Zellen schneidet. Die Zellanzahl muss also so gewählt werden, dass sie möglichst groß ist, aber die Anzahl geschnittener Zellen nicht zu stark steigt. Wie genau die Anzahl der geschnittenen Zellen von der Gesamtzahl an Zellen abhängt, ist leider nur schwer berechenbar, da dies auch von Länge und Ausrichtung der Kanten abhängt. Zumindest für Kanten, die parallel zu einer Achse verlaufen, lässt sich aber sagen, dass sich die Anzahl der von ihnen geschnittenen Zellen in etwa verdoppelt, wenn man die Anzahl der Zellen vervierfacht, sobald die Zellgröße kleiner als die Länge der Kante ist. Damit bleibt der Term $geschnitteneZellenProKante^2/|zellen|$ ab dann konstant und es findet trotz steigender Zellenanzahl keine Beschleunigung mehr statt. Als grober Richtwert lässt sich also sagen, dass die Größe der Zellen in etwa der durchschnittlichen Kantenlänge des Graphen entsprechen sollte.

In der Praxis bringt dieser Ansatz allerdings kaum eine messbare Beschleunigung der Berechnung der Kantenkreuzungen. Auf einem einzelnen Computer lässt sich die Anzahl der Kreuzungen eines Graphen mit gerade einmal 10000 Kanten damit nicht in realistischer Zeit berechnen. Entweder wird der durch die reduzierten Kantenvergleiche gewonnene Geschwindigkeitsvorteil durch die gestiegene, zu verarbeitende Datenmenge direkt wieder negiert, oder aber die Laufzeit des naiven Algorithmus in Flink ist schlicht zu hoch, als dass die Verbesserung reichen würde, um sie auf ein akzeptables Maß zu reduzieren.

Nachdem der Versuch, die Komplexität der Berechnung zu reduzieren, gescheitert ist, wurde nun versucht, die Ausführungsgeschwindigkeit der Berechnung zu erhöhen. Die Ausführung einer Berechnung mittels Flink bringt eine Menge Overhead mit sich, so dass mit Flink implementierte Algorithmen auf einem einzelnen Rechner immer langsamer sind, als solche, die ohne Flink implementiert wurden. Um also wenigstens für kleinere Graphen auf einem einzelnen Rechner die Anzahl an Kantenkreuzungen berechnen zu können, wurde eine Implementierung erstellt, die Flink nur benutzt um die Kanten des Eingabegraphen zu erhalten. Die erhaltenen Kanten werden dann ohne Flink, innerhalb einer normalen Java-Klasse, auf einem einzelnen Rechner auf Kreuzungen untersucht. Um wenigstens den einen Rechner vollständig auszunutzen nutzt die Implementierung Multithreading, um alle Prozessorkerne gleichzeitig auszunutzen. Da man bei der Ausführung ohne Flink flexibler bei der Umsetzung ist, konnten außerdem einige kleinere Optimierungen umgesetzt werden, wie etwa das Weglassen unnötiger Vergleiche. Letztendlich können mit dieser Implementierung wenigstens die Kreuzungen für Graphen mit einigen Zehntausend Kanten auf einem einzelnen Rechner in wenigen Minuten berechnet werden. Bei sehr großen Graphen kann es natürlich zu Problemen kommen, da der komplette Graph an einen einzelnen Rechner gesendet und dort im Arbeitsspeicher gehalten werden muss.

Ein Punkt, der bei der Berechnung der Kantenkreuzungen zu beachten ist, ist die Frage, ob zwei sich (ganz oder teilweise) überlappende Kanten als sich kreuzend angesehen werden. Tatsache ist, dass sich überlappende Kanten ähnlich schlecht auf die Qualität eines Layouts auswirken, da dadurch Details im Layout verdeckt werden [19]. Die Anzahl an Kanten-Überlappungen sollte also ebenfalls zur Qualitätsbeurteilung herangezogen werden, da ansonsten optisch schlechte Layouts gute Qualitätsmaße erreichen könnten. Ein Beispiel hierfür sieht man bei der Evaluierung des Centroid-Layouters in Kapitel 5.3.4. Der einfachste Weg ist es, die Überlappungen einfach als Kantenkreuzungen zu werten, da nur so die Aussagekräftigkeit der Metrik gewahrt werden kann. Das kann aber zu Problemen bei Vergleichen mit den Ergebnissen anderer Arbeiten führen, falls dies dort anders gehandhabt wird, oder schlicht nicht erwähnt wird, wie genau es dort gehandhabt wird. Beispielsweise wird im GiLa-Paper [2] nicht erwähnt, wie mit Kanten-Überlappungen umgegangen wird.

Während der Evaluierung wird diese Metrik mit der in anderen Arbeiten üblichen (siehe [2], [23]) Abkürzung CRE abgekürzt.

### 4.3.2 Verteiltes Zeichnen von Graphen

Die Hauptmotivation hinter dem Berechnen eines Layouts für einen Graphen ist in der Regel, dass man ein (aussagekräftiges) Bild des Graphen haben möchte. Um ein solches Bild zu erhalten, muss das berechnete Layout gezeichnet werden. Dazu kann bestehende Software, wie beispielsweise GraphViz, verwendet werden. Dafür müsste das mit Gradoop/Flink berechnete Layout in einem geeigneten Format exportiert werden und an einen einzelnen Computer übertragen werden, auf dem die Zeichensoftware ausgeführt wird. Da das errechnete Layout im Normalfall mindestens so viel Speicherplatz benötigt, wie der Graph selbst kann das Exportieren und Übertragen einige Zeit in Anspruch nehmen, speziell dann, wenn der Zielcomputer nicht direkt mit dem Flink-Cluster verbunden ist. Außerdem fällt für das Übertragen des exportierten Layouts und das Ausführen der Zeichensoftware (oder die Automatisierung von beidem) manuelle Arbeit auf Seiten des Nutzers an.

Um die Evaluierungen in Kapitel 5 zu erleichtern und es später Gradoop-Nutzern so einfach wie möglich zu machen, die Graphen, mit denen sie Arbeiten zu Visualisieren, wurde die Funktionalität zum Zeichnen errechneter Layouts direkt für Gradoop implementiert. Die dafür entwickelte Plotter-Klasse ist als Gradoop-DataSink ausgelegt. Dadurch unterscheidet sich das Zeichnen des Graphen und Speichern des entstandenen Bildes nicht vom Exportieren eines Graphen in einem üblichen Dateiformat. Zum Speichern eines Bildes des errechneten Layouts (in einem beliebigen von Flink aus erreichbaren Ort, z.B. dem HDFS) ist so nur eine einzelne Zeile Java-Code notwendig.

Der Plotter bietet vielfältige Konfigurationsmöglichkeiten. So können etwa die zu verwendenden Größen und Farben als auch die Knotenbeschriftungen frei gewählt werden. Auch dynamische Anpassungen der Größen pro Knoten sind möglich, um beispielsweise mittels VertexFusing-Layouter (Kapitel 4.2.2) erzeugte Superknoten entsprechend der Anzahl an Unterknoten zu skalieren. Außerdem ist es möglich, die Skalierung des Graph-Layout automatisch der zur Verfügung stehenden Zeichenfläche anzupassen. Alle genannten Einstellungen sind optional, lediglich die Größe des zu erzeugenden Bildes und das gewünschte Dateiformat müssen explizit angegeben werden.

Der erste Schritt im Zeichenprozess ist das Skalieren der Layout-Koordinaten. Die Größe des Zeichenbereiches ist in der Regel nicht identisch zur Größe des Layoutbereichs, weshalb die Layout-Koordinaten entsprechend umgerechnet werden müssen. Da beide Größen bekannt sind, ist das Umrechnen problemlos über einen einzelnen Map-Task und den Dreisatz möglich. Soll der Graph entsprechend der verfügbaren Zeichenfläche automatisch skaliert werden, wird vorher aus den minimalen und maximalen X und Y Koordinaten des Layouts der Teil des Layoutbereichs bestimmt, in dem sich tatsächlich Knoten und Kanten befinden. Anschließend wird dieser Bereich für die Skalierung auf die Bildgröße herangezogen, anstatt des originalen Layoutbereichs.

Da in einem Graph-Layout lediglich die Knoten (und nicht die Kanten) mit Koordinaten versehen sind, müssen die Koordinaten der Start- und Zielknoten in den Kanten hinterlegt werden, um diese zeichnen zu können. Dies geschieht über einen Join der Kanten mit ihren Start- und Zielknoten. Im Rahmen dieses Joins werden die Start- und Zielkoordinaten in den EPGM-Properties jeder Kante gespeichert. Nun können die Kanten gezeichnet werden. Jeder Rechner-Knoten, auf dem Kanten gespeichert sind, führt eine CombineGroup Operation über seine Kanten aus, in der alle seine Kanten in ein einziges Bild eingezeichnet werden. Das Zeichnen selbst geschieht dabei mittels Java-AWT, weshalb dafür keine externen Bibliotheken benötigt werden. Nach dieser Operation existiert auf jedem Rechnerknoten ein Bild, in dem alle Kanten des Rechnerknotens eingezeichnet sind. Diese Einzelbilder werden nun mittels einer Reduce-Operation paarweise kombiniert. Dabei werden die beiden Bilder (die jeweils einen transparenten Hintergrund besitzen) übereinander gelegt und so zu einem neuen Bild zusammengefügt, das die Kanten beider Ursprungsbilder enthält. Dieses paarweise Kombinieren wird parallel und wiederholt durchgeführt, bis nur noch ein Bild übrig ist, das alle Kanten des Graphen enthält.

Das so erzeugte Bild enthält jedoch noch keine Knoten. Da die Knoten immer über den Kanten gezeichnet werden sollen, können diese nicht gleichzeitig mit den Kanten gezeichnet werden, da sonst später gezeichnete Kanten früher gezeichnete Knoten überdecken würden. Für den Fall, dass der Nutzer das Zeichnen von Knoten nicht explizit deaktiviert hat, wird der zuvor beschriebene Prozess zum Zeichnen von Kanten nun mit den Knoten wiederholt. Die beiden Bilder von Knoten und Kanten werden anschließend zu einem Gesamtbild übereinander gelegt (wobei sich das Knoten-Bild oben befindet). Zum Schluss wird dieses eine Bild nun mittels einer Map-Operation mit einer Hintergrundfarbe hinterlegt. Damit ist die Zeichnung fertig. Eine schematische Darstellung des Zeichen-Prozesses ist in Abbildung 10 zu sehen. Diese muss nun lediglich mittels Java-ImageIO in das vom Nutzer gewünschte Bildformat konvertiert und im HDFS gespeichert werden.

Die vorgestellte verteilte Implementierung skaliert sehr gut und ist in der Lage, selbst große Layouts in wenigen Sekunden zu zeichnen.

Alle in dieser Arbeit gezeigten Bilder von Graphen wurden mit Hilfe der in diesem Kapitel erläuterten Plotter-Klasse erzeugt.

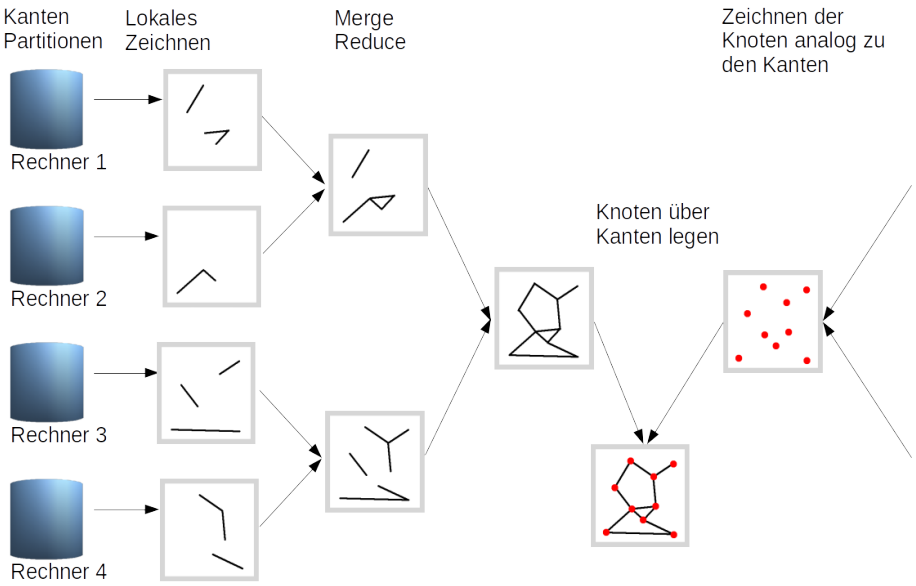


Abbildung 10: Verteiltes Zeichnen von Graphen

# 5 Evaluierung

Im folgenden Kapitel werden die zuvor vorgestellten Implementierung hinsichtlich ihrer Geschwindigkeit und Skalierbarkeit evaluiert. Hierfür werden zuerst die verwendeten Datensätze und die Testumgebung vorgestellt. Es folgen die Einzelevaluierungen der verschiedenen Implementierungen und abschließend ein Vergleich der Ergebnisse der Einzelevaluierungen.

## 5.1 Datensätze

Die für die Evaluierung verwendeten Datensätze sind in Tabelle 2 aufgeführt. Bei all diesen Datensätzen handelt es sich um reale (nicht künstlich generierte) Datensätze, damit die Evaluierung auch den später im Praxiseinsatz verwendeten Datensätzen entspricht. Alle Datensätze stammen aus der "SuitSparse Matrix Collection" der Universität Florida [29].

Es handelt sich dabei um die selben Datensätze, die zur Evaluierung des GiLa-Algorithmus [2] verwendet wurden, weshalb ein direkter Vergleich zwischen den Algorithmen dieser Arbeit und der originalen GiLa-Implementierung möglich ist. Allerdings konnten die im GiLa-Paper als "grund" und "asic-320" bezeichneten Datensätze nicht gefunden werden, weshalb sie in dieser Evaluierung nicht verwendet werden konnten. Außerdem war der facebook-Datensatz nicht Teil der Evaluierung im GiLa-Paper.

Alle Datensätze aus der "SuitSparse Matrix Collection" liegen im MTX-Format [30] vor. Da Gradoop standardmäßig keine Möglichkeit bietet diesen Dateityp zu importieren, wurde eine eigene Gradoop-DataSource implementiert, die es erlaubt, Datensätze im MTX-Format direkt und ohne Zwischenschritte in Gradoop zu laden. Durch diese DataSource kann nun jeder einzelne Datensatz der "SuitSparse Matrix Collection" problemlos mit Gradoop verwendet werden, was auch für spätere Evaluierungen mit Gradoop hilfreich sein kann.

Einige der Datensätze enthalten Selbstkanten und Multikanten. Diese sind für das Layouting irrelevant und werden beim Import des Datensatzes direkt entfernt. Dies ist äquivalent zu der in [2] durchgeführten Evaluierung.

Die Datensätze enthalten lediglich Strukturinformationen für die Graphen und keine weiteren Eigenschaften, die in den EPGM-Properties gespeichert werden würden. Diese sind für das Layouting auch unerheblich, da dabei lediglich mit der Struktur des Graphen gearbeitet wird.

Tabelle 2: Für die Evaluierung verwendete Datensätze

| Name | Knoten | Kanten | Durchmesser |
|---|---|---|---|
| facebook | 4039 | 88234 | 8 |
| add32 | 4960 | 9462 | 28 |
| ca-GrQc | 5242 | 14496 | 17 |
| p2p-Gnutella04 | 10876 | 39994 | 9 |
| pGp-giantcompo | 10680 | 48932 | 17 |
| ca-CondMat | 23133 | 93497 | 14 |
| p2p-Gnutella31 | 62589 | 147892 | 11 |
| amazon0302 | 262111 | 899782 | 32 |
| com-DBLP | 317080 | 1049866 | 21 |
| com-Amazon | 334863 | 925872 | 44 |
| roadNet-PA | 1087562 | 1541514 | 782 |

Es ist anzumerken, dass nicht immer jeder Datensatz für jede Evaluierung eingesetzt wird. Insbesondere die sehr großen Datensätze kommen nur bei Algorithmen zum Einsatz, die in der Lage sind, diese in angemessener Zeit zu verarbeiten. Außerdem werden große Datensätze nur mit maximaler Parallelität bearbeitet, um in akzeptabler Zeit Ergebnisse zu generieren. Aufgrund der hohen Kantenzahl der größten Graphen konnten für diese im Zuge der Evaluierung keine Kantenkreuzungswerte berechnet werden.

