Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Bachelorarbeit

Distributed Force-Directed Graph Collection Layout

Abstract: Kräftebasierte Layoutverfahren haben sich inzwischen besonders etabliert und finden weitläufige Anwendung. Dennoch existiert bislang wenig Forschung, die sich mit dem Entwerfen von für verteilte Systeme optimierten Layoutalgorithmen beschäftigt. Auch modernere Konzepte wie die Graphmengen des Extended Property Graph Model wurden bislang noch nicht in Fragestellungen die das Layouting von Graphen betreffen mit einbezogen. Im Rahmen dieser Arbeit wird die Frage erörtert, inwiefern sich bestehende kräftebasierte Layoutverfahren auf Graphmengen im Sinne des EPGM anpassen lassen. Darüber hinaus wird untersucht, ob sich inhärente Limitationen klassischer kräftebasierter Layoutverfahren mittels Graphmengen umgehen lassen. Zur Beantwortung dieser Fragen werden zwei Algorithmen mit den Namen *Asymmetric Force Directed Layout* und *Super Vertex Layout* beschrieben sowie jeweils eine Referenzimplementierung als Gradoop Operatoren entwickelt und evaluiert.

Leipzig, September 2020

vorgelegt von
Lucas Schons (3711400)
Studiengang Informatik

Betreuer: M. Sc. Kevin Gomez

Prüfender Hochschullehrer: Prof. Dr. Erhard Rahm
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Aufgrund ihrer Eigenschaft, Zusammenhänge zwischen Objekten der realen Welt zu repräsentieren, sind Graphen hervorragend dafür geeignet, komplexe Daten zu modellieren. Eine besondere Bedeutung kommt ihnen dabei bei der Modellierung von Daten aus Sozialen Netzwerken (*Social Network Analysis*), Business Intelligence Anwendungen oder auch der Bioinformatik zu.

Üblicherweise ist es der erste Schritt umfangreicherer Analysen von Daten, sich einen Überblick über den gegebenen Datensatz zu verschaffen. Neben der Erhebung spezifischer Kennzahlen kommt hier vor allem der Visualisierung eine besondere Bedeutung zu. So kann ein gutes Bild eines Graphen den Analysten helfen, Muster und Eigenschaften zu erkennen und zu interpretieren. Ein intuitives Beispiel hierfür kommt aus dem *Social Network Analysis* Bereich. Hier werden soziale Beziehungen zwischen Menschen mittels hochgradig vernetzten Graphen visualisiert. Besonders ausgeprägte Beziehungsgeflechte lassen sich durch eine relativ geringe Distanz der Knoten zueinander darstellen.

Diese Visualisierung eines Graphen, *Graphzeichnen* genannt, erweist sich in den meisten Fällen als nicht trivial, da die überwiegende Mehrheit der Graphen keine Information über die eigene Position in einem mehrdimensionalen Raum enthält. So wurden in den vergangenen Jahren eine Vielzahl von Verfahren entwickelt, die eine solche Anordnung der Knoten eines Graphen in einem Layoutbereich generieren. Die Positionierung soll dabei gewisse Informationen, die der Struktur des Graphen inhärent sind, erkennbar machen.

Eine populäre Klasse von Layoutverfahren zur Anordnung von Knoten bilden jene Algorithmen, die sich das *Force-Directed Placement* Prinzip zu nutze machen. Grundidee dieser Algorithmen ist, einen Graph als eine Art physikalisches System zu betrachten, in dem Knoten untereinander anziehende und abstoßende Kräfte ausüben. In einem iterativen Prozess werden dann solange Kräfte berechnet und angewendet, bis ein energetisches Minimum erreicht wurde. Solche Verfahren sind deswegen beliebt, da sie intuitiv verständlich und einfach zu implementieren sind. Allerdings sind sie prinzipbedingt sehr rechenintensiv und benötigen mit wachsender Größe der Datensätze eine übermäßige Menge an Rechenzeit und Hauptspeicher. Das kann dazu führen, dass das Layouten von großen Graphen auf einem einzelnen System zu viel Zeit in Anspruch nimmt.

Da im Laufe der vergangenen Jahre im Kontext von *Big Data* ein immer größer werdendes Bedürfnis danach entstand, auch sehr große Datensätze verarbeiten zu können, wurden Softwaresysteme entwickelt, die eine parallele Verarbeitung auf kostengünstig ausbaubaren Shared-Nothing (SN) Clustern möglich machen.

Durch die Arbeit von Baumgarten [7] ist es bereits möglich, mittels Apache Flink kräftebasierte Layouts für einen Graphen zu generieren. Allerdings fokussierte er sich in seiner Arbeit auf ein klassisches Graphmodell. Modelle wie das Extended Property Graph Model (EPGM) definieren allerdings neben dem Graphen als singuläre Einheit noch Graphmengen (*Graph Collecions*). Diese erlauben eine beliebige Un-

terteilung des Ursprungsgraphen in so genannte logische Graphen. Dadurch lassen sich eine Vielzahl von Relationen modellieren, die mit einem einfachen Graphmodell nicht ausgedrückt werden können. Darüber hinaus kodieren auch die einzelne Graphen einer Graphmenge gewissen Informationen, welche sich in der Struktur der Graphmenge widerspiegeln.

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist es zu untersuchen, inwiefern sich bestehende kräftebasierte Layout-Algorithmen an das Konzept der Graphmengen anpassen lassen, um neue und innovative Verfahren zu entwickeln. Die Algorithmen sollen zum einen die einzelnen logischen Graphen einer Graph Collection deutlicher im Layout hervor treten lassen, als auch bestehende Layoutverfahren optimieren.

Hinsichtlich der besonderen Eigenschaften kräftebasierter Layout-Algorithmen soll untersucht werden, inwiefern sich etablierte Algorithmen wie der Fruchterman und Reingold (FR)-Algorithmus modifizieren lassen, um das Layouten von Graphmengen zu ermöglichen. Ausserdem soll erforscht werden, ob sich Graphmengen mittels eines Mulitlevel-Ansatzes layouten lassen. Hierbei steht eine etwaige Verbesserung der Laufzeit im Vergleich zu anderen kräftebasierten Layoutverfahren im Vordergrund.

Alle Implementierungen sollen hinsichtlich ihrer Laufzeit und der Skalierbarkeit bezüglich steigender Graph-Größen und stärkerer Parallelisierung durch Einsatz von einer Vielzahl an Maschinen evaluiert und miteinander verglichen werden.

1.3 Aufbau der Arbeit

Der Aufbau der Arbeit gestaltet sich wie folgt. Im zweiten Kapitel werden einige grundlegende graphentheoretische Konzepte sowie die Funktionsweise von *Distributed In-Memory Dataflow* Systemen vorgestellt. Im dritten Kapitel folgt eine Übersicht verwandter Forschung in den Bereichen Force-Directed Layout und verteiltes Layouting. Schließlich werden vierten Kapitel eigene Force-Directed placement basierte Verfahren zum Layouten von Graphmengen vorgestellt und im fünften Kapitel eine Referenzimplementierung beschreiben. Im sechsten Kapitel wird eine Evaluierung dieser Referenzimplementierung diskutiert und schließlich werden die Ergebnisse der Arbeit im siebten Kapitel zusammengefasst.

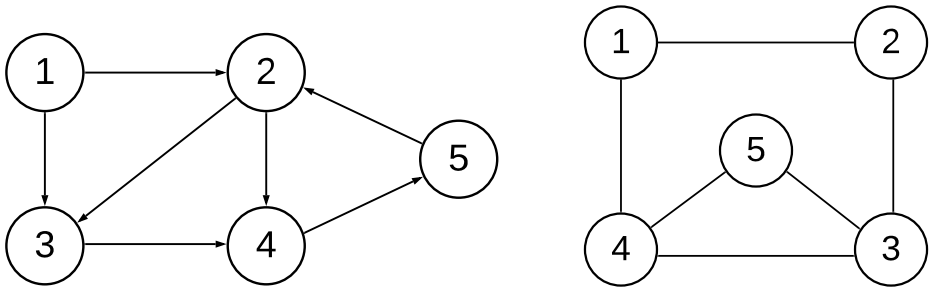
2 Hintergrund

Im folgenden Kapitel werden die für die vorliegende Arbeit relevante Grundlagen und Systeme vorgestellt. Zu Beginn werden essentielle graphentheoretische Konzepte erklärt, daraufhin werden Distributed In-Memory Dataflow Systeme vorgestellt. Abschließend folgt ein Absatz über das verteilte Graphdaten Analyseframework Gradoop.

2.1 Graphentheoretische Grundlagen

2.1.1 Graph

Ein Graph G ist ein 2-Tupel $G = (V, E)$, wobei V eine endliche Menge Knoten und $E \subseteq V^2$ eine endliche Menge Kanten ist. Bei einem *gerichtetem Graph* sind die Kanten $E \subseteq V \times V$ als endliche Menge geordneter Paare definiert. Der erste Knoten des Paars wir Anfangs-, und der zweite Knoten wir Endknoten genannt. Existieren zwischen zwei Knoten $x, y \in V$ mehrere Kanten, so spricht man von Mehrfachkanten. Verlaufen zwei Mehrfachkanten in dieselbe Richtung, so stehen diese parallel zueinander. Ein Graph der über parallele Kanten verfügt heißt Multigraph. Die Kanten ungerichteter Graphen werden häufig als Linien visualisiert, während gerichtete Kanten mit Pfeilen gezeichnet werden, um die Orientierung der Kanten zu markieren. Abbildung 2.1 zeigt jeweils die typische Darstellung eines gerichteten und eines ungerichteten Graphen.



(a) Gerichteter Graph (b) Ungerichteter Graph

Abbildung 2.1: Beispiel Graphen

2.1.2 Graphlayout

Bei einem Graphlayout handelt es sich um eine Funktion $f\colon V \to \mathbb{R}^d$, die jedem Knoten eines Graphen G eine Position in einem d-dimensionalen Layoutbereich zuweist. Im Kontext dieser Arbeit werden ausschließlich 2-dimensionale Layoutbereiche betrachtet. Ziel des Graphzeichnens ist also für einen gegebenen Graphen eine Einbettung im 2-dimensionalen euklidischen Raum zu berechnen [13]. In der resultierenden Darstellung werden die Knoten des Graphen meist durch Kreise oder Quadrate dargestellt.

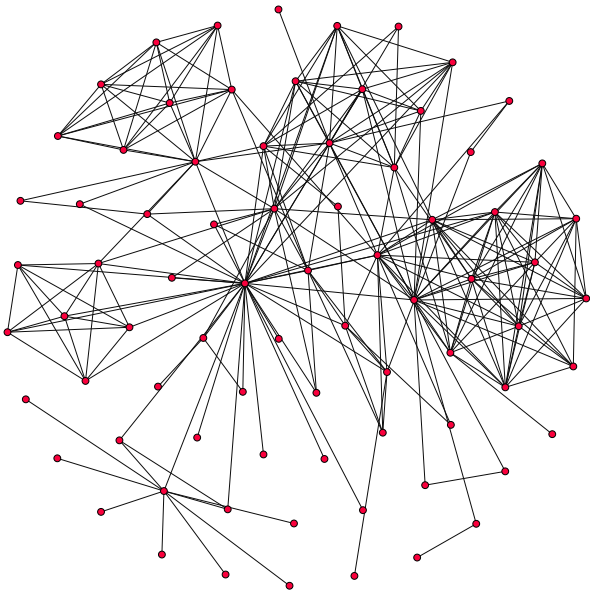


Abbildung 2.2: Beispiel für ein vom FR-Algorithmus erstelltes Layout

Die Kanten des Graphen werden, je nach Anwendungsgebiet und Algorithmus, als Kurven oder Geraden zwischen den Knoten gezeichnet.