## 5.2 Testumgebung und Messmethoden

Um die Evaluierung der Algorithmen zu erleichtern, wurde eine eigene Klasse zum Benchmarken von Layoutalgorithmen erstellt. Dieser können per Kommandozeilenargumenten eine Vielzahl von Optionen übergeben werden, wie etwa der zu verwendende Algorithmus und dessen Parameter, zu berechnende Statistiken, Eingabedatensatz und dessen Format etc. Die Klasse ließt den Datensatz über die passende Gradoop-DataSource ein, konfiguriert den gewünschten Layout-Algorithmus mit den übergebenen Parametern und führt das Layouting aus. Anschließend wird das fertige Layout zwischengespeichert. Im nächsten Schritt werden die gewünschten Statistiken/Qualitätsmetriken für das Layout errechnet und das Layout in das gewünschte Ausgabeformat konvertiert. Als Ausgabeformat kann, neben allen von Gradoop standardmäßig unterstützen Formaten, auch ein Bild generiert werden. In diesem Fall wird mittels der Plotter-Klasse ein Bild des Layouts gezeichnet und im HDFS abgelegt. Am Ende speichert die Benchmark-Klasse die gemessene Laufzeit für das Layouting und die errechneten Werte für spätere Auswertungen in einer CSV-Datei. Da das Ergebnis des Layoutings zwischengespeichert wird, ist das erneute Berechnen von Qualitätsmetriken oder das Zeichnen des Layouts möglich, ohne dafür das Layouting erneut durchführen zu müssen.

Das Messen der Laufzeit geschieht mittels der Flink-eigenen Funktion getNetRuntime(). Diese liefert die Gesamtlaufzeit der Flink-Ausführung abzüglich der Zeit für jegliche Vorlaufoperationen, wie das Erstellens und Optimieren des Ausführungsplans. Damit ist eine relativ genaue Messung der Laufzeit möglich. Da das Zeichnen des Layouts und die Berechnung der Qualitätsmetriken in separaten Flink-Jobs erfolgen, verfälschen diese nicht das Messergebnis. Prinzipbedingt beinhalten die gemessen Laufzeiten alle Operationen des Flink-Jobs, also auch die Operationen zum Laden und Speichern der Datensätze.

Weiterhin wurde ein Shell-Script erstellt, in dessen Konfigurationsbereich je eine Liste von möglichen Werten für jeden Eingabeparameter der Benchmarkklasse hinterlegt werden können. Das Shell-Script wird dann bei Ausführung für jede mögliche Wertekombination über das Flink CLI-Tool die Benchmarkklasse mit diesen Werten starten. Die Ergebnisse der Ausführungen können dann in der von der Benchmarkklasse erstellten CSV-Datei und im HDFS eingesehen werden.

Alle Tests zu Laufzeiten und Skalierbarkeit fanden auf dem Big-Data-Cluster der Datenbankabteilung der Universität Leipzig statt. Das Cluster besteht aus 16 Maschinen. Jede Maschine verfügt über einen Intel Xeon CPU E5-2430 v2 Prozessor mit 6 Kernen und 48GB RAM. Da das Cluster während der Evaluierung zur alleinigen Benutzung zur Verfügung stand, können Beeinträchtigungen der Messergebnisse durch die Aktivität anderer Nutzer weitgehend ausgeschlossen werden. Wenn während der Evaluierung von “Parallelität” die Rede ist, bezieht sich das immer auf die Anzahl der beteiligten Maschinen und nicht auf die Anzahl der genutzten Prozessorkerne.

Einige Testläufe mit kleineren Datensätzen, deren Ziel keine Laufzeit- oder Skalierbarkeitsmessungen waren, wurden außerdem auf einem Thinkpad E470 mit Intel Core i5-7200U Prozessor und 8 GB RAM durchgeführt. An allen Diagrammen, bei denen Laufzeiten gemessen werden, ist vermerkt ob diese Messungen auf dem Rechencluster oder dem Notebook erfolgten.

Sämtliche Algorithmus-Parameter, die in den folgenden Evaluierungen nicht explizit genannt werden (wie beispielsweise die Größe der Layoutbereiche), entsprechen den automatisch generierten Standardwerden des Algorithmus. (siehe Kapitel 4)

## 5.3 Ergebnisse

### 5.3.1 Random-Layouter

Die Evaluierung des Random-Layouters lieferte erwartete Ergebnisse. Sämtliche Datensätze konnten unabhängig ihrer Größe in wenigen Sekunden (zwischen 4s und 16s) verarbeitet werden. Zu beachten ist, dass eine Erhöhung der Parallelität dabei durchgehend zu einer Verlangsamung der Ausführung führt, da die dabei benötigte Übertragung von Daten zwischen den beteiligten Rechnerknoten Zeit kostet und die dadurch gewonnene Rechnerleistung jedoch nicht benötigt wird.

Die damit erzeugten Layouts geben die Struktur der Graphen nicht wieder und die Zeichnungen der Layouts sind für alle Graphen Vierecke in der Farbe der Kanten. Als Beispiel hierfür ist in Abbildung 11 ein zufälliges Layout des Facebook-Datensatzes zu sehen. Außerdem enthalten die Layouts eine große Anzahl an Kantenkreuzungen, wie in Abbildung 12 zu sehen ist. Die Anzahl an Kreuzungen pro Kante für ein zufälliges Layout liefern eine gute Vergleichsbasis für die Leistungsfähigkeit der verschiedenen Layoutalgorithmen.

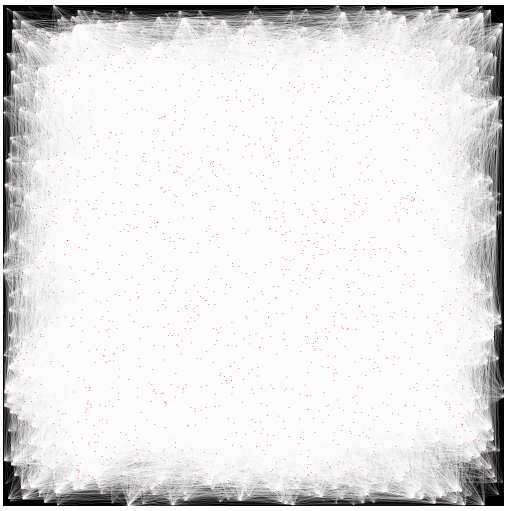

Abbildung 11: Zufälliges Layout des Facebook-Datensatzes

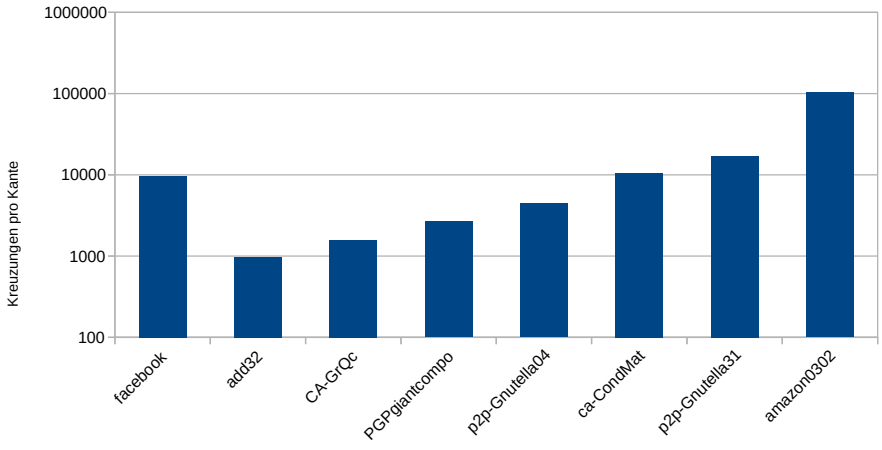


Abbildung 12: Kantenkreuzungen zufälliger Layouts

### 5.3.2 Fruchterman-Reingold-Layouter

Der Fruchterman-Reingold-Layouter bildet die Grundlage für alle Implementierungen in dieser Arbeit. Durch Tests an ihm gewonnene Erkenntnisse lassen sich gut auf die anderen Algorithmen dieser Arbeit übertragen, weshalb dieser Algorithmus besonders ausführlich evaluiert wird.

Das Diagramm in Abbildung 13 und die Bilderserie in Abbildung 14 zeigen, wie sich die Anzahl der Iterationen auf die Qualität des errechneten Layouts auswirkt. Hierfür wird der Facebook-Datensatz verwendet, da sich in diesem - bei einem ausreichend guten Layout - besonders gut einzelne Cluster erkennen lassen. Dabei ist unbedingt zu beachten, dass hier nicht der Durchlauf eines einzelnen Layouting-Vorgangs betrachtet wird, sondern mehrere separate Vorgänge miteinander verglichen werden. Durch den in Kapitel 4.1.2 beschriebenen Simulated Annealing Schedule haben alle Layouting-Vorgänge, unabhängig von der Anzahl ihrer Iterationen, die gleichen Start- und Endschrittweiten. Mehr Iterationen führen zu mehr und kleineren Zwischenschritten und damit zu besserer Qualität. Auf diese Weise wird die zur Verfügung stehende Anzahl an Iterationen bestmöglich genutzt. Allerdings führt das dazu, dass das Layout, welches mit 10 Iterationen erzeugt wird, nicht identisch ist mit dem, welches entstehen würde, falls man das fertige 5-Iterationen-Layout für weitere 5 Iterationen laufen lassen würde. Dennoch veranschaulichen die Bilder näherungsweise, welche Phasen ein Graph während des Layouting-Prozesses durchläuft.

In Abbildung 14 ist zu sehen, dass bei einer einzelnen Iteration kaum eine Veränderung zur ursprünglichen zufälligen Anordnung der Knoten auftritt. Die Knoten sind immer noch weitgehend zufällig und gleichmäßig verteilt angeordnet. Durch die zufällige Anordnung der Knoten überkreuzen sich die Kanten sehr häufig, weshalb sich ein hoher CRE-Wert ergibt. In dem Layout ist die interne Struktur des Graphen in keinster Weise erkennbar. Deshalb und aufgrund der hohen Anzahl an Kantenkreuzungen handelt es sich um ein schlechtes Layout, was nach einer einzelnen Iteration aber zu erwarten war.

Bei 5 Iterationen hatten die Knoten ausreichend Gelegenheit, sich von ihrer ursprünglichen Anordnung weg zu bewegen. Da die Knoten in ihrer ursprünglichen zufälligen Anordnung meist mehr als k (die optimale Distanz zwischen zwei verbundenen Knoten im Fruchterman-Reingold-Algorithmus) voneinander entfernt waren, dominieren die Anziehungskräfte der Kanten und die Knoten bewegen sich aufeinander zu. Da der Facebook-Graph sehr viele Kanten enthält (~88000 Kanten bei nur 4093 Knoten) sind die Anziehungskräfte besonders dominant. Da die Knoten langsam beginnen, sich zu ordnen, sinkt die Anzahl an Kreuzungen pro Kante.

Bei 10 Iterationen ist die Kontraktion des Graphen durch die Anziehungskräfte weitgehend abgeschlossen. Die Kanten haben die Knoten so weit zusammen gezogen, bis die stärker werdenden Abstoßungskräfte zwischen den Knoten die Kontraktion stoppten. Alle Knoten sind auf einem relativ kleinen Bereich

zusammengezogen. Es sind erste kleine Cluster erkennbar. Die Knoten werden immer geordneter (auch wenn das auf dem Bild bisher kaum erkennbar ist), die Anzahl an Kantenkreuzungen sinkt stark.

Bei 25 Iterationen ist die große Ansammlung von Knoten in mehrere klar voneinander getrennte Cluster zerfallen. Die innere Struktur des Graphen wird erkennbar. Wie bei einem Freundesgraphen eines sozialen Netzwerkes zu erwarten ist, sind mehrere Cluster zu erkennen, die innerhalb stark und miteinander nur relativ schwach verbunden sind. Die Anzahl der durchschnittlichen Kreuzungen pro Kante ist verglichen mit dem Anfangswert stark zurückgegangen. Dass absolut gesehen immer noch relativ viele Kreuzungen auftreten liegt an der hohen Dichte dieses Graphen und lässt sich nicht vollständig vermeiden.

Bei 50 Iterationen sind die ursprünglichen größeren Cluster noch weiter in kleinere Untercluster zerfallen. Die Cluster, speziell die, die gegenseitig nur schwach verbunden sind, rücken von einander weg.

Zwischen 50 und 100 Iterationen besteht optisch kaum noch ein Unterschied. Auch die Anzahl an Kantenkreuzungen sinkt nur noch minimal. Das Layout hat ein (womöglich nur lokales) Optimum erreicht. Das Layout war bereits nach 50 Iterationen so gut, dass mit weiteren Iterationen kaum noch eine Verbesserung erreicht werden kann.

Das fertige Layout nimmt nur einen kleinen Teil des ursprünglichen Layoutbereichs ein. Die Größe des ursprünglichen Layoutbereichs wurde anhand der Anzahl an Knoten in dem Graphen festgelegt. Allerdings ist dieser Graph so dicht verbunden, dass die Anziehungskräfte der Kanten die Knoten auf kleinstem Raum zusammenziehen und der Graph weniger Platz einnimmt als ursprünglich angenommen. Die Plotter-Klasse kann in solchen Fällen das fertige Layout auf die komplette Größe des Layoutbereichs hoch skalieren. Darauf wurde hier aber bewusst verzichtet, damit die Entwicklung der Layouts auf den Bildern unverfälscht zu erkennen ist.

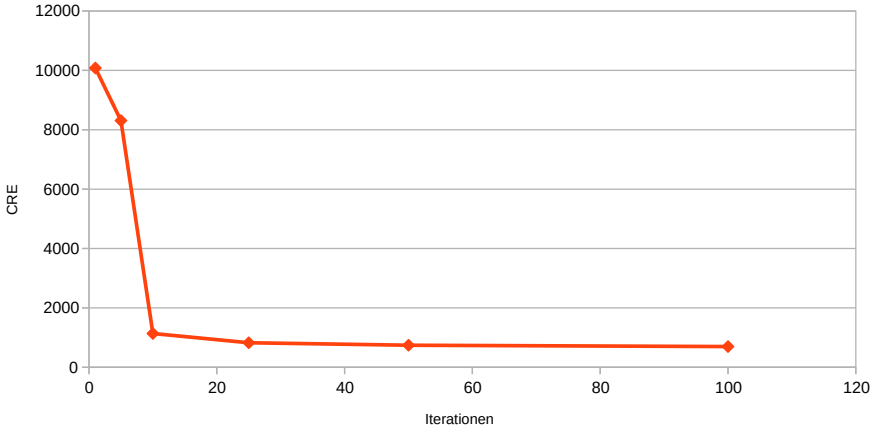


Abbildung 13: Verbesserung des Layouts durch Iterationen

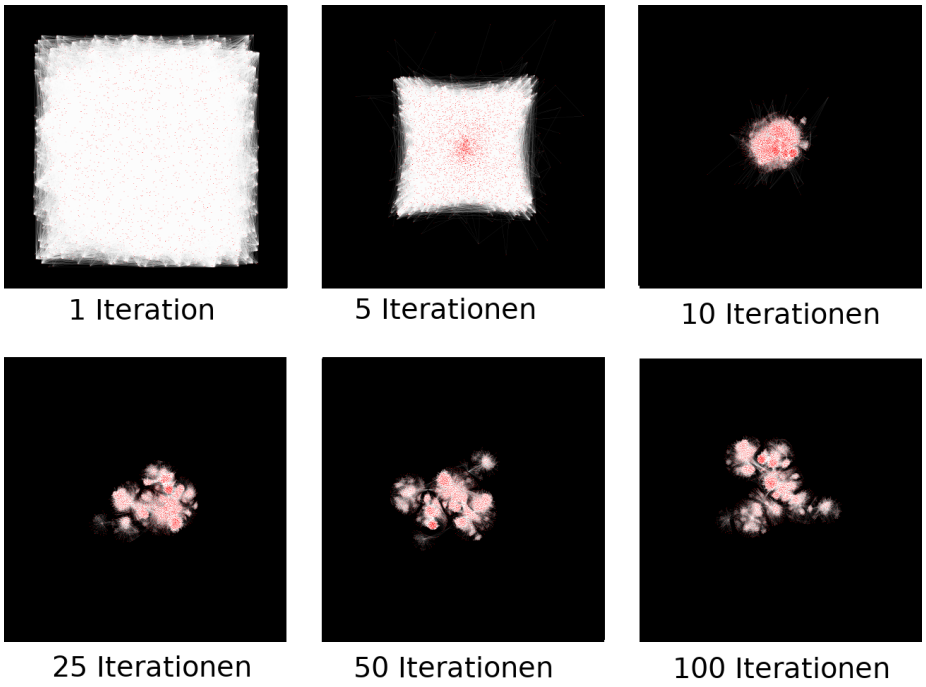


Abbildung 14: Layouts nach Iterationen

Beim Betrachten der Bilder in Abbildung 14 fällt auf, dass ab einem gewissen Zeitpunkt während des Layoutings die Knoten dicht gebündelt vorliegen,

besonders zwischen der Kontraktionsphase und der anschließenden Aufteilung in kleinere Cluster. Eine Idee des Fruchterman-Reingold-Algorithmus war es, Abstoßungskräfte nur zwischen nah beieinander liegenden Knoten zu berechnen und so Rechenzeit einzusparen. Wenn aber - wie hier - die Knoten so dicht gebündelt vorliegen, verliert diese Strategie an Effektivität, da nun trotzdem viele Abstoßungskräfte zwischen Knoten berechnet werden müssen. Dies wirkt sich auch spürbar auf die Laufzeiten aus. Wie in Abbildung 15 zu sehen ist steigt die Gesamtlaufzeit im Bereich der Iterationsanzahlen, bei denen die Knoten stark gehäuft vorliegen (ca. 5 - 10 Iterationen) besonders stark. Davor und danach, wenn die Knoten stärker verteilt sind, steigt die Ausführungsdauer linear mit der Iterationsanzahl. Dieser Effekt ist umso problematischer, je dichter der zu layoutende Graph ist und kann bei besonders großen und dichten Graphen zu massiv verlängerten Laufzeiten führen.

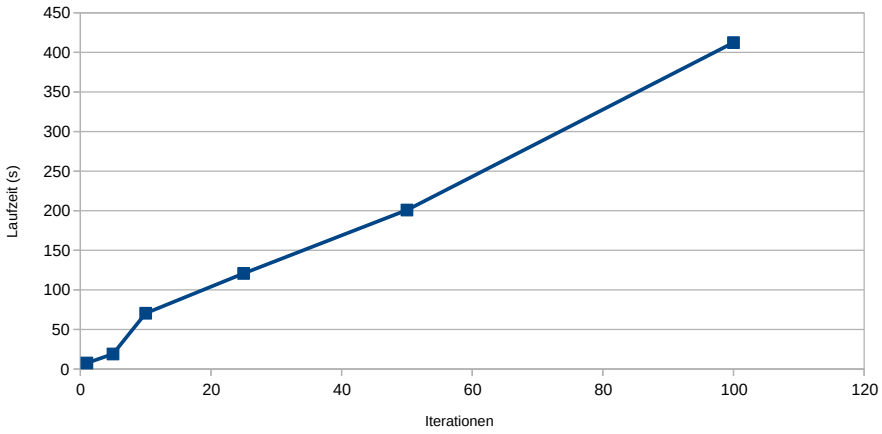


Abbildung 15: Laufzeiten für Durchläufe (Notebook)

An dieser Stelle lässt sich bereits sagen, dass das Prinzip funktioniert und die Implementierung korrekt arbeitet. Durch iteratives Anwenden von Anziehungs- und Abstoßungskräften zwischen Knoten lässt sich ein Layout erzeugen, das die Struktur des Graphen gut wiedergibt. Erfreulicherweise sind dazu nur wenige Iterationen nötig. Bereits ab ca. 25 Iterationen erhält man (zumindest für dieses Beispiel) brauchbare Ergebnisse.

Wie erwartet, ist die Geschwindigkeit auf einem einzelnen Rechner nicht beeindruckend. Die Ausführung in Flink bringt zu viel Overhead, mit um auf einer einzelnen Maschine mit einer zentralisierten Implementierung mithalten zu können. Der Vorteil von Flink ist aber die horizontale Skalierbarkeit. Im folgenden wird untersucht, wie gut die Implementierung im Falle einer verteilten Ausführung skaliert und ob damit auch größere Graphen verarbeitet werden können.

Um die Skalierbarkeit des Algorithmus zu testen, wird dieser mit den Datensätzen facebook, add32, CA-GrQc, pGp-giantcompo, p2p-Gnutella04, ca-CondMat und p2p-Gnutella31 und jeweils auf 1, 2, 4, 8 und 16 Maschinen des Rechenclusters dargestellt. Die Laufzeiten dieser Durchläufe sind in Abbildung 16 aufgeführt. Da sich die Werte teils über einen großen Wertebereich erstrecken aber teils auch nah zusammen liegen, ist die Y-Achse logarithmisch skaliert, um die Lesbarkeit zu verbessern. Auf der horizontalen Achse sind die verschiedenen Datensätze bei jeweils jeder getesteten Parallelität aufgeführt.

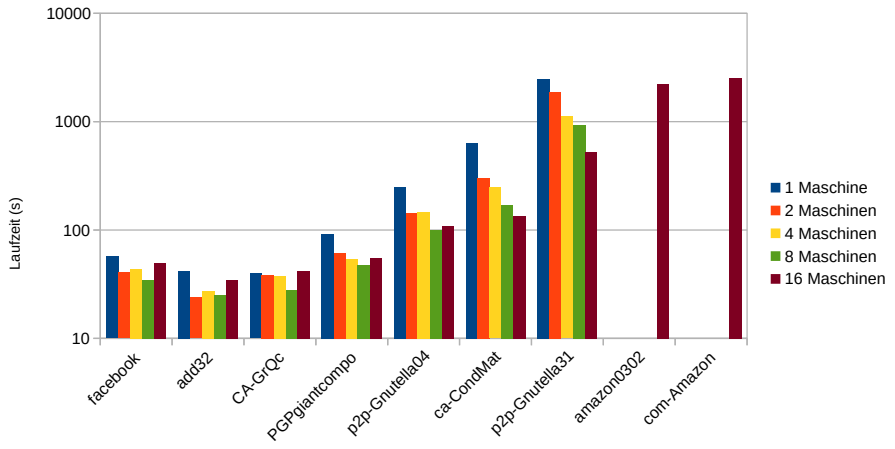


Abbildung 16: Skalierbarkeit Fruchterman-Reingold (Cluster)

In Abbildung 16 ist gut zu erkennen, dass die Ausführung mit einer Parallelität von 1 stets am längsten dauert, selbst für die kleinsten Datensätze. Das bedeutet, dass die verteilte Ausführung grundsätzlich einen Geschwindigkeitsvorteil gegenüber der Ausführung auf einem einzelnen Rechner bringt. Allerdings zeigt sich auch, dass eine Erhöhung der Parallelität nicht immer von Vorteil ist. So ist beispielsweise für die kleinsten Datensätze die volle Parallelität von 16 am zweit langsamsten. Bei kleinen Datensätzen überwiegt anscheinend der, durch die erhöhte Verteilung aufkommende, Kommunikationsoverhead den Gewinn an Rechenleistung. Mit steigender Größe des Graphen verschiebt sich die optimale Parallelität immer weiter nach oben. Während beim zweit kleinsten Datensatz add32 eine Parallelität von 2 zur geringsten Laufzeit führt, ist beim nächst größeren Datensatz CA-GrQc eine Parallelität von 8 am schnellsten. Für die größten Datensätze nimmt die Geschwindigkeit mit zunehmender Parallelität strikt zu.

Außerdem lässt sich in Abbildung 16 gut erkennen, dass nicht nur die Knotenanzahl des Eingabegraphen Einfluss auf die Laufzeit hat. Der erste Datensatz (facebook) hat ca. 1000 Knoten weniger als der zweite (add32). Dennoch benötigt das Layouting von Facebook mehr Zeit. Das ist nur mit der deutlich größeren

Kantenanzahl des Facebook-Datensatzes zu erklären (~88000 verglichen mit 9462). Außerdem hat auch die Struktur des Graphen selbst einen erheblichen Einfluss auf die Laufzeit. Ein gutes Beispiel hierfür sind die Datensätze pGp-giantcompo und p2p-Gnutella04. Beide haben ca. 10000 Knoten und ähnlich viele Kanten. Dennoch ist die Laufzeit für den zweiten Datensatz erheblich höher. Das ist damit zu erklären, dass der zweite Graph einen geringeren Durchmesser aufweist als der Erste. Dadurch sind dessen Knoten dichter gepackt und die Abstoßungskraftberechnung wird aufwändiger, was die gestiegene Laufzeit erklärt.

Auch die Graphen mit mehreren 100.000 Knoten lassen sich mit dem Algorithmus layouten, ohne dass die Laufzeit stärker ansteigt, als zu erwarten wäre. Eine Ausnahme hiervon bildet der Datensatz com-DBLP, für den der Algorithmus nicht in annehmbarer Zeit ein Ergebnis liefert. Durch die hohe Anzahl an Kanten in diesem Graphen in Verbindung mit dem geringen Durchmesser liegen viele Knoten derart dicht aneinander, dass die Abstoßungsberechnung enorm rechenaufwändig wird.

Soweit lässt sich festhalten, dass es möglich ist, den Fruchterman-Reingold-Algorithmus in Flink zu implementieren und mit dessen Hilfe in annehmbarer Zeit brauchbare Layouts für Graphen zu errechnen. Die Implementierung skaliert sowohl mit der Knotenanzahl als auch horizontal mit steigender Parallelität, wobei aber durchaus noch Verbesserungspotential besteht. Auch leidet die Implementierung unter den typischen Problemen des Algorithmus, wie etwa dem Geschwindigkeitsverlust bei Graphen, in denen die Knoten dicht gehäuft vorliegen.

### 5.3.3 Sampling-Layouter

Der Sampling-Layouter unterscheidet sich vom Fruchterman-Reingold-Layouter lediglich darin, dass in jeder Iteration die Abstoßungskräfte nur mit einer zufälligen Teilmenge der Knoten berechnet und dann entsprechend hochgerechnet werden. So müsste sich über die Größe diese Teilmenge zwischen erhöhter Geschwindigkeit und Layoutqualität wählen lassen.

Nun gilt es zu untersuchen, wie genau sich unterschiedliche Prozentsätze (Samplingraten) auf Geschwindigkeit und Qualität des Layoutings auswirken. Hierfür wird wieder der Facebook-Datensatz verwendet, da bei diesem die Qualität eines Layouts optisch gut abschätzbar ist. Außerdem wird vorerst eine einzelne Maschine zur Berechnung verwendet und es werden jeweils genau 50 Iterationen absolviert. Abbildung 17 zeigt, wie sich die Laufzeiten verändern, wenn die Größe der Teilmenge stückweise von 90% zu 10% der ursprünglichen Knoten gesenkt wird und wie sich die Qualität der errechneten Layout mit dieser Absenkung verändert (gemessen an den durchschnittlichen Kantenkreuzungen pro Kante).

In Abbildung 18 sind die mit den unterschiedlichen Samplingraten erzeugten Layouts zu sehen. Die Layouts wurden für eine bessere Erkennbarkeit auf die Größe des Bildbereiches hoch skaliert.

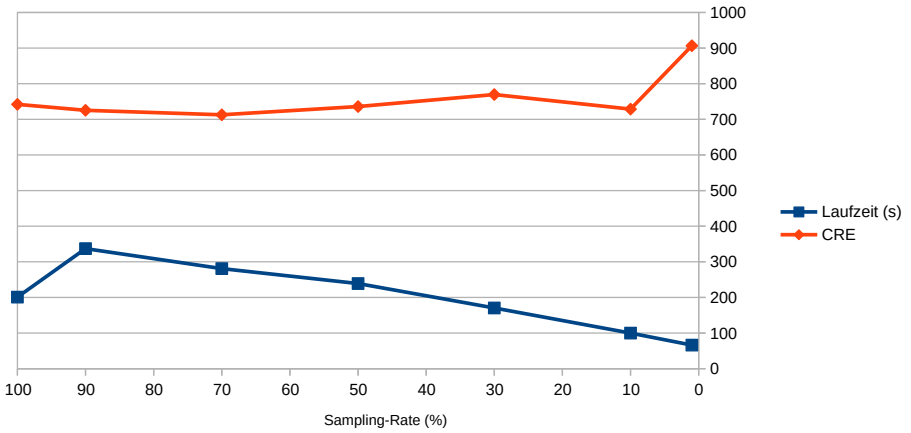


Abbildung 17: Auswirkung unterschiedlicher Samplingraten (Notebook)

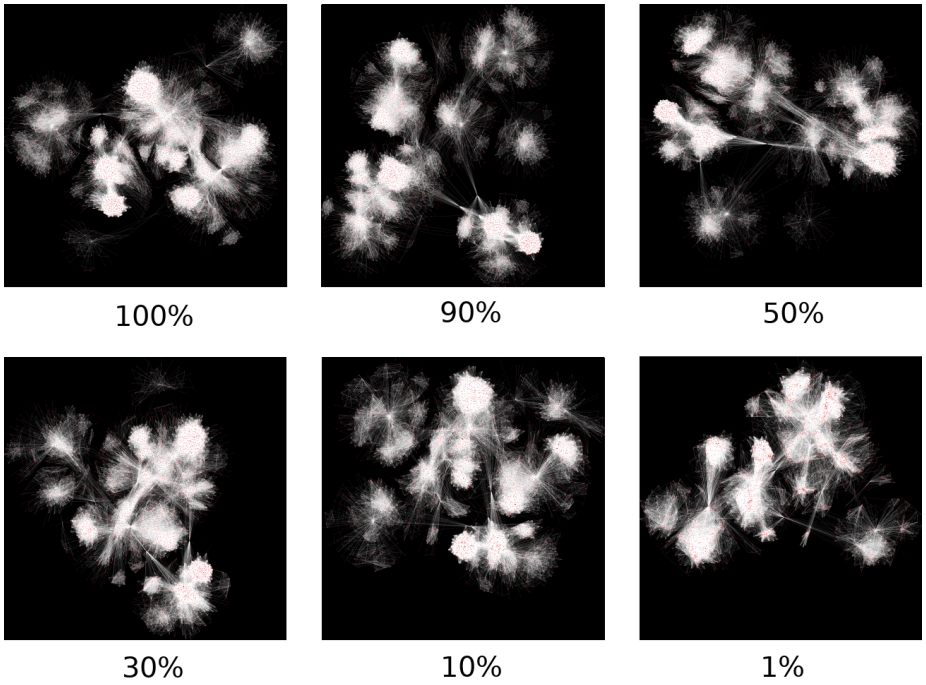


Abbildung 18: Layouts bei unterschiedlichen Samplingraten

Da nicht mehr alle ausreichend nahen Knoten für die Abstoßungsberechnung herangezogen werden können, kann eine Optimierung in der normalen Fruchterman-Reingold-Implementierung nicht mehr verwendet werden (siehe dazu Kapitel 4.1.2, 4.2.1). Das führt dazu, dass die Ausführung mit einer hohen Samplingrate langsamer ist als ohne Sampling. Darin ist auch der Anstieg der Ausführungszeit von 100% (kein Sampling) zu 90% in Abbildung 17 begründet. Sobald Sampling genutzt wird, sinkt die Laufzeit für die Berechnung eines Layouts linear mit der Samplingrate.

Erstaunlicherweise führen moderate Samplingraten kaum zu einer Verschlechterung der Layout-Qualität. Von 100% bis 10% bleibt die über Kantenkreuzungen gemessene Qualität (Abbildung 17) nahezu konstant. Die Qualität steigt mit sinkender Samplingrate stellenweise etwas, was aber wahrscheinlich lediglich Zufall ist. Erst bei sehr kleinen Samplingraten, wie etwa 1%, ist rechnerisch ein Qualitätsverlust erkennbar. Das spiegelt sich auch in den Zeichnungen der Layouts in Abbildung 18 wieder. Optisch ist zwischen den meisten Bilder kein Unterschied zu erkennen. Lediglich das Bild für 1% erscheint minimal weniger detailreich zu sein.