Eine der bedeutsamsten Klassen der Layout-Algorithmen ist die der kräftebasierten (engl.: *force directed*) Layout-Algorithmen. Diese Verfahren wenden in einem iterativen Prozess zwischen allen benachbarten Knoten anziehende und zwischen der Grundmenge der Knoten abstoßende Kräfte an. Dieses Verfahren ist vergleichbar mit der Simulation eines physikalischen Systems, in dem die Knoten einer Kante über eine mechanische Feder verbunden sind, während sich alle Knoten wie Elementarteilchen gleicher Ladung abstoßen. Über eine festgelegte Anzahl an Iterationen werden die Knoten so mittels der simulierten Kräfte verschoben, bis das System ein energetisches Minimum erreicht. Diese Layoutverfahren haben eine gewisse Popularität erlangt, da sie in der Regel qualitativ hochwertige Layouts produzieren. Wichtig sind im Allgemeinen eine gleichmäßige Verteilung der Knoten in der zur Verfügung stehenden Fläche sowie möglichst wenig Kantenkreutzungen. Ausserdem sollten eigenständige Strukturen und Symmetrien innerhalb des Graphen auch als solche hervortreten. All diese Faktoren ermöglichen ein übersichtliches und aufschlussreiches Layout. Weitere Gründe für die Popularität der kräftebasierten Layoutalgorithmen sind ihre intuitive Verständlichkeit sowie ihre Anpassbarkeit. Force-Directed Layout-Algorithmen bergen allerdings zwei Nachteile. Zum einen sind diese Verfahren hinsichtlich der Laufzeit sehr aufwendig. Die Berechnung der abstoßenden Kräfte jeder Iteration verlangt die Bildung des Kreuzprodukts der Knoten, was zu einer quadratischen Laufzeit mit Komplexität $O(|V|^2)$ führt. Dabei bezeichnet $|V|$ die Anzahl der Knoten im Graph. Ein zweites Problem ergibt sich aus der Erzeugung des initialen Layouts. Zu Beginn werden die Knoten des Graphen zufällig im Layoutbereich verteilt. Unterschiedliche Initialzustände können zu unterschiedlichen Layouts führen. Das hat zur Folge, dass das Minimum der Energiefunktion im resultierenden Layout nicht garantiert werden kann. Insbesondere mit steigender Knotenanzahl erhöht sich die Differenz in der Qualität des Layouts bei einem globalen Minimum im Vergleich zu einem Layout, welches lediglich einem lokalen Minimum entspricht. Grundlegende Arbeit in dieser Domäne wurde 1984 von Peter Eades [10] und 1991 von Fruchterman und Reingold [11] geleistet. Ein mittels eines kräftebasierten Algorithmus gezeichneter Graph findet sich in Darstellung 2.2. Dieser erfüllt alle zuvor genannten Anforderungen die an ein Layout gestellt werden.

2.1.3 Property Graph Model

Aufbauend auf der mathematischen Definition des Graph existieren verschiedene Graphmodelle, welche genutzt werden um Daten in einer graphenförmigen Struktur zu formatieren. Diese Graphmodelle besitzen eine Vielzahl zusätzlicher Eigenschaften die Notwendig sind, um die Informationen einer Domäne sinnvoll repräsentieren zu können. Ein populäres Modell ist Beispielsweise das Resource Description Framework (RDF) welches vor allem im Kontext des Semantic Web Anwendung findet [19]. Ein RDF Graph besteht im wesentlichen aus Tripeln der Struktur Subjekt-Prädikat-Objekt, wobei es sich bei Subjekt und Objekt um Knoten handelt und das Prädikat die verbindende Kante darstellt. Eine vergleichbar große Rolle wie das RDF im Kontext des Semantic Web spielt das Property Graph Model (PGM) unter den Graphdatenbanken. So wird dieses beispielsweise von Neo4j [20], Titan [23] oder Neptune [1] implementiert oder unterstützt. Im PGM werden Daten als gerichteter Multigraph modelliert. Die Knoten des PGM sind über Zeichenketten (Label) typisiert und verfügen darüber hinaus über eine beliebig große Menge an Attributen, die als Key-Value Paare kodiert sind. Auch die Kanten des PGM sind über Label typisiert und verfügen ebenso wie die Knoten über eine beliebige Menge an Eigenschaften [2].

2.2 Distributed In-Memory Dataflow System

Zur parallelen Verarbeitung der Graphdaten werden im Kontext dieser Arbeit auf *Shared-Nothing* Cluster operierende, verteilte In-Memory Dataflow Systeme (VIDS) betrachtet. Als Shared-Nothing Architektur (SN) wird eine Architektur einzelner Computer bezeichnet, die über ein Netzwerk miteinander Verbunden sind und sich dem Nutzer des Netzwerks als ein einziges System darstellen. Dabei verfügt jeder Computer über einen eigenen Prozessor und eine zugehörige Speicherkomponente. Es findet also kein Zugriff auf gemeinsame Ressourcen statt, sondern jeder Computer arbeitet vielmehr unabhängig und eigenständig [22]. Shared nothing Cluster bieten den Vorteil, mittels handelsüblicher Hardware kostengünstig ausgebaut werden zu können.

Allgemein bauen VIDS auf einem parallelen, datenorientierten Berechnungsmodell auf. Wesentliche Abstraktionen sind hier die *Datasets* und *Transformationen*. Ein Dataset repräsentiert eine typisierte Sammlung von Daten, die über einem Cluster partitioniert ist. Eine Transformation hingegen bezeichnet in diesem Kontext einen deterministischen Operator welcher ein oder zwei Datasets konsumiert und ihre Elemente zu einem neuen Dataset transformiert. Ein Dataflow ist typischerweise eine Aneinanderreihung aus Transformationen, die verteilt auf dem Cluster ausgeführt werden. Um die konkrete Zuweisung des abstrakten Dataflows auf die einzelnen Maschinen des Clusters kümmert sich ein sog. *Scheduler*. Dieser bricht den Dataflow in einzelne Jobs auf, die wiederum als Knoten eines azyklischen, gerichteten Graphen strukturiert werden. Die Knoten dieses Graphen repräsentieren die Threads, während die Kanten die Eingabe-/Ausgabe-Beziehungen zwischen ihnen ausdrücken. Jeder Thread kann gleichzeitig auf einer assoziierten *Dataset Partition* ausgeführt werden.

Transformationen können in Abhängigkeit zu der Anzahl der Eingabe-Datasets in unäre und binäre Transformationen unterschieden werden. Die Tabelle 2.1 zeigt einen Ausschnitt typischer Transformationen die im Kontext dieser Arbeit von Bedeutung sind.

Aktuell sind die bedeutsamsten VIDS Frameworks Apache Flink [**carbone_apache_nodate**] und Apache Spark [6]. Die Implementierung der im Kontext dieser Arbeit entwickelten Algorithmen erfolgt mittels Apache Flink, da dieses VIDS aktuell in Gradoop verwendet wird. Prinzipiell ist aber eine Implementierung mit beliebigen VIDS denkbar, da die Algorithmen ausschließlich auf Datasets und Transformationen aufbauen.

Transformation	Beschreibung
Join	Join Operation auf zwei DataSets
Map	Map-Funktion auf einem Dataset
FlatMap	Flat-Map-Funktion auf einem Dataset
Distinct	Entfernt Duplikate
Group	Gruppiert das DataSet
Filter	Filter nach gewähltem Kriterium

Tabelle 2.1: Ausgewählte Flink Transformationen

2.3 Gradoop

Gradoop [17] (Graph Analytics on Hadoop) ist ein System für die verteilte Analyse von Graphdaten. Es wird unter einer Open Source Software Lizenz (Apache License 2.0) am ScaDS Dresden/Leipzig entwickelt. Gradoop bietet Operatoren zur Integration, Analyse und Repräsentation von Graphdaten. Das System nutzt einen Hadoop Distributed File System (HDFS) Cluster und HBase [5] zur verteilten Speicherung der Daten. Gradoop ist die Referenzimplementation des EPGM und baut zum aktuellen Zeitpunkt auf Apache Flink in der Version 1.7.2 auf. Die Algorithmen, die in dieser Arbeit entworfen wurden, werden als Gradoop-Operatoren zur Verfügung gestellt.

Gradoop ermöglicht die Nutzung einer Vielzahl von Ein- und Ausgabeformaten wie CSV, JSON oder DOT. Es werden verteilte Dateisysteme wie Apache Hadoop [4] oder Apache Accumulo [3], aber auch lokale Dateisysteme, unterstützt. Gradoop baut für die Repräsentation der Graphen auf die *DataSet API* von Flink auf und nutzt Flink Transformationen für die Umsetzung der Operatoren. Dadurch ermöglicht Gradoop die verteilte Ausführung der Berechnungen.

Das zuvor genannte Extended Property Graph Model wird im Folgenden genauer vorgestellt.

2.3.1 Extended Property Graph Model

Das Extended Property Graph Model (EPGM) stellt eine Erweiterung des PGM dar. Zusätzliche zu den bereits genannten Eigenschaften des PGM verfügt das EPGM zusätzlich über das Konzept des logischen Graphen. Logische Graphen referenzieren Knoten und Kanten einer gemeinsamen Grundmenge und verfügen dabei selbst als Key-Value-Paar kodierte Eigenschaften und Label. Die Abbildung 2.3 zeigt eine beispielhafte EPGM-Instanz. Formal ist das EPGM wie folgt definiert: Eine EPGM-Instanz $DB_{EPGM} = (\mathcal{V}, \mathcal{E}, \mathcal{G}, T, \tau, K, A, \kappa)$ ist ein 8-Tupel, welches aus aus einem Knotenraum $\mathcal{V} = (v_i)$, einem Kantenraum $\mathcal{E} = (e_k)$ und einer Menge logischer Graphen $\mathcal{G} = (G_m)$ besteht. Knoten, Kanten und logische Graphen werden anhand ihrer jeweiligen Indizes $i, k, m \in \mathbb{N}$ unterschieden. Darüber hinaus bezeichnet T das Alphabet der Label und $\tau : (\mathcal{V} \cup \mathcal{E} \cup \mathcal{G}) \to T$ eine Zuordnung der Elemente der EPGM-Instanz zu einem Label. Eigenschaften werden als Key-Value Paare dargestellt, wobei K eine Menge an Schlüsseln, A eine Menge an Werten und $\kappa : (\mathcal{V} \cup \mathcal{E} \cup \mathcal{G}) \times K \to A$ eine Zuweisung ist. [16].

Logischer Graph: Ein logischer Graph $G_m = (V, E)$ ist ein 2-Tupel mit $V \subseteq \mathcal{V}$ und $E \subseteq \mathcal{E}$. Die logischen Graphen einer Graphdatenbank können sich überschneiden so das gilt: $\forall G_i, G_j \in \mathcal{G} : |V(G_i) \cap V(G_j)| \geq 0$ und $|E(G_i) \cap E(G_j)| \geq 0$.

Graphmenge: Mehrere logische Graphen lassen sich als Graphmenge zusammenfassen. Eine Graphmenge ist also im folgenden eine Sammlung von logischen Graphen wie sie das EPGM definiert.

Operatoren: Neben dem Graphmodell definiert das EPGM eine Reihe von Operatoren, welche logische Graphen oder Graphmengen konsumieren. Die Tabelle 2.2 zeigt eine Auswahl. Die Operatoren des EPGM ermöglichen die Beantwortung essentieller graphentheoretischer Fragestellungen wie die den Schnitt zweier logischer Graphen oder deren Vereinigung.

Operator	Definition	Beschreibung
Select	$\sigma_{\varphi} : \mathcal{G}^n \to \mathcal{G}^n$	Wählt Elemente der Sammlung per Prädikatsfunktion
Distinct	$\delta : \mathcal{G}^n \to \mathcal{G}^n$	Entfernt Duplikate aus der Sammlung
Sort by	$\xi_{k,o} : \mathcal{G}^n \to \mathcal{G}^n$	Sortiert Sammlung anhand eines Schlüssels
Top	$\beta_n : \mathcal{G}^n \to \mathcal{G}^n$	Gibt erste n Elemente der Sammlung zurück
Union	$\cup : (\mathcal{G}^n)^2 \to \mathcal{G}^n$	Vereinigt zwei Sammlungen
Intersection	$\cap : (\mathcal{G}^n)^2 \to \mathcal{G}^n$	Bildet den Schnitt zweier Sammlungen
Difference	$\setminus : (\mathcal{G}^n)^2 \to \mathcal{G}^n$	Bildet die Differenz zweier Sammlungen
Combination	$\sqcup : \mathcal{G}^2 \to \mathcal{G}$	Kombiniert zwei Graphen
Overlap	$\sqcap : \mathcal{G}^2 \to \mathcal{G}$	Bildet den Schnitt zweier Graphen
Exclusion	$- : \mathcal{G}^2 \to \mathcal{G}$	Bildet die Differenz zweier Graphen