Da, wie bereits erwähnt, bei der Implementierung des Sampling-Layouters keine Symmetrie-Effekte zur Optimierung verwendet werden können, werden zur Berechnung der Abstoßungskräfte mehr Join-Operationen benötigt als beim Fruchterman-Reingold-Layouter, wenn auch mit weniger Elementen pro Join. Daher muss nun untersucht werden, wie sich dies auf die Skalierbarkeit des Algorithmus auswirkt. Dafür wird, wie bereits beim Fruchterman-Reingold-Layouter, der Algorithmus auf dem Rechencluster mit verschiedenen Parallelitäten und Datensätzen ausgeführt. Als Samplingrate wurde für diese Versuche 10% gewählt, da laut den vorherigen Versuchen so gute Geschwindigkeitsverbesserungen möglich sind, ohne die Qualität des Layouts negativ zu beeinflussen. Das Ergebnis wird in Abbildung 19 gezeigt.

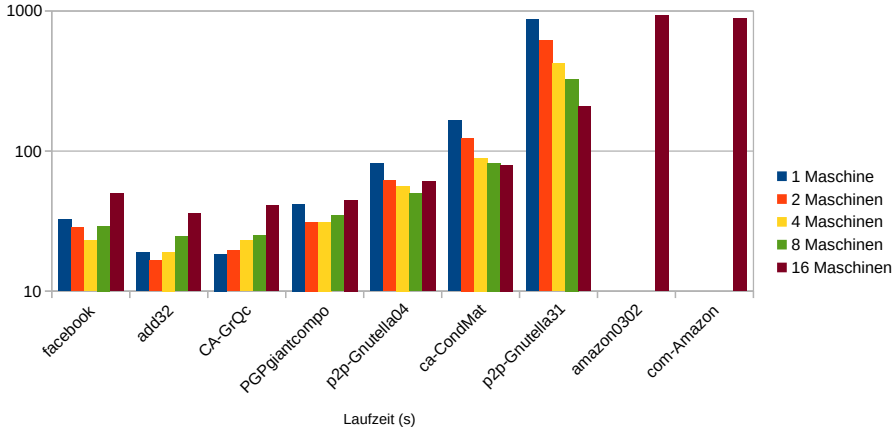


Abbildung 19: Skalierbarkeit Sampling-Layouter (Cluster)

Beim Betrachten von Abbildung 19 fällt auf, dass die Laufzeiten, speziell für die größeren Datensätze, deutlich geringer sind als beim Fruchterman-Reingold-Layouter, so sind die Laufzeiten z.B. für den größten Datensatz je nach Parallelität zwischen 60% und 75% geringer. Das liegt zwar unterhalb der theoretisch erwarteten Einsparung von 90%, ist aber dennoch eine deutliche Verbesserung. Ansonsten verhält sich die Skalierbarkeit im Prinzip ähnlich zum Fruchterman-Reingold-Layouter. Für geringe Datensatzgrößen überwiegt der Overhead der verteilten Ausführung den Gewinn an Rechenleistung, aber je größer der Datensatz ist, desto mehr lohnt sich die Parallelisierung. Allerdings scheint sich hier der Parallelisierungsoverhead für kleine Graphen noch massiver auszuwirken als bei dem Basisalgorithmus, denn hier ist die Ausführung bei voller Parallelität (16) bei den kleinen Datensätzen erheblich langsamer als bei niedriger Parallelität.

Wenn eine geeignete Samplingrate gewählt wird, ist dieser Algorithmus durchaus in der Lage, einen Geschwindigkeitsvorteil gegenüber dem Fruchterman-Reingold-Algorithmus zu erzielen, ohne dabei allzu schwerwiegende Qualitätsprobleme im Layout hervorzurufen. Die prinzipbedingten Schwächen des Fruchterman-Reingold-Algorithmus, wie die langen Laufzeiten bei dicht gepackten Graphen behebt dieser Ansatz allerdings nicht.

### 5.3.4 Centroid-Layouter

Der Ansatz hinter dem Centroid-Layouter verspricht in der Theorie hervorragende Skalierbarkeit, da die Abstoßungsberechnung nur über eine kleine Gruppe von Cluster-Zentren erfolgt. Da der Datensatz der Cluster-Zentren klein genug ist, um per Broadcast an alle beteiligten Rechnerknoten übermittelt zu werden, sollte kaum Netzwerkverkehr im Rahmen der Abstoßungsberechnung anfallen, welcher normalerweise ein Hauptfaktor für den Overhead bei großer Parallelität

ausmacht. Wie üblich wird der Algorithmus bei unterschiedlicher Parallelität mit unterschiedlichen Datensätzen auf dem Cluster ausgeführt und die Laufzeiten für je 50 Iterationen gemessen. Die Ergebnisse der Messung sind in Abbildung 20 zu sehen. Auch hier wird wieder eine logarithmische Skalierung der Y-Achse verwendet, um den großen Wertebereich besser darstellen zu können.

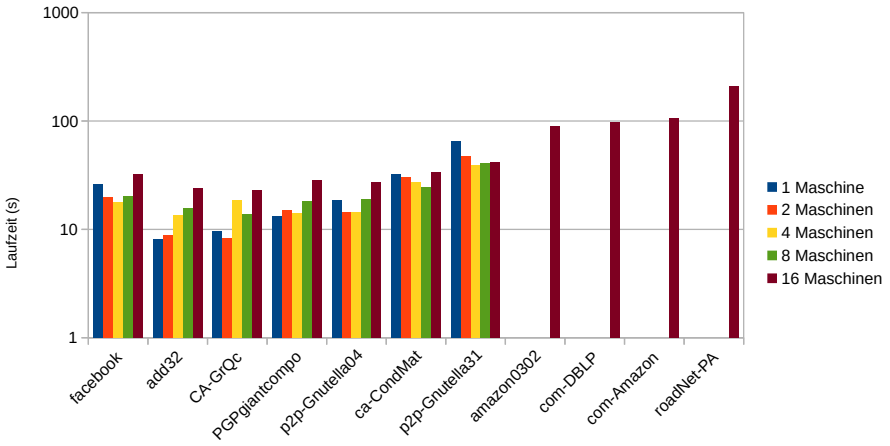


Abbildung 20: Skalierbarkeit Centroid-Layouter (Cluster)

Abbildung 20 liefert einige interessante Erkenntnisse. Offensichtlich ist die horizontale Skalierbarkeit nicht so gut wie erwartet. Anscheinend sorgen Faktoren, wie die Anziehungskraftberechnung oder die Berechnung der Zentroide, für ausreichend viel Kommunikationsoverhead, so dass besonders für kleine Datensätze eine hohe Parallelität eher hinderlich ist. Selbst für mittelgroße Datensätze ist eine maximale Parallelität nicht von Vorteil. Allerdings führt eine moderate Parallelität für mittelgroße Datensätze zu einer merklichen Geschwindigkeitssteigerung. Ab einer gewissen Mindestgröße wirkt sich eine hohe Parallelität nicht mehr störend aus. Gut ist die Laufzeit und Skalierbarkeit bezüglich der Graph-Größe. Selbst der größte Graph kann in etwas mehr als einer Minute verarbeitet werden. Außerdem scheint (ab einer bestimmten Mindestgröße) die Laufzeit nur linear mit der Größe des Eingabegraphen zu wachsen. Dies macht Hoffnung, dass mit diesem Algorithmus auch sehr große Graphen in annehmbarer Zeit verarbeitet werden können.

Da der Centroid-Layouter seine Abstoßungsberechnungen auf einer Approximation des Eingabegraphen durchführt, anstatt wie der Fruchterman-Reingold-Layouter mit allen nahe liegenden Knoten, war die gewonnene Geschwindigkeitssteigerung durchaus zu erwarten. Allerdings stellt sich hierbei die Frage, ob und wie sehr sich die Nutzung einer Approximation auf die Qualität der erzeugten

Layouts auswirkt. Um dies herauszufinden, wird in Abbildung 21 die Qualität der erzeugten Layout des Centroid-Layouters und des Fruchterman-Reingold-Layouters anhand der Anzahl der Kantenkreuzungen verglichen.

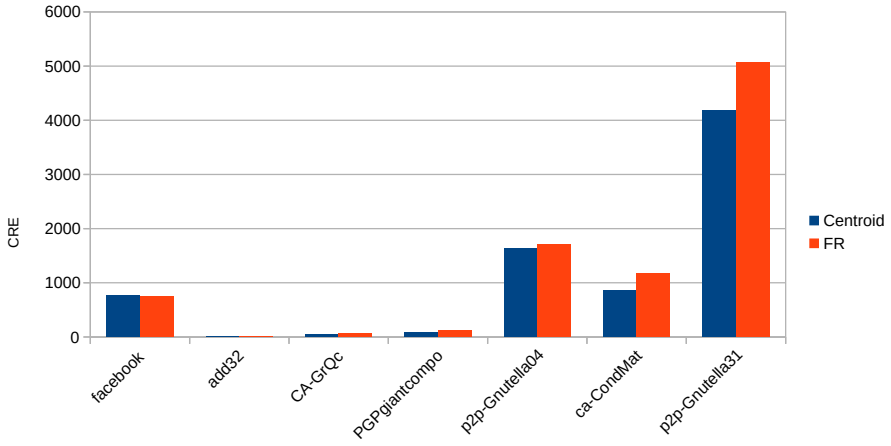


Abbildung 21: Layoutqualität Centroid-Layouter

Das Ergebnis dieses Vergleichs ist durchaus unerwartet. Der Centroid-Layouter liefert beinahe durchgehend rechnerisch bessere Layouts als der Fruchterman-Reingold-Layouter. Dies ist insofern erstaunlich, da eigentlich zu erwarten wäre, dass der Geschwindigkeitsgewinn mit einer Verschlechterung der Layoutqualität einher geht.

Da bezüglich der Layoutqualität Erwartungen und Messergebnisse derart weit auseinander liegen, bietet es sich an, einen händischen Vergleich der generierten Layouts durchzuführen um so eventuell die Ursache für diese Diskrepanz ausfindig zu machen. Bei der Betrachtung der Layouts und dem Vergleich mit den vom Fruchterman-Reingold-Layouter generierten Layouts fielen für einige Datensätze erhebliche Unterschiede auf. Einige dieser Layouts sind in Abbildung 22 zu sehen.

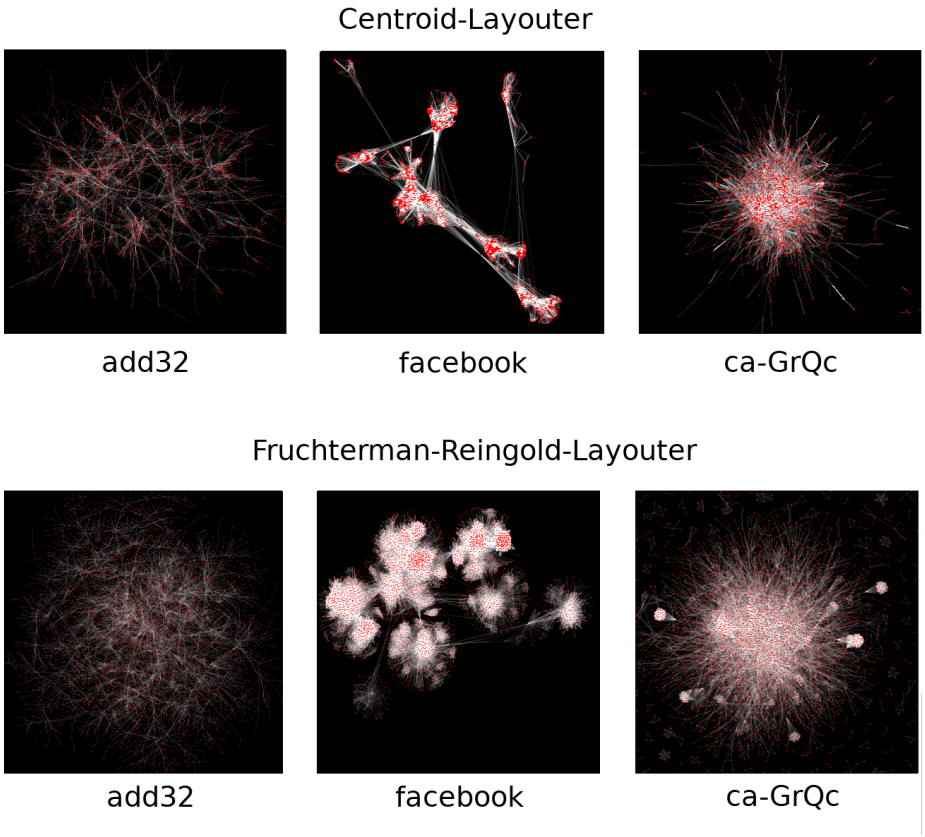


Abbildung 22: Layouts Centroid-Layouter/FR-Layouter

Beim direkten Vergleich fallen bei allen drei gezeigten Datensätzen ganz bestimmte Unterschiede auf. Bei add32 konzentrieren sich beim Centroid-Layouter die meisten Knoten und Kanten auf einigen wenigen "Hauptbahnen". Beim FR-Layouter hingegen sind die Knoten weitgehend gleichmäßig verteilt und abseits der (hier ebenfalls erkennbaren) Hauptbahnen sind kleine Details wie einzelne Verzweigungen zu erkennen, die beim Centroid-Layouter von der Hauptbahn verdeckt werden. Beim Facebook-Datensatz sehen die Layouts der beiden Algorithmen vollständig verschieden aus. Beim Centroid-Layouter sind die einzelnen Cluster erheblich kleiner, weisen kaum eine erkennbare interne Struktur auf und sind viel weiter voneinander entfernt. Beim Datensatz ca-GrQc verschwinden die Details am Rande des Hauptclusters beim Centroid-Layouter fast vollständig. Auch sind die kleinen, beim FR-Layouter deutlich erkennbaren Untercluster nicht mehr zu erkennen. Besonders stark ist außerdem die Auswirkung auf die kleineren Teile des Graphen, die nicht mit dem Hauptteil verbunden sind. Während beim FR-Layouter jeder noch so kleine Teil des Graphen eine Form erhält, die seiner Struktur entspricht, kollabieren die kleinen Teilgraphen beim Centroid-Layouter

zu einzelnen Punkten, sind kaum noch zu erkennen und entfernen sich im Laufe des Layoutings immer weiter vom Hauptteil, bis sie alle vollständig an den Rand des Layout-Bereichs gedrückt wurden.

All diese Unterschiede sind darauf zurückzuführen, dass beim Centroid-Layouter Abstoßungskräfte nicht zwischen einzelnen Knoten sondern nur zwischen Knoten und Clusterzentren berechnet werden. Sämtliche Bereiche des Graphen, die kein eigenes Clusterzentrum enthalten, werden deshalb von den sie umgebenden Abstoßungskräften und den durch ihre Kanten erzeugten Anziehungskräften zu einzelnen Punkten oder Linien zusammengedrückt. Dadurch gehen eine große Menge an Details im Layout verloren. Interessanterweise ist es genau dieses Zusammendrücken zu einzelnen Punkten und Linien, welches dafür sorgt, dass das fertige Layout vergleichsweise wenige Kanten-Kreuzungen enthält und damit rein rechnerisch als gutes Layout betrachtet wird. Dabei werden Überlappungen von Kanten bereits als Kreuzungen mitgezählt. Allerdings zählt es rechnerisch nur als Überlappung, wenn die Kanten exakt übereinander liegen. Um die optische Qualität zu mindern, reicht es aber bereits, wenn die Kanten sehr dicht beieinander liegen.

Obwohl die Layouts nach objektiven Maßstäben als gut zu betrachten sind, sind sie subjektiv weniger gut als die Layouts des Fruchterman-Reingold-Algorithmus. Außerdem ist das Hauptziel des Layoutens und Visualisierens von Graphen der Gewinn von Informationen durch Betrachten einer geeigneten Darstellung des Graphen. Der Mangel an Details in den vom Centroid-Layouter generierten Layouts macht diese weniger informativ, wodurch sie weniger zur Erfüllung des eigentlichen Hauptziels beitragen und damit auch implizit weniger gut sind. Allgemein lässt sich hieraus schließen, dass bei der Bewertung der Qualität eines Graph-Layouts nicht allein Metriken herangezogen werden können und eine händische Beurteilung in jedem Fall sinnvoll ist.

Obwohl die erstellten Layouts aus subjektiver Sicht nicht optimal sind, sind sie aus rechnerischer Sicht qualitativ gut und die Geschwindigkeit, mit der diese Layouts errechnet werden ist erstaunlich gut. Zum schnellen Erstellen eines groben Layouts um einen ersten Überblick über einen Graphen zu erhalten, ist dieser Algorithmus durchaus geeignet.

Ein direkter Vergleich mit der ursprünglichen Implementierung von Auber und Hinge [22] ist leider nicht möglich, da diese, die für ihre Evaluierung benutzten Datensätze, nicht veröffentlicht haben.

### 5.3.5 GiLa-Layouter

Da der GiLa-Algorithmus bei der Evaluierung durch seine Autoren [2] gute Werte lieferte und für die Evaluierung die selben Datensätze verwendet wurden, wie sie auch hier verwendet werden, versprach der Test der GiLa-Implementierung besonders interessant zu werden. Wie üblich wurde die Implementierung auf dem Cluster bei verschiedenen Parallelitäten und mit verschiedenen Datensätzen

getestet und die Laufzeiten für je 50 Iterationen gemessen. Außerdem wurde GiLa mit k=2 und k=3 getestet. (GiLa berechnet Abstoßungskräfte nur innerhalb der k-Nachbarschaft eines Knotens).

Bereits während der Durchführung der Messungen traten massive Probleme auf. Für einige Datensätze terminierte der Algorithmus nicht und musste (nach angemessener Wartezeit) abgebrochen werden. Interessanterweise trat dies schon direkt beim kleinsten Datensatz, dem Facebook-Datensatz, auf. Das dürfte an der großen Kantenanzahl und dichten Vernetzung dieses Graphen liegen, mit der GiLa prinzipbedingt Schwierigkeiten bekommt. Bei einem derart dicht verbundenen Graphen liegen bereits innerhalb der 2-Nachbarschaft enorm viele Knoten, zwischen denen dann Abstoßungskräfte berechnet werden müssen, so dass die Laufzeit des Algorithmus regelrecht explodiert. Für die größeren Graphen (ab einschließlich ca-CondMat) lieferten die kleineren Parallelitäten ebenfalls keine Ergebnisse. Die Laufzeiten für die Durchläufe, die innerhalb einer angemessenen Zeitspanne terminierten, sind in Abbildung 23 zu sehen.

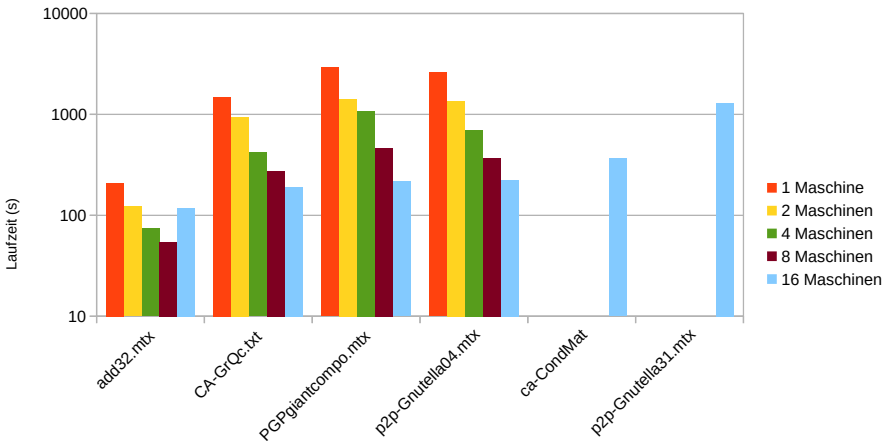


Abbildung 23: Skalierbarkeit GiLa-Layouter k=2(Cluster)

Für den kleinsten hier testbaren Datensatz (add32) liefern alle Parallelitäten schnell Ergebnisse. Da dieser Graph vergleichsweise wenige Kanten und einen hohen Durchmesser aufweist funktioniert die Strategie mit der k-Nachbarschaft hier gut und die Laufzeiten sind gering. Die absoluten Laufzeiten liegen stets deutlich über denen des Fruchterman-Reingold-Layouters (Abbildung 16). Lediglich bei maximaler Parallelität steigen die Laufzeiten mit steigender Graph-Größe nur vergleichsweise moderat an. Zumindest bezüglich der horizontalen Skalierbarkeit schneidet GiLa also wenigstens nicht allzu schlecht ab. Die absolute Laufzeit ist aber auch bei maximaler Parallelität noch sehr hoch. Was für einen starken Einfluss die Anzahl der Kanten im Graph bei GiLa hat, ist daran zu sehen, dass die Laufzeit für den letzten bei geringer Parallelität getesteten Datensatz

(p2p-Gnutella04) geringer ist als für den vorherigen (pGp-giantcompo). Das liegt daran, dass ersterer zwar mehr Knoten aufweist, zweiterer aber ca. 200 Kanten mehr besitzt.

Wird GiLa mit k=3 ausgeführt, ist das Ergebnis noch extremer. Lediglich für die zwei Datensätze mit den wenigsten Kanten (add32 und CA-GrQc) kommt der Algorithmus zu Ergebnissen. Die gemessenen Laufzeiten sind in Abbildung 24 zu sehen.

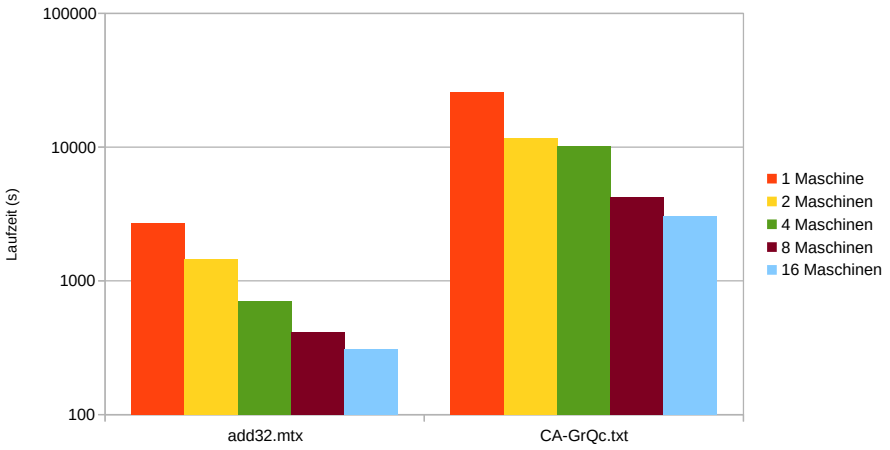


Abbildung 24: Skalierbarkeit GiLa-Layouter k=3(Cluster)

Wie klar zu sehen ist kommt es bei k=3 selbst für diese winzigen Datensätze zu enorm langen Laufzeiten.

Diese Ergebnisse sind durchaus unerwartet, speziell deshalb weil der GiLa-Algorithmus in der ursprünglichen Evaluierung durch seine Autoren ganz erheblich besser abschneidet. Abbildung 25 zeigt die Laufzeiten der originalen GiLa-Implementierung für die Test-Datensätze. Dafür wurde ein Cluster aus 10 Maschinen mit jeweils 4 Prozessorkernen verwendet, was in etwa sieben Rechnern des hiesigen Clusters entspricht.

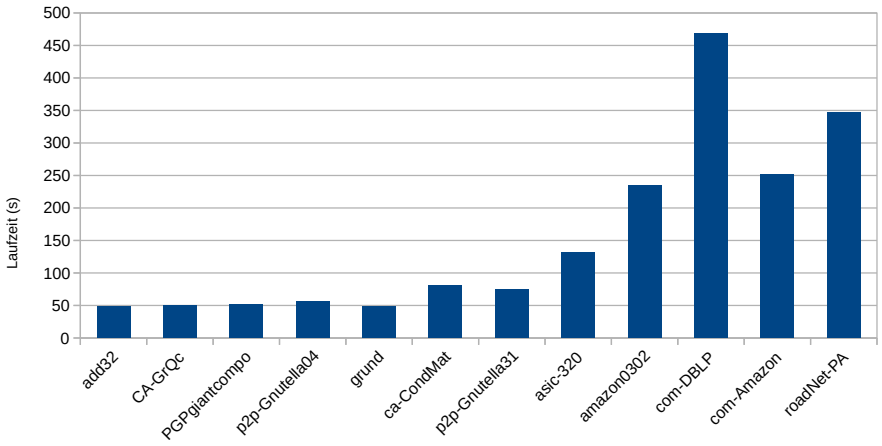


Abbildung 25: Laufzeit GiLa-Referenzimplementierung k=2

Wie unschwer erkennbar ist, bewegen sich die Laufzeiten der Referenzimplementierung von GiLa in ganz anderen Größenordnungen als die hier untersuchte GiLa-Implementierung mittels Flink/Gelly. Leider geben die Autoren des GiLa-Papers nicht an, wie viele Iterationen ihr Algorithmus bei den Testläufen absolviert. Wie in Kapitel 3.3 erläutert, terminiert die originale GiLa-Implementierung anhand eines speziellen Kriteriums automatisch. Um die Laufzeiten also tatsächlich vergleichen zu können, müsste die Zeit verglichen werden, die die Algorithmen benötigen, um ein Layout von festgelegter Qualität zu errechnen. Das Problem hierbei ist, das es praktisch unmöglich scheint, die im GiLa-Paper angegebenen Qualitäten zu erreichen. Abbildung 26 zeigt die im GiLa-Paper angegebene Anzahl an Kantenkreuzungen für die errechneten Layouts der Testdatensätze und vergleicht diese mit den Werten für die GiLa-Implementierung dieser Arbeit, den Fruchterman-Reingold-Layouter und den Werten für zufällige Layouts der Graphen.

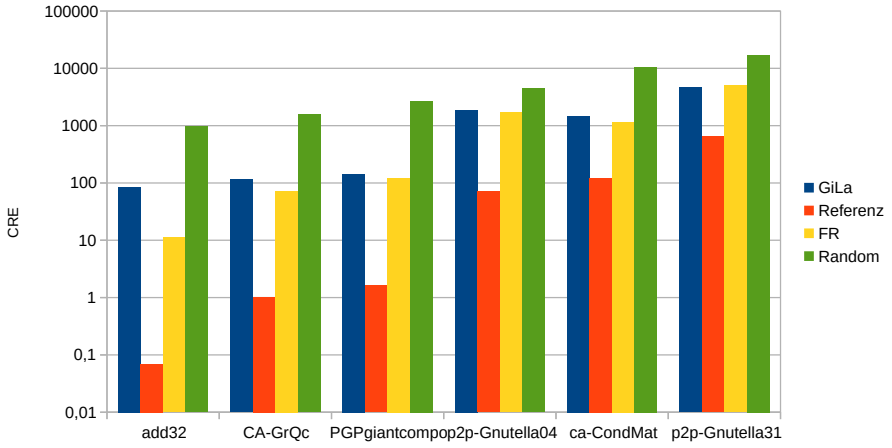


Abbildung 26: Layout-Qualität GiLa-Referenzimplementierung k=2

Die GiLa-Implementierung schneidet für alle Datensätze ähnlich ab wie der Fruchterman-Reingold-Layouter, wobei zweiterer (bis auf den letzten gezeigten Datensatz) durchweg ein wenig besser ist. Die Layouts beider Algorithmen stellen eine signifikante Verbesserung gegenüber den zufälligen Layouts dar. Die GiLa-Referenzimplementierung schneidet allerdings sehr viel besser als die beiden anderen ab. In der Tat schneidet die GiLa-Referenzimplementierung derart gut ab, dass fraglich ist, ob die angegebenen Werte auch nur theoretisch erreichbar sind. Für den add32 Datensatz beispielsweise liefern die verschiedenen Algorithmen die folgenden Werte: GiLa: 85.24, Fruchterman-Reingold: 11.11, GiLa-Referenzimplementierung: 0.07. Die GiLa-Referenzimplementierung liefert also ein um den Faktor 1217 besseres Layout als die Implementierung auf Flink und um ein Faktor 158 besseres als der Fruchterman-Reingold-Layouter. Da beide GiLa-Implementierungen den gleichen Algorithmus benutzen müsste die Qualitätssteigerung pro Iteration ungefähr identisch sein. Da der Fruchterman-Reingold-Algorithmus genauer arbeitet als GiLa, mit seinem k-Nachbarschafts-Ansatz, müsste bei Fruchterman-Reingold die Qualitätssteigerung pro Iteration hier sogar noch höher liegen. Die Qualitätsunterschiede wären also nur damit zu erklären, dass die Referenzimplementierung deutlich mehr Iterationen absolviert hat. Wie in Abbildung 13 zu sehen war, sinkt der Qualitätsgewinn pro zusätzlicher Iteration mit zunehmender Iterationsanzahl sehr stark. Um eine derart niedrige Anzahl an Kantenkreuzungen zu erreichen wie die, die das GiLa-Paper angibt, wäre daher eine gigantische Anzahl an Iterationen nötig, was allerdings nicht zu den angegebenen Laufzeiten zu passen scheint. Auch rein intuitiv erscheinen die Werte unrealistisch. Ein Wert von 0.07 bedeutet, dass sich eine Kante durchschnittlich 0.07 mal mit einer anderen Kante schneidet. Der Datensatz add32 besitzt 9462 Kanten und angesichts des Aussehens des Graphen (siehe beispielsweise Abbildung 22) erscheint ein Wert von 0.07 äußerst unrealistisch.

Die Implementierung von GiLa in Flink/Gelly funktioniert grundsätzlich. Allerdings ist die Implementierung ganz erheblich langsamer als die auf Giraph basierende Referenzimplementierung. Das dürfte unter anderem darauf zurückzuführen sein, dass Gelly schlicht erheblich langsamer ist als Giraph. Speziell die in Gelly fehlende (oder zumindest nicht ausreichend dokumnetierte) Möglichkeit zur effektiven Partitionierung der Knoten über die beteiligten Rechner des Clusters dürfte hier einen großen Einfluss haben. Letztendlich ist die GiLa-Implementierung deutlich langsamer (bis hin zum nicht-terminieren bei bestimmten Datensätzen) als der Fruchterman-Reingold-Layouter und die generierten Layouts sind außerdem beinahe durchgehend schlechter. Aufgrund dessen wurde in dieser Arbeit darauf verzichtet, den GiLa-Ansatz weiter zu verfolgen und beispielsweise Multi-GiLa ([23]) zu implementieren.

### 5.3.6 VertexFusing-Layouter

Die grundlegende Idee hinter dem VertexFusing-Layouter war es, ähnliche Knoten miteinander zu verschmelzen, um die Knotenanzahl im Graphen zu reduzieren, wodurch übersichtlichere Visualisierungen des Graphen erstellt und nebenbei die Geschwindigkeit des Layout-Algorithmus erhöht werden sollte. Neben der in Kapitel 4.2.2.2 vorgestellten Ähnlichkeitsfunktion (die für zwei Knoten einen Ähnlichkeitswert zwischen 0 und 1 angibt) wird außerdem ein Schwellwert benötigt, der angibt, ab welchem Ähnlichkeitswert zwei Knoten miteinander verschmolzen werden dürfen. Die Wahl dieses Schwellwertes hat maßgeblichen Einfluss auf die generierten Layouts und die Geschwindigkeit des Algorithmus. Abbildung 27 zeigt Layouts, die mit verschiedenen Schwellwerten für den Facebook-Datensatz erzeugt wurden. Es wurden jeweils 50 Iterationen absolviert. Um die Vergleichbarkeit zu verbessern, wurde der Algorithmus bei jedem Durchlauf mit dem gleichen zufälligen Startlayout ausgeführt, was dazu führen sollte, dass die grobe Struktur bei allen Layouts in etwa ähnlich bleibt. Da während des Layoutings diverse Zufallsprozesse und auch die An- oder Abwesenheit einzelner Knoten eine Rolle spielen, kann es dennoch zu leichten Abweichungen, wie der Verschiebung einzelner Cluster, zwischen den verschiedenen Layouts kommen.