Tabelle 2.2: Auswahl an EPGM Operatoren

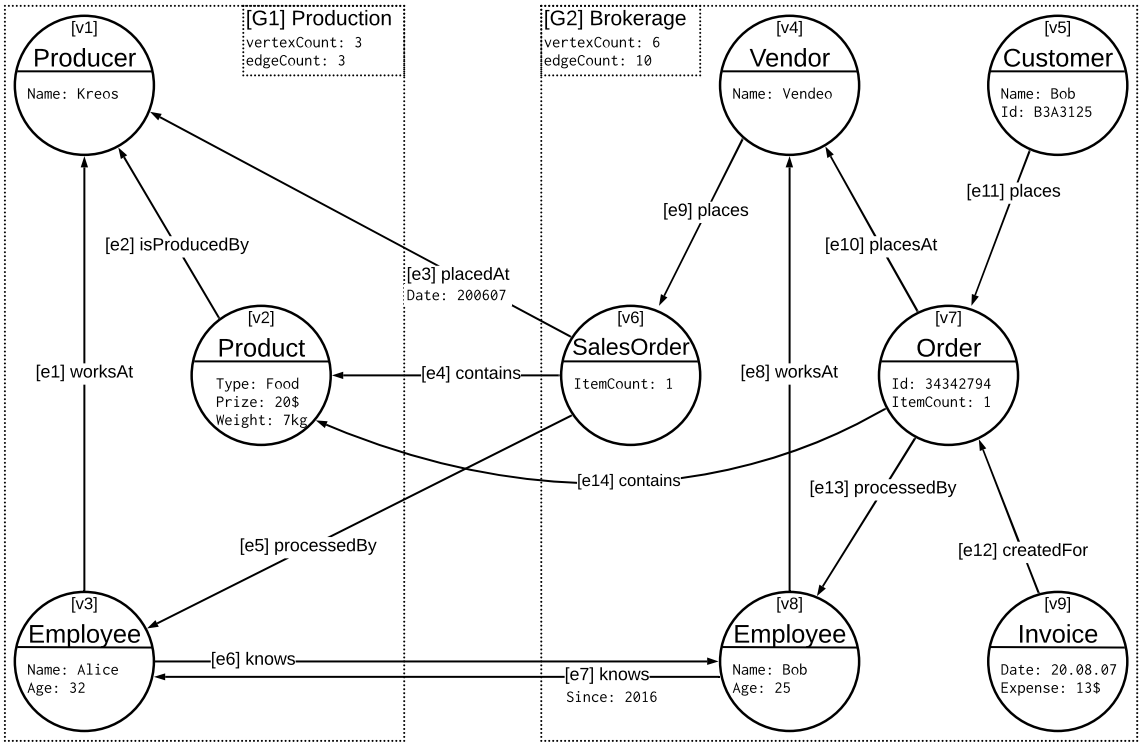


Abbildung 2.3: Beispiel einer EPGM-Instanz

3 Verwandte Forschung

Im folgenden Abschnitt werden einige relevante Forschungsansätze vorgestellt, die für die im Rahmen dieser Arbeit entwickelten Verfahren grundlegend sind. Zunächst wird der von Fruchterman und Reingold entwickelte kräftebasierte Layoutalgorithmus erläutert. Dann folgen darauf aufbauende Arbeiten, die eine optimierung des FR-Algorithmus mittels eines Multilevel-Verfahren erreichen, oder kräftebasierte Layoutalgorithmen für verteilte Systeme entwickeln.

3.1 Fruchterman-Reingold Verfahren

Aufbauend auf das *Spring Embedder Model* von Eades entwickelten Fruchterman und Reingold (FR) 1991 ein kräftebasiertes Layoutverfahren, welches sie in ihrer Veröffentlichung *Graph Drawing by Force-directed Placement* vorstellen [11]. Der FR-Algorithmus war damals der ausgereifteste unter den kräftebasierten Layout-Algorithmen und produzierte hochwertige Layouts. Natürlich treffen die in Abschnitt **??** genannten Probleme auch auf den FR-Algorithmus zu.

Nachfolgend werden die wesentlichen Parameter und Formeln des FR-Algorithmus eingeführt.

Die Formeln für die anziehenden Kräfte (engl.: *attractive forces*) $f_a(v_1, v_2)$ und die abstoßenden Kräfte (engl.: *repulsive forces*) $f_r(v_1, v_2)$ zwischen zwei beliebigen Knoten $v_1, v_2 \in V$ lauten

$$f_a(v_1, v_2) = \begin{cases} 0 & (v_1, v_2) \notin E \\ \frac{d(v_1, v_2)}{k} & \text{sonst} \end{cases} \tag{3.1}$$

$$f_r(v_1, v_2) = \frac{-k^2}{d(v_1, v_2)} \tag{3.2}$$

Die Funktion $d(v_1, v_2)$ berechnet die euklidische Distanz zwischen zwei Punkten. Es gilt also, dass die anziehenden Kräfte mit zunehmender Distanz größer werden, während die abstoßenden Kräfte entsprechend abnehmen. Der Parameter k bezeichnet hierbei die optimale Distanz zweier verbundener Knoten, in der die anziehenden und abstoßenden Kräfte gleich groß sind. Diese optimale Distanz ist für alle zwei Knoten v_1, v_2 einer Kante $e = (v_1, v_2)$ gleich. Sie ist durch die Größe der zur Verfügung stehenden Layoutfläche und der Anzahl der Knoten bestimmt und garantiert so eine gleichmäßige Verteilung der Knoten innerhalb der Fläche. Der Parameter k wird wie folgt bestimmt:

$$k = \sqrt{\frac{\text{Fläche}}{\text{Anzahl der Knoten}}} \tag{3.3}$$

Um zu gewährleisten, dass sich die Summe der Kräfte mit zunehmender Zahl an Iterationen einem globalen Minimum annähern, wendet der FR-Algorithmus eine Temperaturfunktion (engl.: *cooling schedule*) an. Es wird also für jede Iteration ein stetig sinkender Wert, im folgenden als Temperatur benannt, berechnet,

der als Grenzwert für die Kräfte fungiert, die auf einen gegebenen Knoten wirken. Damit soll sichergestellt werden, dass große Verschiebungen nur Anfangs stattfinden können und das Layout zunehmend feiner wird. In dem Beitrag von FR, in dem sie ihr Verfahren vorstellen, wird keine explizite Temperaturfunktion genannt. Abhängig vom Anwendungsfall können hier verschiedene Temperaturfunktionen in Betracht genommen werden. Es gibt Implementierungen die eine lineare Temperaturfunktion [8] verwenden, während andere exponentielle Lösungen anwenden [7].

Zusätzlich beschreiben FR eine optimierte Variante ihres Verfahrens, die sie *Grid-variant* nennen. Hier wird die Anzahl der Knoten, die für die Berechnung der abstoßenden Kräfte miteinander Verglichen werden, reduziert. Diese Reduktion wird erreicht, indem die Layoutfläche gitterförmig in quadratische Zellen mit Seitenlänge $2k$ zerteilt wird. Für einen gegebenen Knoten v werden nun ausschließlich die Knoten aus den acht angrenzenden Zellen und die Knoten aus der selben Zelle für die abstoßenden Kräfte in Betracht gezogen. Hinzu kommt, dass alle Knoten, die weiter als $2k$ von v entfernt sind, nicht in die Berechnung mit einbezogen werden. Das beeinträchtigt die Qualität des Layouts nicht erheblich, da die abstoßenden Kräfte ab dieser Distanz keinen großen Einfluss mehr nehmen.

3.2 Multilevel Ansatz

Ende der 90er wurden die ersten kräftebasierten Layoutverfahren entwickelt die in der Lage waren, Graphen mit weit über 1000 Knoten zu layouten. Diese Verfahren nutzen einen Multilevel Ansatz (auch *Multi-Scale* oder *Multi-Dimensional* genannt) um die Anzahl der für die Berechnung der abstoßenden Kräfte notwendigen Vergleiche zu reduzieren. Kobourov fasst im sechsten Abschnitt des Artikels „Spring Embedders and Force Directed Graph Drawing Algorithms“ diese Entwicklungen hinsichtlich des Layouts von großen Graphen gut zusammen [18].

Grundlegende Arbeit wurde in diesem Bereich von Hadany und Harel geleistet [14]. Diese stellten 1999 einen Algorithmus vor, welcher ausgehend von einem Graph G mehrere, gröbere Abstraktionen des Graphen berechnet. Dabei sollen diese Abstraktionen gewisse topologische Eigenschaften des ursprünglichen Graphen beibehalten. Der Layoutprozess dieses Algorithmus gestaltet sich dann dermaßen, dass im einem iterativen Prozess wechselseitig die Knoten des Graphen und die Knoten der Abstraktion verschoben werden. Dabei beeinflusst, die Positionierung eines Knotens des Graphen die Knoten der Abstraktion und andersherum. Der Layoutprozess wechselt also zwischen diesen beiden Ebenen.

Eine ähnliche Strategie verfolgen auch Gajer et. al [12]. Diese berechnen ausgehend von einem Graphen mehrere, zunehmende gröbere Filtrationen der Knotenmenge. Diese Hierarchie an zunehmend gröberen Graphen $G_0, G_1, ..., G_k$ ist die Grundlage für das Layout. Der gröbste Graph G_k wird initial positioniert, woraufhin das Layout sukzessiv für die feineren Graphen verbessert wird. Dabei bildet das Layout des Graphen $G_i + 1$ die Grundlage für das Layout des Graphen G_i. Durch diese Technik soll das Problem der hohen Laufzeit verbessert werden. Außerdem soll durch die intelligente Platzierung der Knoten das Eintreten von schlechten lokalen Minima verhindert werden. Eine wichtiger Unterschied zum Algorithmus zu Hadany und Harel ist, dass sich die Menge der gefilterten Graphen wesentlich leichter berechnen lässt als die Abstraktionen, die immer gewisse Eigenschaften des Ursprungsgraphen beibehalten müssen.

Hier ist auch die Arbeit von Walshaw [24] zu nennen. Dieser entwickelte ein kräftebasiertes multi-level Layoutverfahren, in dem die Hierarchie aus zunehmende gröberen Graphen durch das verschmelzen von benachbarten Knoten erstellt wurde. Der Layoutprozess selbst gestaltet sich ähnlich wie bei Gajer et. al..

3.3 Verteiltes Layout

Das Berechnen von Graphlayouts auf verteilten Systemen wurde in der bisherigen Forschung wenig betrachtet. Hinge und Auber [15] haben ein Verfahren auf Apache Spark [6] entwickelt um eine kräftebasierte Positionierung der Knoten zu erreichen.

Grundlegende Konzepte des Algorithmus sind zum einen die sog. Schwerpunkte (engl.: *barycenter*) und das Layoutzentrum. Die Schwerpunkte sind hier Strukturen, die jeweils einen Teil des Graphen zusammenfassen sollen. Sie bestehen aus einer Position im Layout, der aktuellen Anzahl an Knoten die mit dem gegebenen Schwerpunkt assoziiert werden und der Distanz zum am weitesten entfernten assoziierten Knoten. Die Knoten des Graphen sind jeweils mit dem Schwerpunkt assoziiert, der ihnen am nächsten ist. Das Layoutzentrum ist als der Durschnitt aller Koordinaten der Knoten definiert.

Die kostenintensive Bildung des Kreuzprodukts der Knotenmenge zur Berechnung der abstoßenden Kräfte wird nun so umgangen, indem nur mehr die Schwerpunkte und das Layoutzentrum abstoßende Kräfte auf die Knoten ausüben. Die Berechung dieser Kräfte wird mittels Broadcasts der Schwerpunkt- und Layoutzentrum Positionen an alle Knoten umgesetzt.

Damit aber die Schwerpunkte und das Layoutzentrum weiterhin den aktuellen Zustand des Layouts wiedergeben, werden sie in jedem Schritt des Algorithmus aktualisiert. Dies geschieht indem die neuen Positionen der Knoten ihrem jeweiligen Schwerpunkt zugewiesen werden. Die Position des Layoutzentrum ergibt sich für jede Iteration aus aus dem Durschnitt der Knotenpositionen. Damit Over- oder Underfitting vermieden wird, löst der Algorithmus Schwerpunkte auf, wenn diese zu spezifisch sind, oder er teilt sie in zwei neue Schwerpunkte auf, sollten zu viele Knoten mit ihnen assoziiert sein. Für die Berechnung der abstoßenden und anziehenden Kräfte werden die Formeln des FR-Verfahren verwendet.

Für die vorliegende Fragestellung ist insbesondere die Arbeit von Baumgarten relevant, in der dieser sich mit der verteilten Berechnung von Graphlayouts mit Hilfe von Gradoop beschäftigt hat. In seiner Arbeit sind sowohl bekannte kräftebasierte Layoutalgorithmen, als auch neue Ansätze in Gradoop mit berücksichtigt worden. Auch die Implementierungen der vorliegenden Arbeit nutzen zum Teil diese Klassen. Allerdings ging es Baumgarten in seiner Arbeit primär um eine Umsetzung kräftebasierter Layoutverfahren auf einem verteilten System. Dementsprechend operieren seine Implementierungen nur auf logischen Graphen, da diese am ehesten der bekannten, mathematischen Definition des Graphen entsprechen. Die Graphmengen des EPGM sind eine Abweichung davon und wurden deswegen von Baumgarten nicht näher betrachtet. Allerdings weist er im Fazit seiner Arbeit bereits darauf hin, dass das parallele Layouten von Graphmengen einen interessante Folgefragestellung darstellen würde.