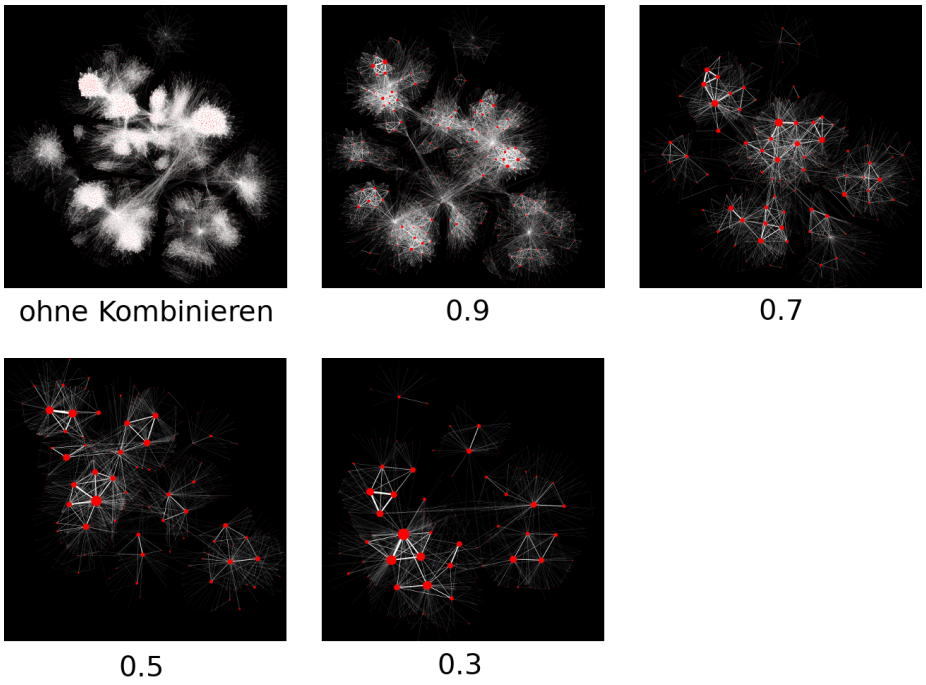


Abbildung 27: Layouts für unterschiedliche Ähnlichkeitsschwellwerte

Bei einem Schwellwert von 0.9 weist das Layout große Ähnlichkeit zum Referenzbild (ohne Kombinieren von Knoten) auf. Alle Cluster des Referenzbildes sind weiterhin zu erkennen und liegen an den selben Stellen. Trotzdem hatte das Verschmelzen der Knoten sichtbare Auswirkungen, obwohl durch den relativ hohen Schwellwert eigentlich nur wenige Knotenpaare zur Verschmelzung in Frage kommen. Es sind deutlich einige Superknoten (Knoten, die aus mehreren Unterknoten bestehen) zu erkennen. In Bereichen des Graphen, die vorher durch die schiere Anzahl von Kanten nur als weiße Flächen erkennbar waren, sind nun einzelne Kanten zu erkennen.

Das Layout zum Schwellwert 0.7 verhält sich ganz ähnlich, die zu beobachtende Vereinfachung des ursprünglichen Graphen ist aber deutlich extremer. Die Anzahl einzelner Knoten und Kanten im Graphen ist sichtbar zurück gegangen, dafür sind nun mehr und größere Superkanten erkennbar. Trotz allem besteht weiterhin eine markante Ähnlichkeit zum Ursprungsgraphen und zu dem Layout für den Schwellwert 0.9. Alle Cluster des ursprünglichen Graphen finden sich auch in dieser stark vereinfachten Version wieder und, auch wenn das aufgrund der geringen Auflösung des Bildes nur schwer zu erkennen ist, auch die Verbindungen zwischen den Clustern entsprechen denen des Ursprungsgraphen.

Bei den Bildern zu den Schwellwerten 0.5 und 0.3 sieht die Situation hingegen anders aus. Eine Ähnlichkeit zum Ursprungsgraphen ist kaum noch gegeben, weshalb diese Bilder keine brauchbaren Darstellungen des Graphen sind. Bei genauerer Betrachtung ist dies nicht weiter verwunderlich. Laut der ursprünglichen Idee dürfen nur ähnliche Knoten miteinander verschmolzen werden, damit das Ergebnis weiterhin ungefähr dem Ursprungsgraphen entspricht. Bei einem Schwellwert von 0.5 oder kleiner werden aber auch Knoten miteinander verschmolzen, die kaum Ähnlichkeit zueinander aufweisen (sofern alle besseren Verschmelzungen bereits ausgeführt wurden). Das führt dazu, dass der Graph auch dann noch weiter vereinfacht wird, wenn diese Vereinfachung nicht länger gerechtfertigt ist und das Ergebnis eine völlig andere Struktur aufweist als der Ursprungsgraph.

Ein Vergleich der Qualität der vereinfachten Layouts, beispielsweise mittels Kantenkreuzungen, ist nicht sinnvoll. Da die Anzahl der Kanten einen ganz erheblichen Einfluss auf die Anzahl der Kantenkreuzungen hat, würden Layouts immer umso besser abschneiden, je stärker vereinfacht sie sind. Auch würde sich die Frage stellen, wie genau mit Superkanten (die ja mehrere vereinte Kanten darstellen) bezüglich Kreuzungen und Überlappungen umzugehen ist. Letztendlich stellt sich hier die Frage, wie ähnlich die vereinfachten Layouts dem Original sind und bei dieser Frage ist die Betrachtung der Kantenkreuzungen nicht sinnvoll.

An dieser Stelle lässt sich bereits festhalten, das die Grundannahme hinter dem VertexFusing-Layouter korrekt ist. Wenn ausreichend ähnliche Knoten zu neuen Knoten verschmolzen werden, lässt sich ein Graph mit weniger Knoten und Kanten erzeugen, dessen Struktur dennoch der des Ursprungsgraphen entspricht. Außerdem ist die Zeichnung des vereinfachten Graphen tatsächlich übersichtlicher, wenn auch natürlich weniger detailliert, als die des Ursprungsgraphen.

Ein weiteres Ziel bei der Entwicklung des VertexFusing-Layouters war es, durch die Vereinfachung des Graphen die Geschwindigkeit des Layoutings zu verbessern. Die Frage dabei ist, ob der durch die Vereinfachung erzielte Geschwindigkeitsgewinn, den Aufwand für die Vereinfachung übersteigt. Um dies zu testen, wurde der Algorithmus wie üblich bei verschiedenen Parallelitäten mit verschiedenen Datensätzen für je 50 Iterationen ausgeführt. Als Schwellwert wurde 0.7 gewählt, da bei diesem Wert in den vorherigen Versuchen eine gute Vereinfachung ohne Beeinträchtigung des Graphen beobachtet wurde. Abbildung 28 zeigt die gemessenen Laufzeiten.

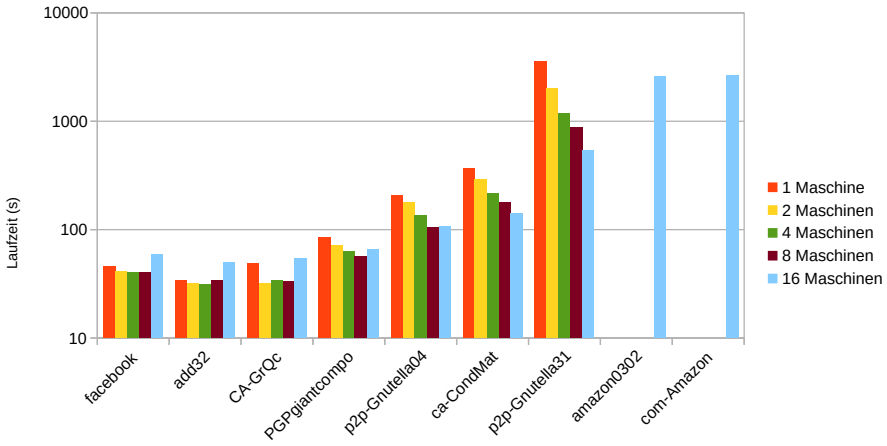


Abbildung 28: Laufzeit VertexFusing-Layouter

Das beobachtete Skalierungsverhalten entsprach weitgehend dem des Fruchterman-Reingold-Layouters (Abbildung 16), weshalb an dieser Stelle auf eine ausführliche Diskussion des Diagramms verzichtet wird.

Viel interessanter ist der relative Geschwindigkeitsgewinn gegenüber dem Fruchterman-Reingold-Layouter. Abbildung 29 zeigt den relativen Geschwindigkeitsgewinn des VertexFusing-Layouters gegenüber dem Fruchterman-Reingold-Layouter in Prozent. Positive Werte stehen für eine Beschleunigung und negative Werte für eine Verlangsamung.

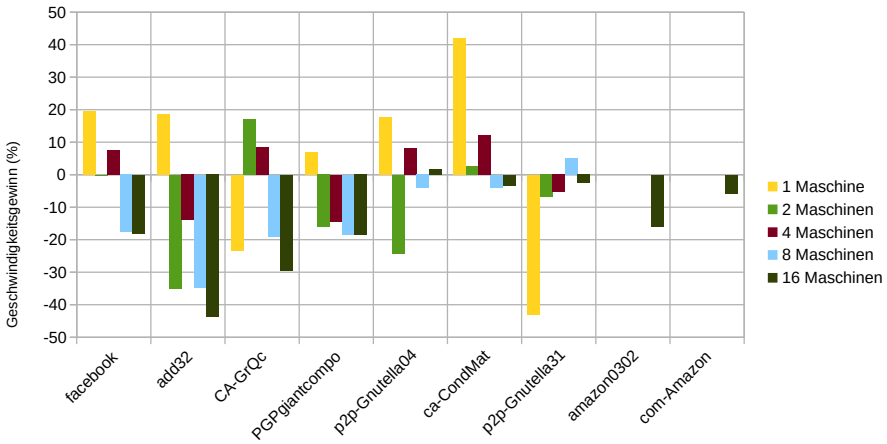


Abbildung 29: Speedup VertexFusing-Layouter

In Abbildung 29 lässt sich kein eindeutiges Muster für das Beschleunigungsverhalten des VertexFusing-Layouters erkennen. Für einige Datensätze wird die Geschwindigkeit um über 40% gesteigert. In anderen Fällen ist der VertexFusing-Layouter mehr als 40% langsamer. Auch zwischen den Extremen schwanken die Werte sehr stark und es ist kein klarer Zusammenhang zwischen Parallelität, Datensatz und Geschwindigkeitssteigerung zu erkennen. Verschiedene Parallelitäten ergeben für den gleichen Datensatz mal eine Verbesserung und mal eine Verschlechterung und bei verschiedenen Datensätzen führen verschiedene Parallelitäten zu verschiedenen Steigerungen.

Dies dürfte verschiedene Gründe haben. Zum einen hängt der Erfolg der Vereinfachung stark vom jeweiligen Graphen ab. Wenn beispielsweise der Graph kaum trennbare Cluster enthält, ist es schwierig, ihn vernünftig zu vereinfachen. Ein weiteres Problem ist, dass für die Vereinfachung des Graphen das Layout schon zu einem gewissen Teil fertig gestellt sein muss. Da die Ähnlichkeiten aus Positionen und einwirkenden Kräften errechnet werden, müssen zwei Knoten bereits relativ nahe beieinander liegen, um verschmolzen werden zu können. Um jedoch nahe zusammen zu liegen müssen sie durch den Layouting-Prozess dorthin bewegt worden sein. Die Vereinfachung und der damit einher gehende Geschwindigkeitsgewinn treten also erst nach einigen Iterationen ein. Bis dahin muss das Layouting auf dem vollständigen Graphen ausgeführt werden und zusätzlich müssen auch die (anfangs noch wenig erfolgreichen) Verschmelzungsversuche durchgeführt werden.

Obwohl vereinzelt durchaus deutliche Geschwindigkeitsgewinne erzielt werden, konnte insgesamt die Erwartung auf einen merklichen Geschwindigkeitsvorteil durch Vereinfachung des zu layoutenden Graphen nicht erfüllt werden. Allerdings ist die Qualität der erzeugten vereinfachten Graphen durchaus beachtenswert.

### 5.3.7 Combi-Layouter

In den bisherigen Tests zeigte der Centroid-Layouter ein gutes Laufzeitverhalten und gute Skalierbarkeit. Aber obwohl die damit generierten Layouts rechnerisch gut sind, sind sie optisch nicht optimal und weisen einen gewissen Mangel an Details auf. Der Fruchterman-Reingold-Layouter dagegen lieferte sehr detailreiche Bilder, benötigte für deren Berechnung aber auch viel Zeit. Daher lag die Idee nahe, den Versuch zu unternehmen, die individuellen Stärken beider Algorithmen zu verbinden. Der Combi-Layouter versucht dies, indem er auf dem Datensatz zuerst einige Iterationen des Centroid-Layouters durchführt und das dabei entstandene grobe Zwischenergebnis mit dem Fruchterman-Reingold-Layouter weiter verfeinert.

Ob sich dadurch tatsächlich ein Geschwindigkeitsvorteil gegenüber dem reinen Fruchterman-Reingold-Layouter erzielen lässt, wurde untersucht, indem der Combi-Layouter bei verschiedenen Parallelitäten auf die verschiedenen Test-Datensätze angewendet wurde und die dabei gemessenen Laufzeiten mit denen

des Fruchterman-Reingold-Layouter verglichen wurden. Die Ergebnisse dieses Vergleiches sind in Abbildung 30 zu sehen. Dort sind für jeden Datensatz und jede Parallelität die prozentuale Zeitersparnis des Combi-Layouters gegenüber dem Fruchterman-Reingold-Layouter aufgetragen. Es wurden je 50 Iterationen absolviert, wovon 45 Iterationen auf den Centroid-Layouter und anschließend 5 Iterationen auf den Fruchterman-Reingold-Layouter entfielen.

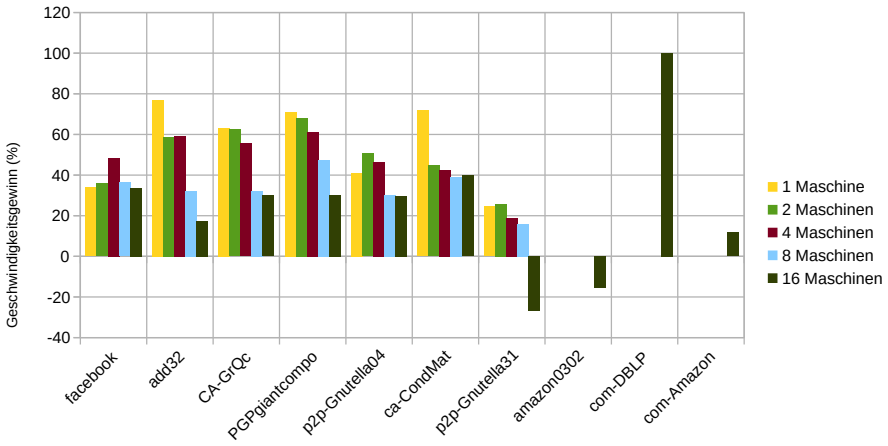


Abbildung 30: Speedup Combi-Layouter

In Abbildung 30 ist deutlich zu erkennen, dass dieser Ansatz in den meisten Fällen einen Geschwindigkeitsvorteil gegenüber dem normalen Fruchterman-Reingold-Layouter bringt. Stellenweise sind Geschwindigkeitssteigerungen von beinahe 80% messbar. Da der Centroid-Layouter, auf den bei dieser Messung der Großteil der Iterationen entfällt, erheblich schneller ist als der Fruchterman-Reingold-Layouter, war ein solches Ergebnis im Prinzip zu erwarten. Interessant ist, dass es anscheinend einen Zusammenhang zwischen Parallelität und der erzielten Geschwindigkeitssteigerung gibt. Bei niedriger Parallelität fällt der Geschwindigkeitsgewinn in der Regel größer aus und für zwei der größeren Datensätze ist der Combi-Layouter bei maximaler Parallelität sogar langsamer als der normale Fruchterman-Reingold-Layouter, wofür jedoch eigentlich keine naheliegende Begründung erkennbar ist. Erfreulich ist aber, das mit diesem Algorithmus der Datensatz com-DBLP erfolgreich verarbeitet werden kann, was mit dem Fruchterman-Reingold-Algorithmus allein nicht möglich war. Der Centroid-Layouter leistet ausreichend Vorarbeit, um dem Fruchterman-Reingold-Layouter die Arbeit so weit zu erleichtern, dass dieser auch bei diesem, für ihn extrem schweren, Datensatz, in annehmbarer Zeit zu Ergebnissen kommt.

Die Steigerung der Geschwindigkeit gegenüber dem Fruchterman-Reingold-Layouter war somit erfolgreich. Nun gilt es zu klären, ob der Combi-Layouter auch qualitativ bessere Layouts liefert als der Centroid-Layouter. Da in Kapi-

tel 5.3.4 bereits festgestellt wurde, dass die Anzahl der Kantenkreuzungen zur Bewertung der Qualität der Layouts des Centroid-Layouters nur eingeschränkt sinnvoll sind, wird hier auf einen Vergleich anhand dieser Metrik verzichtet. Stattdessen wird ein optischer Vergleich durchgeführt.

Abbildung 31 zeigt die während der vorherigen Geschwindigkeitsmessung des Combi-Layouters errechneten Layouts im direkten Vergleich mit den durch den Centroid-Layouter errechneten Layouts. Die vom Fruchterman-Reingold-Layouter errechneten Layouts für diese Datensätze sind in Abbildung 22 zu sehen.

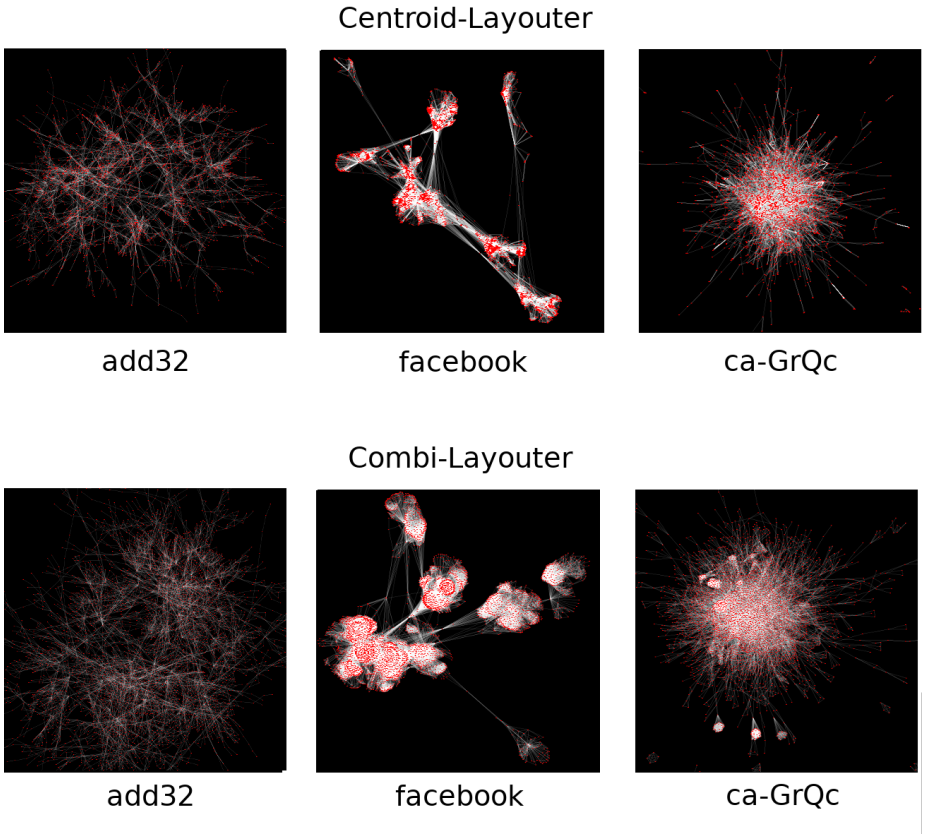


Abbildung 31: Layouts Centroid-Layouter/Combi-Layouter

Bei dem Layout des add32 Datensatzes sind nun die Knoten und Kanten nicht mehr auf wenige Hauptbahnen verteilt, sondern gleichmäßiger verteilt. Speziell an den Außenrändern sind wieder einzelne Verzweigungen und feine Details erkennbar. Das Layout für den Facebook-Datensatz behält den groben Aufbau des vom Centroid-Layouter errechneten Layouts bei. Die einzelnen Cluster sind

weit voneinander entfernt und die Verbindungen zwischen ihnen sind gut sichtbar. Allerdings sind die einzelnen Cluster jetzt intern wieder besser geordnet. Die Cluster und ihre Untercluster sind klar als kreisförmige Objekte erkennbar, während sie beim Centroid-Layouter eher unförmig wirken. Diese Beobachtungen decken sich gut mit den Erwartungen, da ja eine der grundlegenden Ideen beim Centroid-Layouter war, das grobe Layout mittels des Centroid-Layouters zu berechnen und dann vom Fruchterman-Reingold-Layouter die Details ergänzen zu lassen. Auch für den dritten abgebildeten Datensatz sind deutliche Unterschiede erkennbar. Die Ränder sind deutlich detailreicher und zeigen anstatt einzelner, aus mehreren überlappenden Kanten bestehenden, Linien fein aufgegliederte Verzweigungen. Die kleinen stark verbundenen Untercluster sind wieder deutlich erkennbar und auch die kleinen Teile des Graphen die nicht direkt mit dem Hauptteil verbunden sind, sind nicht mehr nur als einzelne Punkte, sondern strukturiert erkennbar. Allerdings haben sich viele dieser kleinen Teile im Zuge der Iterationen des Centroid-Layouters recht weit vom Hauptgraphen entfernt und liegen nun außerhalb des hier abgebildeten Bildausschnittes.

Selbst die wenigen verwendeten Fruchterman-Reingold-Iterationen machen einen merklichen Unterschied aus. Insofern ist also auch das Ziel, die Qualität gegenüber dem Centroid-Layouter zu verbessern, erreicht worden.

Der Combi-Layouter ermöglicht es, einen Kompromiss zwischen Geschwindigkeit und Layoutqualität zu erreichen. Durch das Verhältnis zwischen den Iterationsanzahlen der verwendeten Algorithmen lässt sich beinahe stufenlos zwischen Geschwindigkeit und Qualität wählen, wobei allerdings schon eine sehr geringe Anzahl an Fruchterman-Reingold-Iterationen ausreicht, um die Qualität der erzeugten Layout merklich zu verbessern.

## 5.4 Vergleich der Algorithmen Untereinander

Nachdem alle Algorithmen nun einzeln evaluiert wurden, sollen diese nun abschließend in Geschwindigkeit, Qualität und Skalierbarkeit miteinander verglichen werden.

Ein Vergleich der Laufzeiten ist nicht ganz einfach, da sich diese für jeden Algorithmus je nach genutzter Parallelität und verarbeitetem Datensatz stark unterscheiden können. Ein Vergleich aller Algorithmen bei allen Parallelitäten über alle Datensätze würde äußerst unübersichtlich werden, weshalb für den Vergleich der Laufzeiten der Faktor Parallelität eliminiert werden soll. Anhand der bisher gesammelten Testergebnisse wäre es zumindest prinzipiell möglich, für jeden Algorithmus eine geeignete Parallelität für einen bestimmten Datensatz grob abzuschätzen, indem verschiedene Kennzahlen für den Datensatz (Anzahl Knoten/Kanten, Durchmesser, Dichte etc.) betrachtet werden. Wenn so jeder Algorithmus für jeden Datensatz mit der jeweils optimalen Parallelität ausgeführt wird, müsste jeder Algorithmus seine maximal mögliche Leistung erzielen. Basierend auf dieser Annahme werden in Abbildung 32 die jeweiligen Bestzeiten der

verschiedenen Algorithmen für die verschiedenen Datensätze gezeigt (wie üblich 50 Iterationen). Um die stark unterschiedlichen Laufzeiten in einem einzelnen Diagramm darstellen zu können, war es nötig, die Y-Achse logarithmisch zu skalieren. Dies muss beim Bewerten der Unterschiede zwischen den Algorithmen unbedingt berücksichtigt werden.

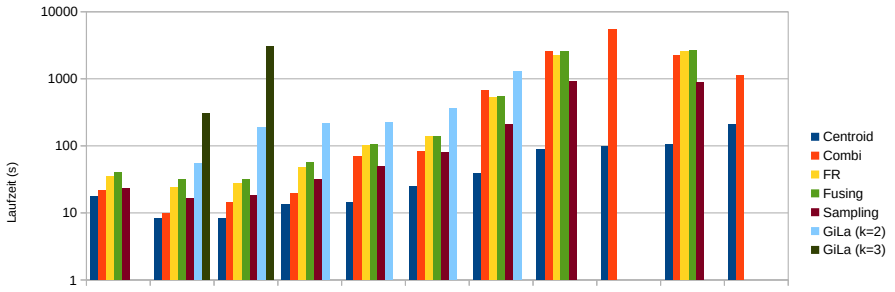


Abbildung 32: Laufzeiten bei optimaler Parallelität (Cluster)

Abbildung 32 zeigt recht anschaulich die Geschwindigkeitsunterschiede für die verschiedenen Algorithmen. Weit abgeschlagen liegt die GiLa-Implementierung mit k=3, bei der auch für kleine Graphen erstaunlich lange Laufzeiten anfallen und nur wenige Datensätze überhaupt bewältigt werden konnten. GiLa mit k=2 ist etwas schneller und bewältigt einige Datensätze mehr, ist aber dennoch erheblich langsamer als die anderen Algorithmen.

Der Fruchterman-Reingold-Layouter liegt im Hinblick auf die Geschwindigkeit im Mittelfeld. Die gemessenen Laufzeiten sind nicht sehr gut, aber akzeptabel. Auch große Datensätze können verarbeitet werden, benötigen aber entsprechend mehr Laufzeit.

Der VertexFusing-Layouter befindet sich meist etwa auf dem gleichen Niveau wie der Fruchterman-Reingold-Layouter, ist aber in der Regel etwas langsamer. Wie in Kapitel 5.3.6 bereits erwähnt, konnte der VertexFusing-Layouter leider nicht die erhofften Geschwindigkeitsvorteile liefern.

Der Combi-Layouter ist für kleine und mittlere Datensätze merklich schneller als der Fruchterman-Reingold-Layouter. Bei großen Datensätzen nähern sich die Laufzeiten an und in bestimmten Fällen ist der Combi-Layouter sogar langsamer. Allerdings schafft er es, den Datensatz com-DBLP in angemessener Zeit zu verarbeiten, was den anderen Algorithmen (ausgenommen den Centroid-Layouter) nicht gelang.

Der Sampling-Layouter erfüllt die auf ihn gesetzten Erwartungen und ist konstant merklich schneller als der Fruchterman-Reingold-Layouter.

Wie erwartet, ist der Centroid-Layouter in sämtlichen Fällen erheblich schneller als die anderen Algorithmen, was zweifellos auf die bei ihm deutlich einfachere Abstoßungsberechnung zurückzuführen ist. Selbst für die größten Datensätze zeigt er nur einen moderaten Anstieg der Laufzeit.

Das zweite Vergleichskriterium neben der Laufzeit ist die Qualität der erzeugten Layouts. Da die Qualität unabhängig von der Parallelität ist, ist der Vergleich zwischen den Algorithmen deutlich simpler als bei der Laufzeit. Abbildung 33 zeigt die Qualität der, von den verschiedenen Algorithmen berechneten, Layouts für die Testdatensätze, nach je 50 Iterationen. Wie gewohnt, wird die Qualität anhand der durchschnittlichen Kantenkreuzungen pro Kante bewertet.

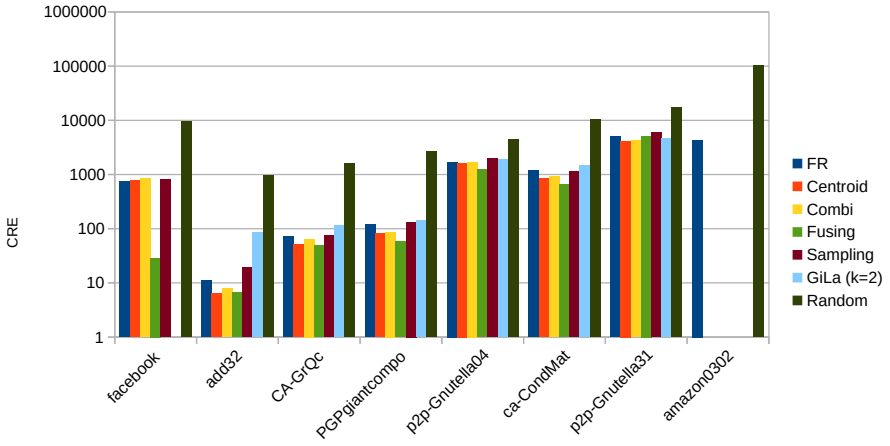


Abbildung 33: Layoutqualität der Algorithmen

Die durch die verschiedenen Algorithmen erzeugten Layouts sind sich bezüglich der Qualität relativ ähnlich. Alle Algorithmen liefern deutlich bessere Ergebnisse als eine zufällige Anordnung der Knoten. Wie viel besser die errechneten Layout gegenüber zufälligen Layouts sind, hängt jedoch stark vom jeweiligen Datensatz ab. So funktioniert das Layouting beim Datensatz add32 beispielsweise besonders gut. Hier besitzt bei einem zufälligen Layout jede Kante ca. 1000 Kreuzungen mit anderen Kanten. Nach dem Layouting sind es weniger als 10. Beim Datensatz p2p-Gnutella04 hingegen funktioniert es weniger gut. Hier wird die Anzahl der Kantenkreuzungen gerade einmal um Faktor 2,5 reduziert. Letztendlich hängt es von der Struktur des Graphen ab, ob sich dieser mit wenigen Kantenkreuzungen zeichnen lässt oder nicht. Besonders der Durchmesser scheint hier eine entscheidende Rolle zu spielen. Der Datensatz p2p-Gnutella04 hat den niedrigsten Durchmesser und lässt sich schlecht layouten, der Datensatz add32 dagegen hat einen der höchsten Durchmesser und lässt sich besonders gut layouten. Zwischen den Ergebnissen der einzelnen Algorithmen gibt es durchaus Unterschiede.

Der VertexFusing-Layouter schneidet häufig am Besten ab, was aber zu erwarten war, da im Zuge der Vereinfachung des Graphen die Anzahl an Kanten sinkt und so die Anzahl an Kreuzungen abnimmt. Je besser sich die Knoten des Graphen gruppieren lassen, desto weniger Kanten und damit Kantenkreuzungen existieren. Die Knoten des facebook-Datensatzes lassen sich besonders gut gruppieren, weshalb der VertexFusing-Layouter dort erheblich bessere Werte liefert als die anderen Algorithmen.

GiLa schneidet meist eher schlecht ab, was daran liegen dürfte, dass die Beschränkung auf die k-Nachbarschaft bei den Abstoßungskräften eine zu grobe Vereinfachung darstellt.

Wie bereits in Kapitel 5.3.4 erläutert, erzeugt der Centroid-Layouter immer rechnerisch vergleichsweise gute Layouts, was aber in diesem speziellen Fall nicht zwingend bedeutet, das diese Layouts auch aus subjektiver Sicht gut sind.

Erfreulich ist, dass sowohl der Sampling-Layouter als auch der Combi-Layouter ähnlich abschneiden wie der Fruchterman-Reingold-Layouter. Die Bemühungen, die Geschwindigkeit dieser beiden Algorithmen gegenüber dem Fruchterman-Reingold-Layouter zu steigern, hat sich nicht negativ auf die Qualität der Layouts ausgewirkt.

Abschließend wird die Skalierbarkeit der Algorithmen wird mithilfe des Speedups verglichen. Der Speedup gibt Auskunft darüber, wie stark eine vertielte Ausführung eine Berechnung beschleunigt. Der Speedup $s$ für eine gegebene Parallelität $n$ ist definiert als $s(n) = \frac{laufzeit(1)}{laufzeit(n)}$. Dabei steht $laufzeit(n)$ für die Zeit, die für die Berechnung benötigt wird, wenn $x$ Rechner parallel arbeiten. Abbildung 34 zeigt die Speedup-Diagramme für die getesteten Algorithmen.

Bei der Betrachtung der einzelnen Diagramme fällt auf, dass diese sich zum Teil erheblich unterscheiden. Es ist klar erkennbar, dass der bearbeitete Datensatz bei jedem Algorithmus einen großen Einfluss auf den erzielten Speedup hat. Die Größe des Datensatzes (im Bezug auf Knoten und Kanten) scheint eine große Rolle zu spielen. So erreicht der größte hier untersuchte Datensatz, p2p-Gnutella31, nahezu immer die besten Werte. Nur beim Fruchterman-Reingold-Layouter erzielt der nächst kleinere Datensatz bessere Werte. Allerdings spielt nicht allein die Größe eine Rolle, da z.B. der kleinste Datensatz, add32, zwar durchgehend vergleichsweise schlechte Werte erzielt, aber nicht immer die Schlechtesten.