4 Konzept

Ziel dieser Arbeit ist es, Verfahren zu entwickeln und zu evaluieren, welche ein Layout für eine Graphmenge auf einem VIDS berechnen. Hierbei sollen sich die Verfahren an dem kräftebasierten Ansatz orientieren, und dabei gleichzeitig die spezifischen Möglichkeiten und Anforderungen die sich aus der Anwendung des EPGM ergeben berücksichtigen. Im Folgenden werden Verfahrenskonzepte erläutert, die mittels der Graphmenge eine Optimierung des FR Ansatzes anstreben, und zugleich versuchen den logischen Graphen selbst als Einheit neben Knoten und Kanten im Layout sichtbar zu machen.

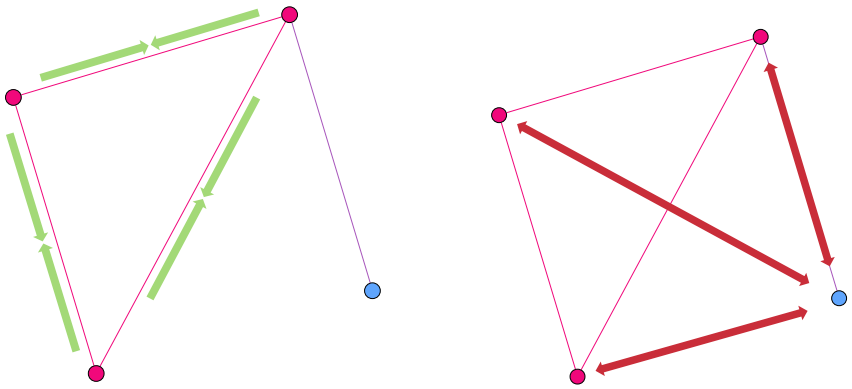
4.1 Asymmetric Force Directed Layout

Asymmetric Force Directed Layout (AFDL) ist eine Anpassung des FR-Algorithmus an das EPGM. FR gingen bei der Beschreibung ihres Layoutverfahrens von einem einfachen Graphmodell aus. Das hat zur Folge, dass bei ihnen die Kräftefunktionen lediglich Knoten und Kanten berücksichtigen. Das EPGM kennt hingegen neben diesen grundlegenden Elementen noch logische Graphen, welche den Ursprungsgraphen in beliebige Untergraphen aufteilen. Da die Information, welche Knoten- bzw. Kantenmenge in welchen logischen Graphen liegen, sich in einem Layout, welches mittels des FR-Algorithmus erstellt worden ist, nicht widerspiegelt, wird nachfolgend ein Verfahren vorgestellt das zum Ziel hat, die logischen Graphen einer Graphmenge im resultierenden Layout als erkennbare Struktur hervortreten zu lassen.

Die zugrunde liegende Annahme ist die, dass sich die Zugehörigkeit zu einem logischen Graphen für einen gegebenen Knoten am besten durch eine, relativ zu den Knoten anderer logischer Graphen, geringere Distanz zu den Knoten aus dem selben logischen Graphen ausdrücken lässt. Das Ziel ist also eine Art Gruppierung der logischen Graphen innerhalb des Layout zu erwirken. So soll es dem Betrachter ermöglicht werden, Rückschlüsse nicht nur auf stärker vernetze Strukturen innerhalb der logischen Graphen zu ziehen, sondern darüber hinaus auch Aussagen über die logischen Graphen selbst treffen zu können. Um die eben genannten Ziele zu erreichen, wird eine Skalierung der anziehenden sowie der abstoßenden Kräfte auf Basis der Graphzugehörigkeit eines gegebenen Knotens vorgenommen. Dementsprechend sollen die anziehenden Kräfte entlang einer Kante höher sein, wenn die benachbarten Knoten Teil desselben logischen Graphen sind. Konträr dazu sollen die abstoßenden Kräfte zwischen zwei Knoten größer sein, wenn diese nicht Teil des selben Graphen sind. Diese asymmetrische Modifizierung der Kräfte soll dann zu einem Layout führen, welches die logischen Graphen erkennbar macht.

Sei $G(v) : V \to G^*$ eine Funktion, die für einen gegebenen Knoten $v \in V$ die Menge der logischen Graphen zurück gibt, in denen v enthalten ist. Die Funktionen zur Berechnung der asymmetrisch modifizierten Kräfte lauten wie folgt:

$$f_{aa}(v_1, v_2) = \begin{cases} c * f_a(v_1, v_2) & \text{falls } G(v_1) \cap G(v_2) \neq \varnothing \\ f_a(v_1, v_2) & \text{sonst} \end{cases} \tag{4.1}$$



(a) Verstärkung der anziehenden Kräfte zwischen Knoten eines logischen Graphens

(b) Verstärkung der abstoßenden Kräfte zwischen Knoten unterschiedlicher logischer Graphen

Abbildung 4.1: Asymmetrische Modifizierung der anziehenden und abstoßenden Kräfte

$$f_{ar}(v_1, v_2) = \begin{cases} c * f_r(v_1, v_2) & \text{falls } G(v_1) \cap G(v_2) = \varnothing \\ f_r(v_1, v_2) & \text{sonst} \end{cases} \tag{4.2}$$

Die Bestimmung der Konstante $c \in \mathbb{N}$ erfolgt hierbei experimentell. Der geeignete Wert kann je nach Größe und Gestalt der Graphmenge variieren.

Die Abbildung 4.1 illustriert die modifizierten Kräfte. Die Knoten der Beispiel Graphmenge sind nach ihrer Graphzugehörigkeit eingefärbt. Es wird deutlich, dass die anziehenden Kräfte die zwischen zwei benachbarten Knoten auftreten nur dann verstärkt werden, wenn diese auch Teil des selben logischen Graphen sind. Konträr dazu werden die abstoßenden Kräfte nur zwischen Knoten unterschiedlicher logischer Graphen, somit zwischen den roten und dem blauen Knoten verstärkt.

Die Qualität des erstellten Layouts hängt wesentlich davon ab, ob sich die vorhandenen logischen Graphen in großen Teilen überlappen oder nicht. Generell gilt, je kleiner die Überlappung, desto deutlicher treten die einzelnen logischen Graphen im Layout hervor. Im Extremfall teilen sich alle vorhandenen logischen Graphen sämtliche Knoten und Kanten, was dazu führen würde, dass die anziehenden Kräfte zwischen allen benachbarten Knoten verstärkt werden, während die abstoßenden Kräfte gleich blieben. Im resultierenden Layout wären die Knoten also wesentlich dichter gelegen als es beim FR-Algorithmus der Fall wäre.

Eine Abschätzung der Laufzeit ist einfach, da die Anzahl der notwendigen Schleifendurchläufe von der Struktur der gegebenen Graphmenge abhängt. Für manche Graphmengen wird, je nach Beschaffenheit, bereits nach wenigen Durchläufen ein akzeptables Layout berechnet worden sein, für andere nicht. Obwohl die Gittervariante des FR-Algorithmus die Laufzeitkomplexität theoretisch von $O(|V|^2 + |E|)$ auf $O(|V| + |E|)$ herab setzt, liegt dieser Betrachtung doch die Annahme zu Grunde, dass die Knoten über alle Zellen des Gitters gleich verteilt sind. Das kann in der Praxis allerdings auch anders sein. Da die Implementierung des AFDL-Algorithmus auf der Gittervariante des FR-Algorithmus aufbaut, gelten für beide die selben laufzeittechnischen Limitierungen. Da sich der AFDL-Algorithmus nur in der Modifizierung der Kräfte nach Vergleich auf Graphzugehörigkeit vom FR-Algorithmus unterscheidet, sind ihre Laufzeitkomplexitäten identisch.

4.2 Super Vertex Layout

Das Super Vertex Layout (SVL) Verfahren stellt einen weiteren Versuch dar, den FR-Algorithmus durch die Ausnutzung der Graphzugehörigkeit zu optimieren. Durch die Berechnung eines Super-Layouts soll der kostspielige Vergleich aller Knoten miteinander zur Berechnung der abstoßenden Kräfte vermieden werden. Der Entwurf dieses Algorithmus ist von den Multilevel-Verfahren inspiriert die im Abschnitt 3.2 vorgestellt worden sind. Im Folgenden wird das Verfahren ausführlicher beschrieben.

4.2.1 Layout des Super Vertex Graph

Man kann im wesentlichen zwei Stufen der Berechnung unterscheiden: zu Beginn wird auf Grundlage der Eingabe-Graphmenge ein neuer Graph erstellt, der eine abstrakte Repräsentation der Graphmenge darstellt. Dieser neue Graph, im folgenden Super Vertex Graph (SVG) genannt, wird dann in der Layoutfläche positioniert. Dieses Layout ist die Grundlage für den nächsten Schritt, in dem die Knoten der Graphmenge um die zuvor positionierten SVG Knoten herum platziert werden.

Da sich die logischen Graphen des EPGM schneiden können, also ein Knoten Teil von mehreren logischen Graphen sein kann, ist es für dieses Verfahren wichtig, diese Mehrdeutigkeit in der Graphzugehörigkeit aufzulösen, und jeden Knoten mit einem primären Graphen zu assoziieren, welcher im folgenden Kopfgraph genannt wird. Generell spielt keine Rolle, welcher Graph genau gewählt wird. Dieses Verfahren wählt den Kopfgraph da dieser am einfachsten zu bestimmen ist. Für einen gegebenen Knoten $v \in \mathcal{V}$ sei der Kopfgraph wie folgt definiert: $f_{G_H} : V \to G \quad f_{G_H}(v) = \text{first}(G(v))$. Die Funktion $G : V \to \mathcal{G} \quad G(v) = \{G | v \in G\}$ gibt die Menge der Graphen von denen v Teil ist zurück. Die Funktion first(G) gibt das erste Element einer aufsteigend nach Id geordneten Menge von logischen Graphen zurück.

Im ersten Schritt wird ein neuer logischer Graph $G_s = \langle V, E \rangle$ aus der gegebenen Graphmenge erstellt. Dieser neue Graph enthält genau einen Knoten für jeden logischen Graph aus der Eingabe-Graphmenge. Seien $v_1, v_2 \in V$ zwei Knoten des SVG, die die beiden logischen Graphen $G_1, G_2 \in \mathcal{G}$ der gegebenen Graphmenge repräsentieren. Es gilt: $\langle v_1, v_2 \rangle \in E$ genau dann wenn mindestens ein Knoten aus G_1 mit einem Knoten aus G_2 über eine Kante verbunden ist.

Für den logischen Graph G_s wird dann ein Layout berechnet. Hierzu wird der FR-Algorithmus verwendet. Die Position eines Knotens aus G_s bezeichnet nun den Mittelpunkt des Layoutbereichs, in dem im nächsten Schritt die Knoten des jeweiligen Graphen positioniert werden sollen.

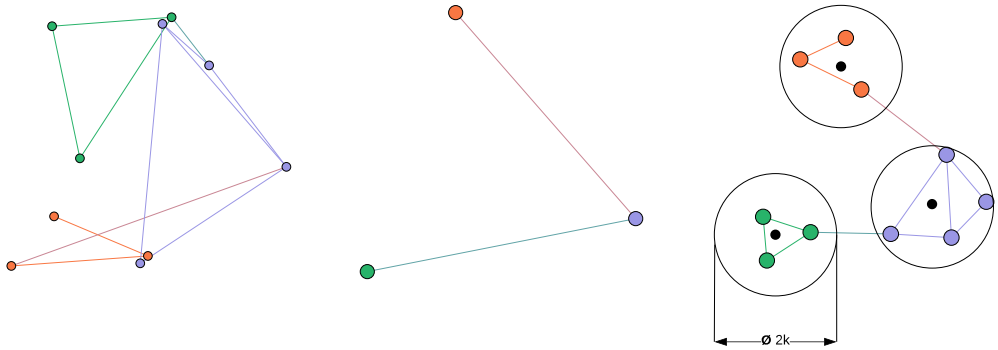
4.2.2 Anordnung der Koten um ihr Zentrum

Im zweiten Schritt werden nun im Bereich um einen solchen Mittelpunkt die Knoten des jeweiligen Graphen angeordnet. Dies erfolgt wiederum mittels FR-Algorithmus. Nun müssen lediglich die Knoten eines Graphen für die Berechnung der anziehenden und abstoßenden Kräfte in Betracht gezogen werden.

Um zu verhindern das sich die Knoten der jeweiligen logischen Graphen zu sehr im gesamten Layoutbereich verteilen, und somit die Graphen nicht mehr im Layout hervortreten, wird der Bereich in dem sie positioniert werden können kreisförmig um die Postion des jeweiligen SVG Knoten beschränkt. Der Radius des Kreises entspricht dabei der Konstante $superK$ die wie folgt definiert ist:

$$superK = \sqrt{\frac{\text{Fläche}}{\text{Anzahl der SVG Knoten}}} \tag{4.3}$$

Wie bereits in der Gleichung 3.3 bezieht sich die Fläche hier auf die gesamte Layoutfläche. Es ist offensichtlich, dass die Konstante $superK$ immer größer ist als k, da die SVG Knoten die logischen Graphen



(a) Schritt 1 (b) Schritt 2 (c) Schritt 3

Abbildung 4.2: Schritte bei der Anwendung des Super Vertex Algorithmus

der Graphmenge repräsentieren und es in den meisten Fällen wesentlich mehr Knoten als logische Graphen in einer Graphmenge gibt. Der Extremfall, dass jeder logische Graph einer Graphmenge genau einen Knoten enthält, und die Zahl der SVG Knoten damit gleich der Anzahl der Knoten in der Graphmenge ist, ist zu vernachlässigen.

Die Anzahl der notwendigen Iterationen die der FR-Algorithmus benötigt um jeden logischen Graphen innerhalb einer begrenzten Layoutfläche zu layouten ist wiederum von der Struktur des jeweiligen logischen Graphen abhängig. Der Algorithmus hat deswegen zwei Parameter. Einen der die Anzahl der Iterationen für den Layoutvorgang des SVG Graphen bestimmt und einen für die Anzahl der Schleifendurchläufe beim layouten der logischen Graphen.

Nachdem der FR-Algorithmus für jeden logischen Graphen eine Position berechnet hat ist das SVL-Verfahren beendet und die Graphmenge damit gelayoutet.

4.2.3 Problematisierung

Wie vielen Layoutalgorithmen, so liegen auch dem SVL-Algorithmus einige Annahmen zu Grunde, die in gewissen Sonderfällen nicht zutreffen und in der Konsequenz zu einem qualitativ minderwertigen Layout führen. So wird davon ausgegangen, dass die Anzahl der Knoten in den jeweiligen logischen Graphen einer Graphmenge ungefähr gleich verteilt ist. Ist dies nicht der Fall, kann es passieren, dass der Layoutbereich eines logischen Graphen, der als Kreis mit Radius $superK$ um die Position des assoziierten SVG Knotens definiert ist, nicht ausreicht um alle Knoten des logischen Graphen zu fassen.

Desweiteren wird davon ausgegangen, dass sich die logischen Graphen der Graphmenge nicht übermäßig schneiden. Nur dann kann die Assoziation eines Knotens mit dem ersten Graphen aus der Menge der Graphen die ihn enthalten zu einer guten Aufteilung der Knoten führen. Im schlechtesten Fall enthalten alle logischen Graphen alle Knoten der Graphmenge. Das würde dazu führen das alle Knoten den selben Kopfgraph hätten. In dem Fall würde das Verhalten des Verfahrens auf den FR-Algorithmus zurückfallen.

Hier wie auch beim Asymmetric Force Directed Layout kommt es also darauf an, wie das EPGM zur Modellierung der Daten genutzt wird.

5 Implementierung

Im Folgenden wird genauer auf die Implementierung des Asymmetric Force Directed Layout und des Super Vertex Layout Algorithmus eingegangen. Die Referenzimplementierungen sind als Gradoop Operatoren verwirklicht und erweitern somit Gradoop's Repertoire an Layout-Operatoren. Bei der Implementierung wurde auf ein einheitliches Interface und leichte Zugänglichkeit geachtet. So lassen sich viele Graphmengen bereits über die Angabe der Knotenanzahl layouten. Gleichzeitig sind die Operatoren sehr flexibel. Alle wesentlichen Parameter der Layoutverfahren lassen sich anpassen.

Die `GraphCollection` ist bei Gradoop der Datentyp, der eine Graphmenge im Sinne des EPGM repräsentiert. Das in Abschnitt 2.2 beschriebende Konzept des *Dataset* wird bei Apache Flink durch die Klasse `DataSet` abgebildet. Um iterative Operationen auf einem `DataSet` zu ermöglichen, stellt Flink ausserdem die Klasse `IterativeDataSet` zur Verfügung. Die Layout-Operatoren konsumieren jeweils ein `GraphCollection` Objekt und geben eine neue `GraphCollection` zurück, in der die per X und Y Koordinaten kodierten Positionen der jeweiligen Knoten als `PropertyValue` gespeichert sind. Die Algorithmen selbst sind dem Objekt-Orientierten Paradigma folgend als Klassen implementiert, welche mittels einer Builder-Klasse instanziiert werden können. Neben den für die Instanziierung notwendigen `builder()` Methoden verfügen sie lediglich über eine öffentliche `execute(GraphCollection)` Methode, welche das Layout für die gegebene `GraphCollection` berechnet. Alle für den jeweiligen Algorithmus relevanten Parameter werden also während der Objekt-Instanziierung gesetzt, was zu einer übersichtlichen API führt.

5.1 AsymmetricForceDirectedLayout

Die Klasse `AsymmetricForceDirectedLayout` stellt die Referenzimplementierung des gleichnamigen Algorithmus dar. Die Abbildung 5.3a liefert einen Überblick über die Attribute und Methoden der Klasse.

Bis auf die Berechnung der anziehenden und abstoßenden Kräfte orientiert sich der Aufbau der Klasse am `FRLayouter` von Baumgarten, die dieser im Rahmen seiner Masterarbeit „Verteiltes Graph-Layouting mit Gradoop“ [7] implementiert hat. Allerdings unterscheiden sich die Klassen hinsichtlich der Datentypen auf denen sie operieren.

Nachfolgend wird erläutert, wie die zuvor beschriebenen Schritte des AFDL-Algorithmus technisch umgesetzt worden sind. Ist die gegebene `GraphCollection` nicht bereits in einem vorhergehenden Schritt im Layoutbereich positioniert worden, wird ihr zunächst ein initiales Layout zugewiesen. Dies kann mit beliebigen geeigneten Klassen vom Typ `MapFunction<EPGMVertex, EPGMVertex>` geschehen, per Default wird in der gegebenen Implementierung eine zufällige Positionierung vorgenommen. Daraufhin beginnt die Berechnung des eigentlichen Layouts in einer Schleife mit im Vorhinein festgelegter Iterationsanzahl. Diese Schleife ist durch ein `IterativeDataSet<EPGMVertex>` verwirklicht. Die Anzahl der Durchläufe ist durch das Feld `iterations` definiert, welches den Defaultwert 1 hat, um das Testen der Klasse zu vereinfachen. In der Praxis ist die Anzahl der notwendigen Iterationen von Struktur und Größe der `GraphCollection` abhängig.

In der Schleife werden nun zuerst die abstoßenden Kräfte mittels der privaten Methode

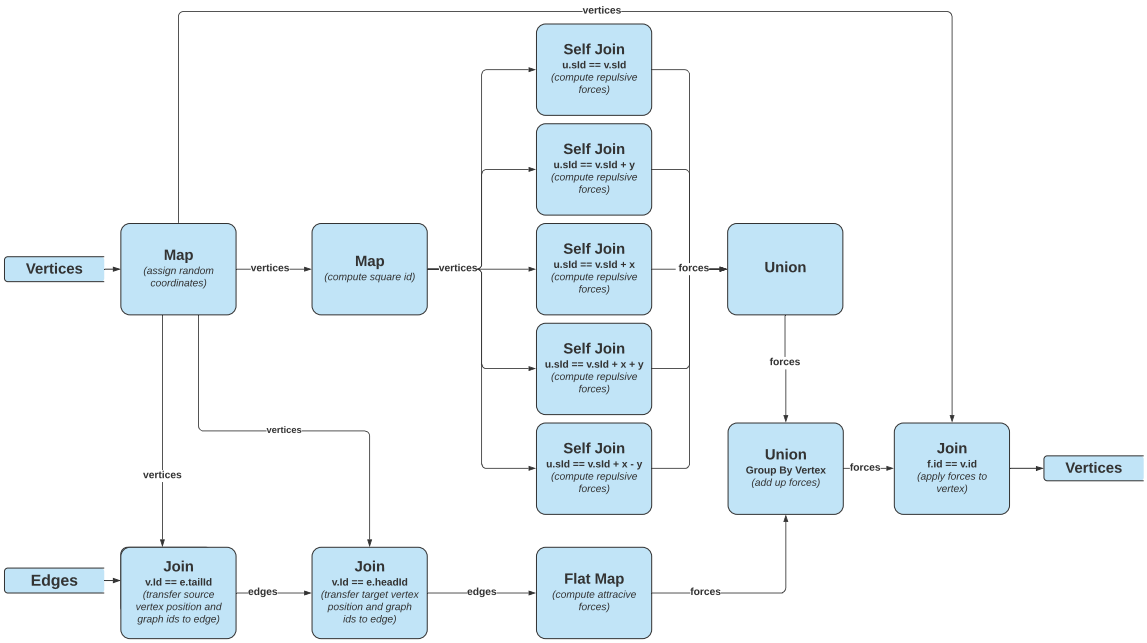


Abbildung 5.1: AFDL Datenfluss

`computeRepulsiveForces()` berechnet. Eine Implementierung des naiven FR-Algorithmus bietet hierbei die Klasse `NaiveRepulsiveForces` die mittels der `DataSet.cross()` Methode das Kreuzprodukt der Menge der Knoten mit sich selbst die abstoßenden Kräfte berechnet und diese in Form eines Objekts vom Typ `DataSet<Force>` zurück gibt. Eine effizientere Option bietet die Klasse `GridRepulsiveForces`, welche die schon von FR vorgeschlagene Optimierung mittels der Unterteilung des Layoutbereichs in Zellen implementiert. Wie die FR-Implementierung von Baumgarten so nutzt auch die Klasse `GridRepulsiveForces` Symmetrieeffekte bei der Berechnung der abstoßenden Kräfte aus. Es ist ausreichend einen Knoten mit der Hälfte der ihn umgebenden Zellen zu Joinen, da die Kraft die auf den anderen Knoten wirkt einfach das inverse der ersten Kraft ist. Die Berechnung von Kraft und inverser Kraft lässt sich mit einer `FlatJoinFunction` in einem Schritt erledigen.

Im nächsten Schritt werden die anziehenden Kräfte zwischen benachbarten Knoten der Graphmenge berechnet. Dies geschieht mittels der private Methode `computeAttractiveForces()`. Diese transferiert zunächst mittels eines doppelten Join der Menge der Knoten mit der Menge Kanten die Koordinaten des Start- und des Zielknotens an das `EPGMEdge` Objekt. Mittels der `DataSet.flatMap()` Methode wird dann aus dem `DataSet` das die Kanten enthält das `DataSet` der anziehendenKräfte erstellt.

Nachdem die abstoßenden und die anziehenden Kräfte berechnet und in den jeweiligen DataSets gefasst sind, werden diese mittels des `DataSet.union()` Operators zusammengefasst und nach der `GradoopId` des jeweils assoziierten Knotens gruppiert. Die Kräfte einer Gruppierung werden dann mittels `DataSet.reduce()` auf addiert, um die für die jeweilige Iteration resultierende Kraft die auf den jeweiligen Knoten wirkt, zu erhalten.

Schlussendlich wird die eigentliche Verschiebung der Knoten aufgrund der auf ihnen wirkenden Kräfte vorgenommen. Die letztendliche Position eines Knotens ist von der auf ihn wirkenden Kraft, den Grenzen des Layoutbereichs sowie einer Simulated Annealing Funktion bestimmt. Durch das Simulated Annealing sollen die Distanzen um die die Knoten verschoben werden mit zunehmender Iterationsanzahl verkleinert werden um das Layout zu verfeinern und das Finden eines globalen Minimums der Energiefunktion zu ermöglichen.