Die mit Abstand besten Speedup-Werte erzielt der GiLa-Layouter. Allerdings hat dieser, trotz der guten Skalierbarkeit, insgesamt sehr lange Laufzeiten. Die Laufzeiten für die größten Datensätze sind dabei derart lang, dass deren Layouting abgebrochen werden mussten und so kein Speedup für diese berechnet werden konnte.

Die schlechtesten Speedup-Werte erzielt der Centroid-Layouter. Stellenweise liegt der Speedup bereits für eine Parallelität von 2 unterhalb von 1, was bedeutet, dass die verteilte Ausführung sogar langsamer ist als die nicht-verteilte.

Die restlichen Algorithmen verhalten sich weitgehend ähnlich. Eine Parallelität von 2 ist für viele Datensätze noch nützlich. Höhere Parallelitäten aber nur für größere Datensätze. Sehr hohe Parallelitäten für kleine Datensätze führen dabei sogar immer zu einer Verlangsamung, gegenüber der nicht-verteilten Ausführung.

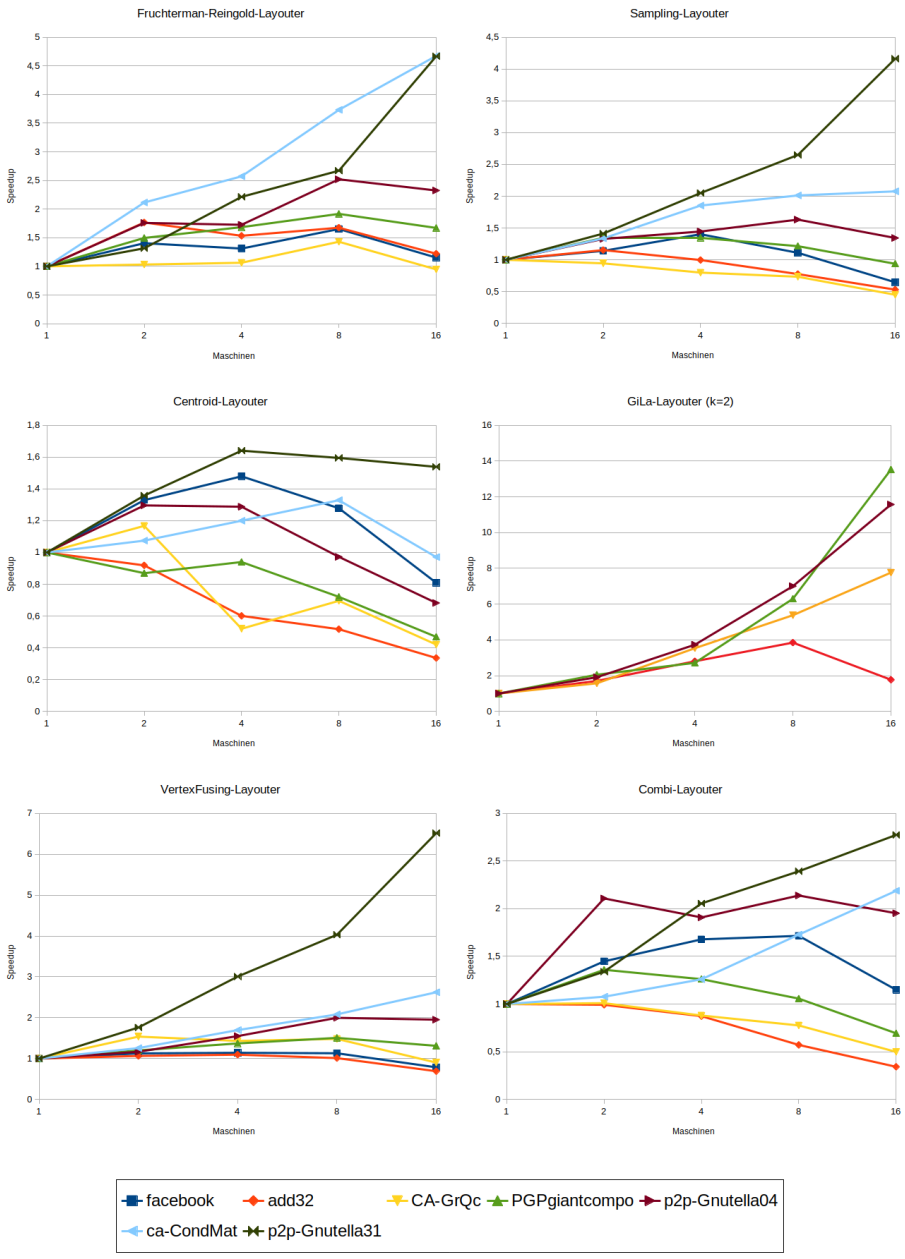


Abbildung 34: Speedup der Algorithmen

# 6 Zusammenfassung, Fazit und Ausblick

Im Laufe dieser Arbeit wurden verschiedene Algorithmen zum Layouten von Graphen als Operatoren für das Gradoop-Framework in Flink implementiert. Neben bestehenden Algorithmen, wie etwa dem Fruchterman-Reingold-Algorithmus wurden auch einige neue Ansätze wie der VertexFusing-Layouter implementiert. Außerdem wurden Metriken zum objektiven Bewerten der erzeugten Layouts implementiert und die Möglichkeit geschaffen, die erzeugten Layouts, mittels Gradoop, verteilt zu zeichnen. Eine besondere Herausforderung bei der Implementierung war die Umsetzung der Algorithmen nur mit den von Apache Flink zur Verfügung gestellten Operatoren. Speziell die Umsetzung des eigentlich für zentralisierte Berechnungen entworfenen Fruchterman-Reingold-Algorithmus und die Umsetzung der Knotengruppierung mittels Ähnlichkeitsfunktion für den VertexFusing-Layouter waren hier besonders herausfordernd.

Alle Implementierungen wurden unter Verwendung diverser Testdatensätze auf einem Rechner-Cluster evaluiert und bezüglich ihrer Laufzeit, Skalierbarkeit und der Qualität ihrer Layouts getestet. Die Ergebnisse dieser Evaluierung sind vielfältig, lassen sich aber wie folgt zusammenfassen:

- Das Erzeugen guter Graph-Layouts mittels verteiltem Layouting mit Gradoop und Flink ist möglich.
- Die Implementierungen skalieren mit Graph-Größe und Anzahl der verwendeten Rechner. Die Effektivität der verteilten Ausführung ist allerdings stark vom verwendeten Datensatz abhängig. Je größer der zu Layoutende Graph ist, desto eher lohnt sich in der Regel die parallele Ausführung des Layoutings.
- Die gemessenen Laufzeiten sind (mit Ausnahme der Laufzeiten des GiLa-Algorithmus) für die hier getesteten Datensätze akzeptabel. Für noch größere Graphen dürfte sich zwar die Effektivität der verteilten Ausführung verbessern, die Laufzeiten würden aber dennoch steigen.
- Algorithmen, die auf anderen Plattformen gute Ergebnisse liefern (GiLa mittels Giraph), liefern nicht zwingend auch auf Apache Flink gute Ergebnisse.
- Die Implementierungen leiden unter den selben Problemen wie zentralisierte Force-directed Layouter. Bestimmte Graphen lassen sich aufgrund ihrer speziellen Struktur (viele dicht beieinander liegende Knoten) nur schwer Layouten.
- Das Layouten sehr großer Graphen ist zwar prinzipiell möglich, oft aber nicht lohnend, da sie meist zu komplex sind, um sie übersichtlich 2-Dimensional darzustellen.
- Mittels Sampling von Knoten lässt sich die Laufzeit des Layoutings reduzieren, ohne die Qualität des Layouts wesentlich zu verschlechtern.
- Der VertexFusing-Layouter ist in der Lage, übersichtliche vereinfachte

Bilder von Graphen zu erzeugen, die für den Betrachter durchaus einen Mehrwert haben können. Die erhoffte Geschwindigkeitssteigerung durch diese Vereinfachung blieb allerdings aus.
- Mit dem Centroid-Layouter lassen sich auch die größten Graphen vergleichsweise schnell und rechnerisch gut layouten. Die damit erzeugten Layouts sind aber unter Umständen aus subjektiver Sicht nicht optimal.
- Der Versuch, die Geschwindigkeit des Centroid-Layouters und die Qualität des Fruchterman-Reingold-Layouters im Combi-Layouter miteinander zu kombinieren, gelang grundsätzlich. Die Kombination der Algorithmen liefert bessere Layouts als der Centroid-Layouter und ist schneller als der Fruchterman-Reingold-Layouter. Der Geschwindigkeitsgewinn war allerdings nicht ganz so groß wie erhofft.

Von hier ausgehend gibt es verschiedenen Richtungen in die diese Arbeit fortgeführt werden kann. Eine hohe Priorität sollte dabei die Erprobung von Multilevel-Techniken, wie beispielsweise Multi-GiLa [23], haben. Allerdings wird hier eine bloße Implementierung von Multi-GiLa nicht ausreichen, da, wie sich gezeigt hat, GiLa auf Flink äußerst langsam ist und Multi-GiLa unter ähnlichen Problemen leiden würde. Überlegenswert wäre hier die Implementierung des verwendeten Solar-Merger-Algorithmus (im Optimalfall ohne Gelly) und die Kombination mit dem Fruchterman-Reingold-Layouter.

Den Ansatz des Combi-Layouters weiter zu verfolgen könnte lohnenswert sein. Bisher ist dieser eine reine Aneinanderreihung der beiden Basisalgorithmen. Eine bessere Abstimmung des Fruchterman-Reingold-Layouters auf die vom Centroid-Layouter geleistete Vorarbeit (z.B. über die Anpassung der Maximaldistanz für die Abstoßungsberechnung) könnte weitere Geschwindigkeitsvorteile erzielen.

Der VertexFusing-Layouter erzielt kaum Geschwindigkeitsvorteile, da bei diesem erst eine gewisse Vorarbeit durch den verwendeten Layouting-Algorithmus geleistet werden muss, bevor die Vereinfachung des Graphen erfolgen kann. Interessant wäre hier, ob diese Vorarbeit auch vom wesentlich schnelleren Centroid-Layouter geleistet werden kann, anstatt - wie bisher - durch den Fruchterman-Reingold-Layouter. Auch könnte das Verfahren zum Gruppieren von Knoten mittels einer Ähnlichkeitsfunktion auch für andere Anwendungen interessant sein, weshalb über eine Implementierung als eigener Gradoop-Operator nachgedacht werden sollte.

Sämtliche hier vorgestellten Algorithmen befassen sich ausschließlich mit dem Layouten einzelner Graphen. Das EPGM sieht aber auch Sammlungen von Graphen mit überlappenden Knoten (Graph-Collections) vor. Ein Algorithmus, der in der Lage ist diese Graph-Collections parallel zu layouten, wäre eine interessante Ergänzung.

Direkt nach Abschluss dieser Arbeit, werden die vorgestellten Implementierungen (oder ein Teil dieser), nach vorheriger Abstimmung mit den Gradoop-Entwicklern, in den Haupt-Entwicklungszweig von Gradoop übernommen. Dieser ist unter `https://github.com/dbs-leipzig/gradoop` zu finden. Im Zuge der Integration

kann es zu kleineren Änderungen an den Implementierungen kommen. Der Stand des Codes, der für die Evaluierungen in dieser Arbeit benutzt wurde, ist unter `https://github.com/dbaumgarten/gradoop/tree/thesis-final` zu finden.

# 7 Literatur


[1] R. Diestel, *Graph Theory*, Bd. 173. 2017.

[2] A. Arleo, W. Didimo, G. Liotta, und F. Montecchiani, „A Distributed Force-Directed Algorithm on Giraph: Design and Experiments“, Juni 2016.

[3] A. Aldabobi und J. Alsakran, „Enhancing the Visual Perception of Graph Communities“, 2017.

[4] P. Carbone *u. a.*, „Apache Flink™: Stream and Batch Processing in a Single Engine“, *IEEE Data Engineering Bulletin*, Bd. 38, Jan. 2015.

[5] T. Rabl, J. Traub, A. Katsifodimos, und V. Markl, „Apache Flink in current research“, *it - Information Technology*, Jan. 2016.

[6] „Flink DataSet Transformations“. https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/dataset_transformations.html.

[7] L. Valiant, „A Bridging Model for Parallel Computation.“, *Commun. ACM*, Bd. 33, S. 103–111, Aug. 1990.

[8] „Apache Giraph“. https://giraph.apache.org/.

[9] „Gelly: Flink Graph API“. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/.

[10] „GRADOOP: Scalable Graph Data Management and Analytics with Hadoop“. https://github.com/dbs-leipzig/gradoop.

[11] M. Rodriguez und P. Neubauer, „Constructions from Dots and Lines“, *Bulletin of the American Society for Information Science and Technology*, Bd. 36, Aug. 2010.

[12] M. Junghanns, A. Petermann, N. Teichmann, K. Gómez, und E. Rahm, „Analyzing Extended Property Graphs with Apache Flink“, 2016.

[13] M. Junghanns, M. Kießling, N. Teichmann, K. Gómez, A. Petermann, und E. Rahm, „Declarative and distributed graph analytics with GRADOOP“, *Proceedings of the VLDB Endowment*, Bd. 11, S. 2006–2009, Aug. 2018.

[14] M. A. Rostami *u. a.*, „BIGGR: Bringing Gradoop to Applications“, *Datenbank-Spektrum*, Bd. 19, Feb. 2019.

[15] M. Junghanns, M. Kießling, A. Averbuch, A. Petermann, und E. Rahm, „Cypher-based Graph Pattern Matching in Gradoop“, 2017.

[16] M. Junghanns, A. Petermann, und E. Rahm, „Distributed Grouping of Property Graphs with Gradoop“, 2017.

[17] S. Kobourov, „Spring Embedders and Force Directed Graph Drawing Algo-

rithms“, Jan. 2012.

[18] T. M. J. Fruchterman und M. Reingold, „Graph Drawing by Force-directed Placement“, Apr. 1997.

[19] H. Gibson, J. Faith, und P. Vickers, „A Survey of Two-Dimensional Graph Layout Techniques for Information Visualisation“, *Information Visualization*, Bd. 12, S. 324–357, Juli 2012.

[20] S. Chae, A. Majumder, und M. Gopi, „HD-GraphViz: Highly distributed graph visualization on tiled displays“, *ACM International Conference Proceeding Series*, Dez. 2012.

[21] A. Tikhonova und K.-L. Ma, „A Scalable Parallel Force-Directed Graph Layout Algorithm.“, 2008, S. 25–32.

[22] D. Auber und A. Hinge, „Distributed Graph Layout with Spark“, 2015.

[23] A. Arleo, W. Didimo, G. Liotta, und F. Montecchiani, „A Distributed Multi-level Force-Directed Algorithm“, *IEEE Transactions on Parallel and Distributed Systems*, Bd. PP, S. 1–1, Sep. 2018.

[24] S. Hachul und M. Jünger, „Large-Graph Layout Algorithms at Work: An Experimental Study“, *J. Graph Algorithms Appl.*, Bd. 11, S. 345–369, Jan. 2007.

[25] J. McQueen, „Some methods for classification and analysis of multivariate observations“, *Computer and Chemistry*, Bd. 4, S. 257–272, Jan. 1967.

[26] C. Martella, D. Logothetis, A. Loukas, und G. Siganos, „Spinner: Scalable Graph Partitioning in the Cloud“, 2017.

[27] I. Mushketyk, „4 Ways to Optimize Your Flink Applications“. https://dzone.com/articles/four-ways-to-optimize-your-flink-applications.

[28] I. Newton und J. Machin, „Mathematical Principles of Natural Philosophy“, 1721.

[29] Universität Florida, „SuitSparse Matrix Collection“. https://sparse.tamu.edu.

[30] „Matrix Market: File Formats“. https://math.nist.gov/MatrixMarket/formats.html.

# 8 Erklärung

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

Ort:_____ Datum:_____ Unterschrift:_____