Die Darstellung 5.1 illustriert die beschrieben Abfolge in Form eines Datenfluss Diagramms. Hierbei ist zu

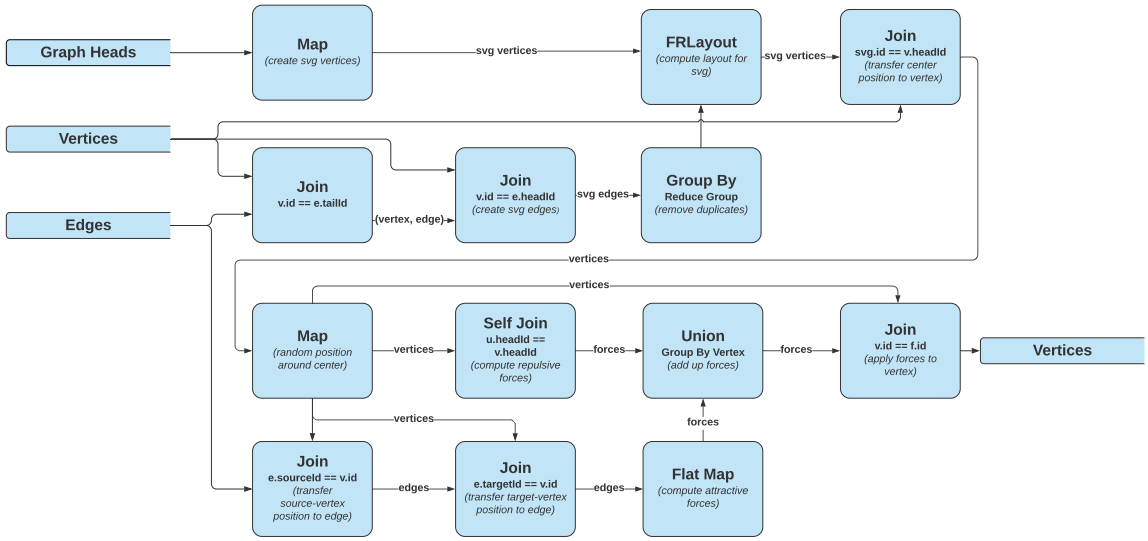


Abbildung 5.2: SVL Datenfluss

beachten, dass die Transformationen nach der Zuweisung der zufälligen Koordinaten und vor der Ausgabe der positionierten Knoten implizit in einer Schleife ausgeführt werden. Die Beschriftungen der Pfeile geben den Datentyp an, der von der vorhergehenden Transformation ausgegeben wird und von der kommenden Transformation konsumiert wird.

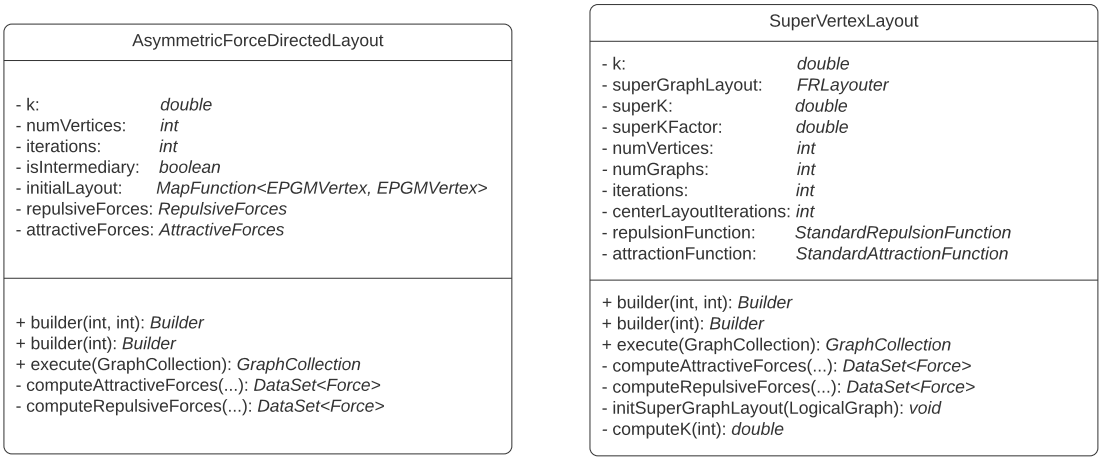
5.2 SuperVertexLayout

Wie zuvor erwähnt, ähnelt der generelle Aufbau der Klasse `SuperVertexLayout` dem der AFDL Implementierung. Die Abbildung 5.3b bietet einen Überblick über ihre Felder und Methoden. Hier fallen zwei neue Felder auf. Das Feld `superK` beinhaltet den Wert der Konstante $superK$ des SVG. Das Feld `superKFactor` beinhaltet einen zum Zeitpunkt der Objekterzeugung festlegbaren Wert, mit dem der Wert für `superK` skaliert werden kann. Dies ist nützlich für den Fall, in dem der SVG sehr dicht vernetzt ist. Dann kann es zu einer ungenügenden Ausdehnung des SVG im Layoutbereich kommen, der man mit der Erhöhung des Parameters `superKFactor` entgegen wirken kann.

Zu beginn wird in der `SuperVertexLayout.execute()` Methode aus der Eingabe der namensgebende Super Vertex Graph erzeugt. Hierzu werden zuerst für jeden logischen Graphen der Graphmenge ein Knoten des SVG erzeugt. Dies geschieht mittels einer `MapFunction`, die für jedes Element im *Dataset* der `EPGMGraphHead` Objekte ein neues `EPGMVertex` Objekt erzeugt, welches die selbe `GradoopId` besitzt wie das `EPGMGraphHead` Objekt. Ein `EPGMGraphHead` Objekt repräsentiert dabei die Metadaten eines logischen Graphen. Deswegen ist eine Betrachtung dieses *Dataset* für die Erstellung der SVG Knoten ausreichend.

Für der Erstellung der Kanten des SVG ist die Klasse `SuperVertexEdgeMapper` zuständig. In ihrer `join()` Methode werden díe Kopfgraphen der Start- und der Zielknoten aller Kanten der Graphmenge miteinander verglichen. Sind die Kopfgraphen verschieden, erzeugt die Methode ein neues `EPGMEdge` Objekt mit den Gradoop Ids der Kopfgraphen als Start- bzw. Zielknoten. Eventuell existierende parallel Kanten werden mittels des `DataSet.reduceGroup()` Operators entfernt. Aus den resultierenden DataSets wird dann ein neuer `LogicalGraph` erzeugt.

Der `LogicalGraph` der den SVG repräsentiert wird daraufhin durch den `FRLayouter` gelayoutet. Die Anzahl der Iterationen die dieser Layoutprozess in Anspruch nimmt ist über das Feld `centerLayoutIterations`



(a) AsymmetricForceDirectedLayout UML

(b) SuperVertexLayout UML

Abbildung 5.3: UML Klassendiagramme der Referenzimplementierungen

bestimmt. Die aus diesem Layoutprozess resultierenden Positionen $x, y \in \mathbb{R}$ der Knoten sind als Properties der Form `(X: x, Y: y)` vorhanden. Da die Knoten des Super Vertex Graphen die selbe `GradoopId` besitzen wie das jeweilige `LogicalGraph` Objekt das sie repräsentieren, lassen sich die Positionen der SVG Knoten mittels eines Join mit den Knoten der Graphmenge an diese propagieren. Dort werden sie als Properties der Form `(CENTER_X: x, CENTER_Y: y)` gespeichert.

Im nächsten Schritt werden die Knoten der Graphmenge zufällig um die jeweiligen Zentren der logischen Graphen herum positioniert. dabei beträgt der maximale Abstand eines Knotens zu seinem Zentrum dem Feld `superK`. Nun folgt wiederum eine durch ein `IterativeDataSet` umgesetzte Schleife über die DataSets der Knoten und Kanten. Allerdings müssen jetzt für die Berechnung der abstoßenden Kräfte lediglich die Knoten gejoint werden, die Teil des selben Kopfgraph sind.

Der Datenfluss der durch die beschriebene Implementierung des SVL-Algorithmus entsteht ist in der Abbildung 5.2 in einem Datenfluss Diagramm dargestellt.

6 Evaluierung

Nachfolgend werden die vorhergehend beschriebenen Implementierungen sowohl hinsichtlich der Messgrößen Geschwindigkeit und Skalierbarkeit, als auch nach visuellen Aspekten evaluiert. Anfangs werden die verwendeten Datensätze sowie die Testumgebung vorgestellt. Abschließend folgt eine Evaluation für jede der Implementierungen.

6.1 Datensätze

Die für die Evaluation verwendeten Datensätze sind synthetisch erzeugt und stammen vom FoodBroker-Generator [21]. Der von FoodBroker produzierte Graph bildet Geschäftstransaktionen, wie sie im Bereich der Wirtschaftsanalytik von Interesse sind, ab. Das Modell basiert auf einem fiktiven Lebensmittelmakler, welcher Waren zwischen Produzenten und dem Einzelhandel vermittelt.

Die von FoodBroker erstellten Graphen lassen sich beliebig skalieren, was sie dazu eignet, als Grundlage für die Evaluation der Skalierbarkeit über verschieden große Eingabegrößen zu verwenden. Im Rahmen dieser Arbeit wurden FoodBroker Datensätze mit den Skalierungen 1, 10 und 100 verwendet, die im Folgenden mit den Kürzeln FB1, FB10 und FB100 bezeichnet werden.

Da die ursprüngliche FoodBroker Implementierung den erstellten Datensatz als SQL-Dump ausgibt, und Gradoop aktuell über keine Möglichkeit verfügt, SQL-Formatierte Daten einzulesen, wurde die Implementierung aus dem *gradoop-generators* [9] Repository verwendet, da diese den Datensatz in einem mit Gradoop kompatiblen Format ausgibt.

Die Graphmenge die in Abbildung 6.4 von unterschiedlichen Layout- Algorithmen gelayoutet wurde ist manuell erstellt worden. Die Graphmenge die den Abbildungen 6.5 und 6.3 zugrunde liegt, ist ein Ausschnitt des FB1 Datensatzes. Sie besteht aus 16 logischen Graphen, die zusammengenommen aus 182 Knoten und 767 Kanten bestehen. Im Folgenden wird dieser Datensatz mit dem Kürzel FBS bezeichnet.

Details über eine Auswahl an Eigenschaften der verwendeten Datensätze sind Tabelle 6.1 zu entnehmen.

Skalierung	Knoten	Kanten	Logische Graphen
1	8505	40187	1000
10	72374	384791	10000
100	728580	3891576	100000

Tabelle 6.1: Anzahl der Elemente in verwendeten FoodBroker Datensätzen

6.2 Testumgebung

Die Evaluierung wurde auf einem SN-Cluster der Abteilung für Datenbanksysteme der Uni Leipzig vorgenommen. Der Cluster ist aus 16 physischen Maschinen zusammengesetzt, die jeweils über eine Intel Xeon E5-2430 CPU mit 6 Kernen und 48GB RAM verfügen. Die Benchmarkprogramme liefen jeweils auf der

Flink-Version 1.6.0. Die Skalierbarkeit wurde hinsichtlich der Anzahl an Clusterknoten betrachtet. Die für die Evaluation der visuellen Aspekte der Layouts notwendigen Abbildungen 6.4 sowie 6.5 und 6.3 wurden auf einem Laptop mit einer Intel Core i5-6300U CPU und 12GB RAM erstellt.

Die für die Evaluation notwendigen Statistiken über die Laufzeit der Berechnungen wurden mittels der Klassen `AsymmetricLayoutBenchmark` und `SuperVertexLayoutBenchmark` ermittelt. Diese bieten die Möglichkeit die Layout-Klassen beliebig zu konfigurieren, ein Layout zu erstellen und schließlich die Benötigte Zeit und andere Metadaten in eine Datei zu schreiben. Die Laufzeit wird mittels der Funktion `getNetRuntime()` des jeweiligen Flink-Jobs ermittelt. Dies bietet den Vorteil, das sie lediglich die Berechnungszeit wiedergibt und alle vorhergehenden Schritte ignoriert.

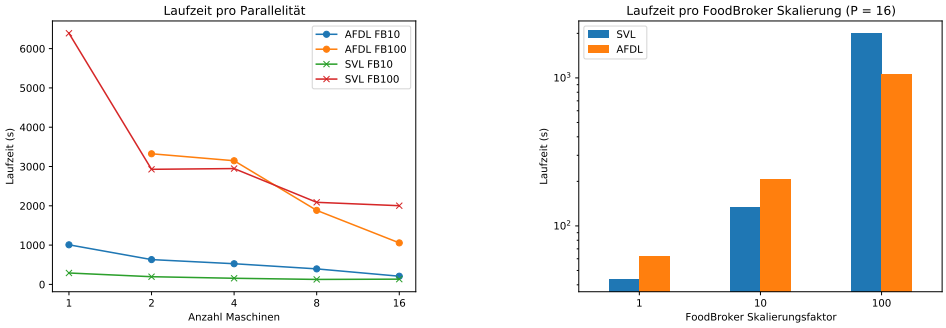
6.3 Testdurchführung

Die Algorithmen wurden auf verschieden großen Datensätzen mit unterschiedlicher Parallelität ausgeführt. Die hier vorgestellten Messergebnisse sind Mittelwerte aus je drei Testläufen mit identischen Parametern. Aus den erhobenen Messdaten werden dann bestimmte Metriken berechnet, welche einen Rückschluss auf die allgemeine Performance und Skalierbarkeit der Algorithmen zulassen. Im folgenden werden in erster Linie die Laufzeiten der Benchmarkprogramme für verschiedene Parallelitäten und verschiedene Eingabegrößen betrachtet. Ausserdem wird für jede Implementierung der sog. Speedup S_P berechnet, der das Verhältnis der Laufzeit bei einer Parallelität 1 zur Laufzeit bei einer Parallelität p für $p \in \mathbb{N}, p \geq 1$ angibt. D bezeichnet dabei die Eingabe. Der Speedup ist also wie folgt definiert:

$$S_p(D) = \frac{T_1(D)}{T_p(D)} \tag{6.1}$$

Der optimale Speedup entspricht einer halbierung der Laufzeit bei einer Verdopplung der Parallelität.

6.4 Ergebnisse



(a) Worker Scale

(b) Data Scale

Abbildung 6.1: Skalierbarkeit je Parallelität & Skalierbarkeit je Eingabegröße

6.4.1 Asymmetric Force Directed Layout

Einen guten ersten Eindruck für die Eigenheiten der Layouts die der AFDL-Algorithmus produziert gewinnt man, wenn man die Darstellung 6.4 betrachtet. Die abgebildete Graphmenge ist verhältnismäßig klein und erlaubt deswegen einen unverstellten Blick auf die Struktur des Layouts. Wie schon zuvor in Darstellung 4.1 ist hier die Graphzugehörigkeit der Knoten über ihre Färbung ersichtlich. Zweifarbige Knoten sind Teil

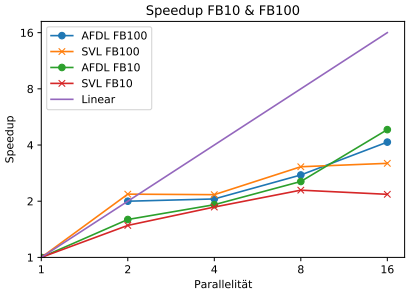


Abbildung 6.2: Speedup

von zwei logischen Graphen. Gerade im Vergleich zum Layout das vom FR-Algorithmus erstellt wurde wird deutlich, dass die logischen Graphen der Graphmenge durch eine geringere Distanz ihrer Knoten zueinander deutlicher hervortreten.

Die Abbildung 6.3 zeigt verschiedene Layouts der FBS Graphmenge, die das Resultat des AFDL-Algorithmus mit zunehmenden Iterationsanzahlen sind. Sowohl hier wie auch in Abbildung 6.5 entspricht die Färbung der Knoten ihrem jeweiligen Kopfgraph. Wenig überraschend entspricht das Layout nach einer Iteration dem einer zufälligen Verteilung der Knoten auf der der Layoutfläche. Ab 10 Iterationen zeichnet sich eine profilierte Darstellung der Graphmenge ab. Nach 40 Iterationen sind die logischen Graphen der Graphmenge klar als eigenständige Strukturen erkennbar. Sind zwei logische Graphen verbunden, so sind die verbindenden Kanten jeweils deutlich länger, als die innerhalb der logischen Graphen. Dennoch ist eine Zusammengehörigkeit klar ersichtlich. Insgesamt weißt das Layout nach 40 Iterationen wenig Kantenkreutzungen auf. Auch sind die Kantenlängen innerhalb der logischen Graphen größtenteils uniform. Die Strukturen weisen Symmetrien auf und sind gleichmäßig in der Layoutfläche verteilt. Das Layout erfüllt also die grundlegenden Anforderungen, die im Kapitel 2.1.2 genannt worden sind.

Die Abbildung 6.1 stellt die Skalierbarkeit sowohl hinsichtlich der Parallelität als auch hinsichtlich der Eingabegröße dar. Aus der Abbildung 6.1a wird ersichtlich, dass die Skalierbarkeit hinsichtlich der Parallelität stark von der größe des Datensatzes abhängt. So verringern sich die Laufzeiten für den FB10 pro Verdopplung der Parallelität nur unwesentlich. Dies ist durch den Overhead zu erklären, der durch die Kommunikation der Clusterknoten untereinander entsteht. Dieser dominiert bei kleineren Eingabegrößen die Laufzeit und führt somit zu einer geringfügigen Skalierbarkeit. Für den FB100 Datensatz ergebend sich allerdings bessere Werte. Zwar existiert kein Wert für die Laufzeit bei Ausführung auf einer einzigen Maschine, für die restlichen Parallelitäten ist allerdings eine stetige Verringerung der Laufzeiten zu beobachten. Diese Verringerung entspricht zwar nicht dem Optimum, sie ist aber stetig. Daraus folgt das der AFDL-Algorithmus hinsichtlich der Parallelität skaliert.

Was die Skalierbarkeit hinsichtlich der Eingabegröße betrifft, so lassen sich aus der Darstellung 6.1b ähnliche Schlüsse ziehen. Das Diagramm zeigt die Laufzeiten des AFDL-Algorithmus für steigende Skalierungen des FoodBroker Datensatzes bei einer maximalen Parallelität von 16 Maschinen. Es zeigt sich, dass die Laufzeiten in erwartbaren Abständen ansteigen. Daraus folgt, dass die Implementierung des AFDL-Algorithmus auch hinsichtlich der Eingabegröße skaliert.

Die Speedup-Metrik bestätigt das beschriebene Verhalten des Algorithmus. Die Laufzeiten für die beiden Datensätze FB10 und FB100 resultieren in zwei ähnlichen Speedup-Kurven. Zwar reichen die Werte nicht an ein Optimum heran, es existiert aber durchaus eine stete Steigerung. Insgesamt entsprechen die Werten denen, die Baumgarten für seine Implementierung des FR-Algorithmus auf dem ca-CondMat Datensatz gemessen hat [7]. Das war zu erwarten, da sich die Implementierungen in vielerlei hinsicht ähneln. Abschließend lässt sich festhalten, dass die Implementierung des AFDL-Algorithmus grundsätzlich

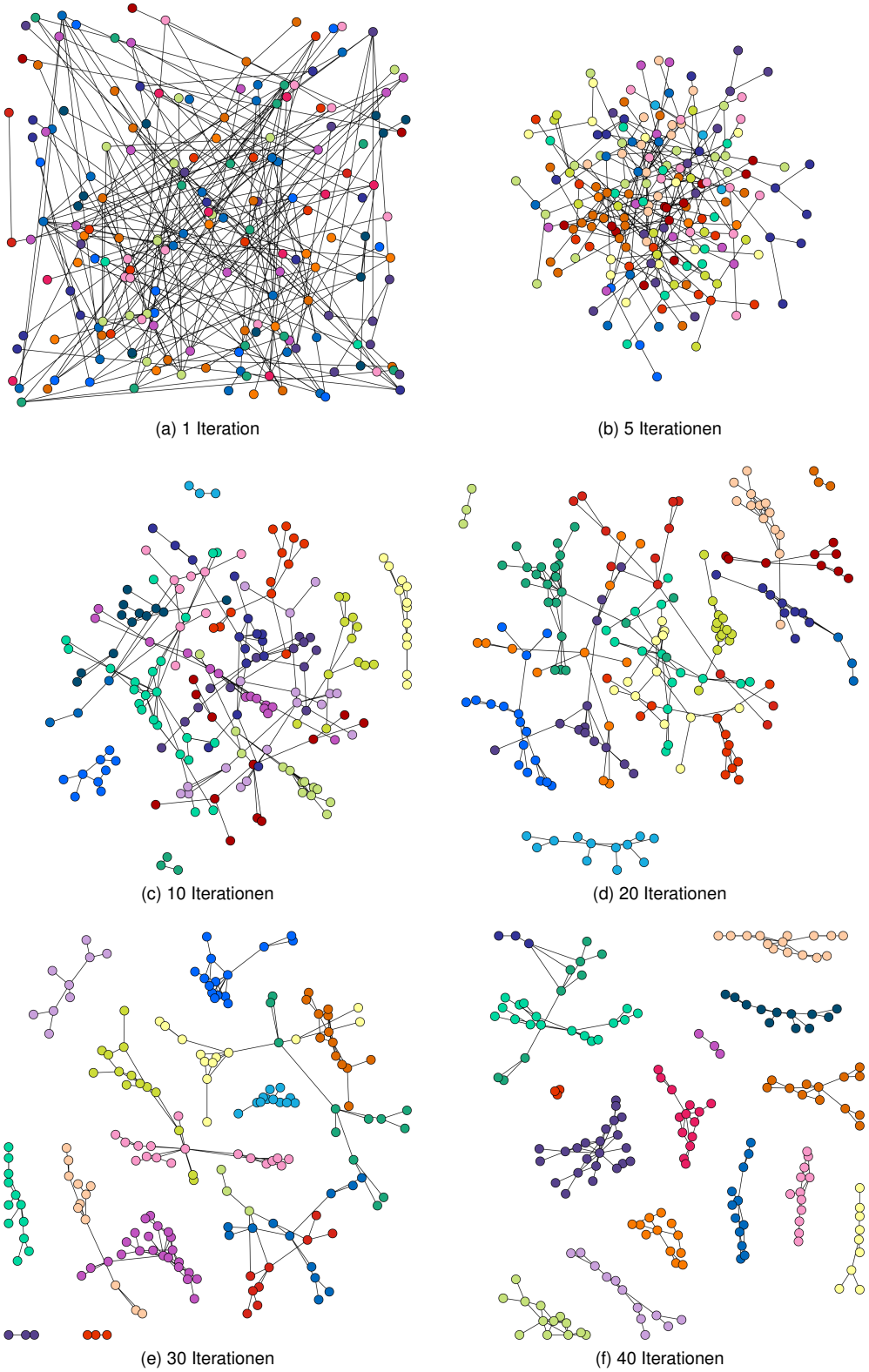
skaliert. Allerdings ist die Skalierbarkeit limitiert, so dass hier noch Raum für Verbesserungen existiert.

6.4.2 Super Vertex Layout

Wie zuvor bei der Evaluation des AFDL-Algorithmus ist es auch hier hilfreich, den visuellen Teil der Evaluation anhand der kleineren Beispielgraphen in Abbildung 6.4 zu Beginnen. Auch hier lässt sich eine tendenzielle Gruppierung der logischen Graphen im Vergleich zum FR-Layout erkennen. Allerdings treten hier Überschneidungen auf, die aus einer unzureichenden Verteilung der SVG-Knoten resultieren können. Auch ist die Struktur des Layouts insgesamt nicht so symmetrisch wie es beim FR-Layout der Fall ist. Dadurch, dass die abstoßenden Kräfte ausschließlich zwischen jenen Knoten berechnet werden, die den selben Kopfgraph haben, kommt es teilweise zu suboptimalen Überschneidungen der Kanten. Dies ist beispielsweise bei der Kante zwischen dem gelben und dem blau-gelben Knoten der Fall. Auch sind die Knoten mancher logischer Graphen zu nah an den Knoten anderer logischer Graphen positioniert, was hier bei den gelben und den violetten Knoten zu sehen ist. Es ist allerdings davon auszugehen, dass diese Effekte mit zunehmender Größe der Graphmengen tendenziell weniger von Bedeutung sind.

Analog zur Abbildung 6.3 enthält die Abbildung 6.5 sechs verschiedene Visualisierungen der FBS Graphmenge die alle ein Ergebnis des SVL-Algorithmus bei steigender Iterationsanzahl sind. Die Iterationsanzahl bezieht sich hier auf die Anzahl der Iterationen die aufgewendet wurden, um die Knoten um ihr jeweiliges Zentrum herum zu positionieren. Der SVG wurde in allen Fällen in fünf Iterationen positioniert. Zum einen fällt auf, dass sich das Layout nach bereits 20 Iterationen nicht mehr wesentlich ändert. Das weist darauf hin, dass das SVL-Verfahren deutlich weniger Iterationen benötigt als der FR-Ansatz. Was das Layout selbst angeht, so erfüllt dies die Anforderungen an ein Graphlayout nur teilweise. Zwar sind die jeweiligen logischen Graphen gut gelayoutet, allerdings überdecken sie sich auch in großen Teilen. Das führt zu vielen Kantenkreutzungen und dementsprechend zu einem unübersichtlichen Layout. Auch treten die logischen Graphen hier nicht als eigenständige Struktur innerhalb des Layouts hervor. Der Grund für diese Ergebnisse ist die Tatsache, dass die logischen Graphen der Graphmenge größtenteils unverbunden sind. Das hat zur Folge, dass der resultierende SVG fast keine Kanten besitzt. Die Knoten solcher Graphen werden beim FR-Layout tendenziell an den Rand der Layoutfläche gedrängt, da ausschließlich abstoßende Kräfte zwischen ihnen wirken. So wird die Fläche nicht optimal genutzt, was zu den Überlagerungen führt die hier zu sehen sind. Der Vergleich mit Abbildung 6.4c zeigt aber, dass das Verfahren bei Graphmengen mit verbundenen logischen Graphen durchaus überschneidungsfreie Layouts erzeugen kann.

Nachdem die visuellen Aspekten des SVL-Algorithmus behandelt wurden, wird die Skalierbarkeit der Implementierung betrachtet. Die beiden Kurven in der Darstellung 6.1a weisen ähnliche Charakteristiken wie die des AFDL-Algorithmus auf. Auch hier scheint sich die Laufzeit bei kleineren Datensätzen wie dem FB10 mit zunehmender Parallelität nur unmerklich zu verringern, was an dem bereits genannten kommunikativen Overhead liegt. Auf dem FB100 verringert sich die Laufzeit mit zunehmender Parallelität stetig, wobei dieser Effekt beim Wechsel von einer auf zwei Maschinen deutlich stärker ausfällt als bei acht auf 16. Wie auch beim AFDL-Algorithmus so liegt auch hier der Zuwachs der Laufzeit bei zunehmender Eingabegröße im erwartbaren Bereich. Auffällig ist, dass die Laufzeiten der SVL Implementierung für den FB1 und den FB10 Datensatz geringer ausfallen als die der AFDL Implementierung, sich dieses Verhältnis für den FB100 Datensatz aber umkehrt. Das könnte ein Hinweis darauf sein, dass der AFDL-Algorithmus besser auf der Eingabegröße skaliert. Auf kleineren Datensätzen weißt der SVL-Algorithmus allerdings deutlich kürzere Laufzeiten auf. Die Speedup-Kurven in Abbildung 6.2 legen den Schluss nahe, dass die Skalierbarkeit geringer ausfällt als beim AFDL-Algorithmus.



(a) 1 Iteration

(b) 5 Iterationen

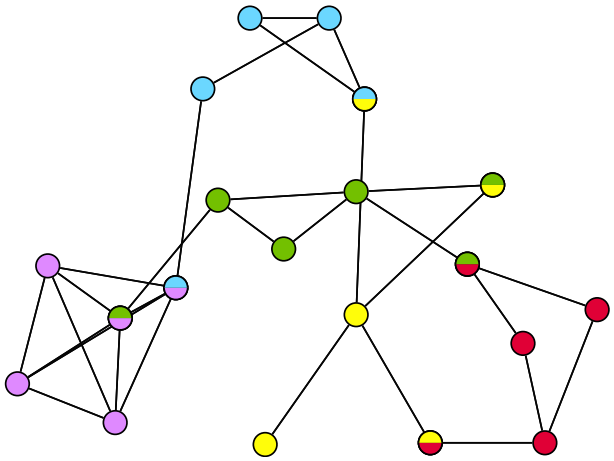
(c) 10 Iterationen

(d) 20 Iterationen

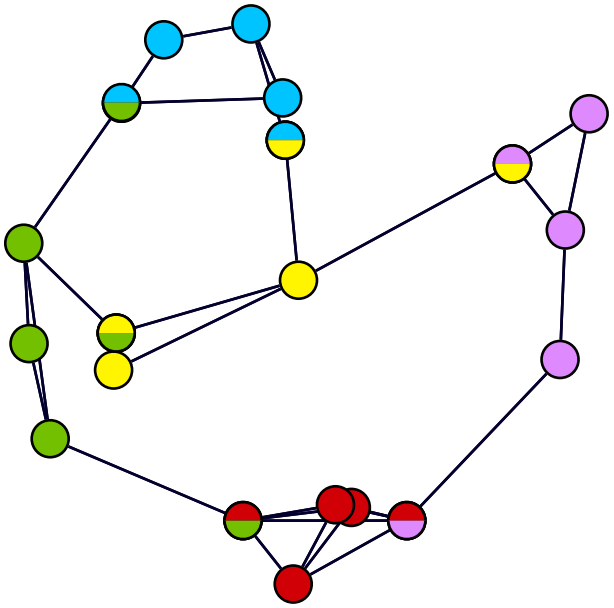
(e) 30 Iterationen

(f) 40 Iterationen

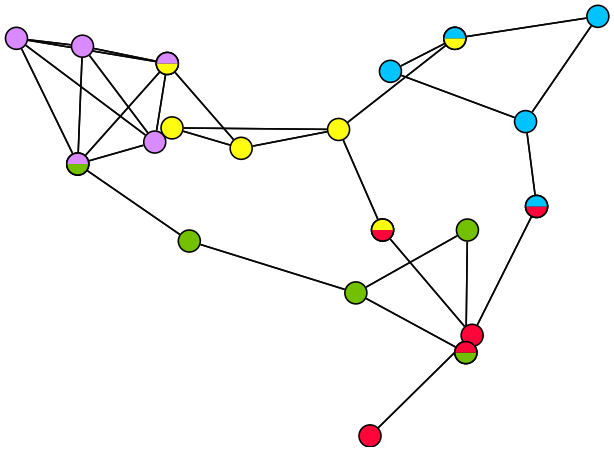
Abbildung 6.3: AFDL nach steigender Iterationsanzahl



(a) FR Layout

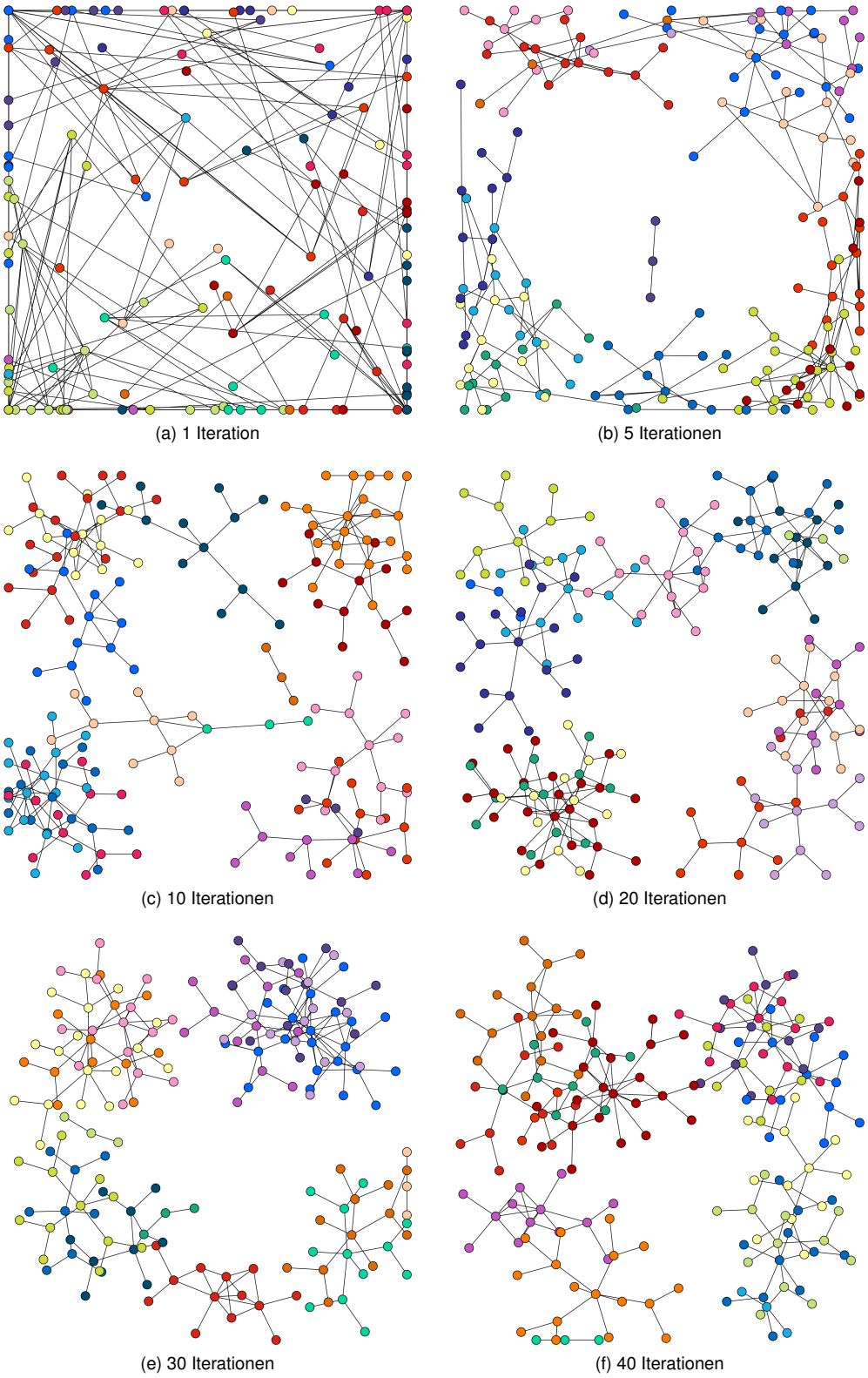


(b) Asymmetric Layout



(c) Super Vertex Layout

Abbildung 6.4: Layout eines kleinen Graphen



(a) 1 Iteration

(b) 5 Iterationen

(c) 10 Iterationen

(d) 20 Iterationen

(e) 30 Iterationen

(f) 40 Iterationen

Abbildung 6.5: SVL nach steigender Iterationsanzahl

7 Zusammenfassung und Ausblick

Im folgenden Abschnitt werden die Ergebnisse dieser Arbeit zusammengefasst und es wird ein Ausblick auf mögliche sich anschließende Fragestellungen gegeben.

Die Evaluation des AFDL und SVL Algorithmus hat gezeigt, dass sich kräftebasierte Layoutverfahren gut dazu eignen Graphmengen zu layouten. Es konnten sinnvolle erste Überlegungen dazu, wie sich eine Graphmenge überhaupt layouten lässt, umgesetzt werden. Somit verfügen Graphmodelle die das Konzept der Graphmenge beinhalten nun nicht nur über ein flexibleres Framework zur Modellierung von Graphdaten, sondern darüber hinaus auch über neue Möglichkeiten diese Daten zu visualisieren. Dies ist gerade für Datenanalysten eine wichtiger Voraussetzung um Informationen aus den Daten extrahieren zu können.

Die Entwicklung des AFDL-Algorithmus war ein erster Schritt, das relative junge Konzept der Graphmenge mit etablierten Layoutverfahren zusammen zu bringen. So bietet die Tatsache, dass sich jeder Knoten einer Graphmenge zu einem oder mehreren logischen Graphen zuordnen lässt interessante Möglichkeiten, das populäre FR Layout derart zu modifizieren, dass zusätzliche Eigenschaften der Graphmenge aus der Struktur des Layouts ersichtlich werden. So lassen sich mit diesem Verfahren besonders gut solche Graphdatensätze visualisieren, welche möglichst disjunkte logische Graphen enthalten und bei denen ein logischer Graph als strukturelles Element der Graphmenge überhaupt von Interesse ist. Aus dieser Beobachtung folgt allerdings auch die Einschränkung, dass die Qualität des so erzeugten Layouts maßgeblich von der Modellierung der Daten durch den Anwender abhängt. Das unterscheidet das Verfahren vom FR-Algorithmus, der nur die grundlegendsten Elemente eines Graphen, Ecken und Kanten, voraussetzt. Mögliche anschließende Forschungsfragen die sich aus der Arbeit an AFDL ergeben betreffen in erster Linie die Skalierbarkeit. So ist die Berechnung nach wie vor zeitaufwendig. Häufig kommt es vor, dass sich gerade in späteren Phasen des Layoutprozesses viele Knoten in der selben Zelle befinden, was die Laufzeitkomplexität wesentlich erhöht. Hier wäre zu prüfen ob sich bereits bekannte Optimierungsstrategien, sowie beispielsweise Multilevel-Ansätze, mit einer Modifizierung der anziehenden und abstoßenden Kräfte zusammenführen lassen würden. Zusammenfassend bestätigt der AFDL-Algorithmus die Ergebnisser der Arbeit von Baumgarten. Es ist möglich, den FR-Algorithmus auf verteiltes In-Memory Dataflow System (VIDS) umzusetzen. Es ist ausserdem möglich ihn an das Konzept der Graphmenge anzupassen.

Obgleich der SVL-Algorithmus hinsichtlich der Modellierung sowie der Skalierbarkeit ähnliche Einschränkungen wie der AFDL-Algorithmus hat, so ist doch auffällig, dass die Laufzeit insgesamt wesentlich geringer ist. Das macht ihn für gewisse Eingabegrößen zu einer geeigneten Alternative zu klassischeren kräftebasierten Verfahren, die auf dem Flink-Cluster, auf dem Evaluierung vorgenommen worden ist, zu Problemen führen können. Dadurch, dass die initiale Platzierung nicht zufällig ist, sondern auf dem Super-Graphen basiert erreichen die Layouts schon nach wenigen Iterationen eine relativ gute Qualität. Die Tatsache, dass insgesamt weniger Iterationen gebraucht werden ist ein großer Vorteil. Ein weiterer Vorteil ist der, dass die Komplexität der Berechnung der abstoßenden Kräfte im Verlauf des Layouts konstant bleibt, da sie nur zwischen den Knoten eines logischen Graphen berechnet werden und sich die Graphzugehörigkeit nicht verändert. Eine plötzliche Vergrößerung der Menge der für die Berechnung der abstoßenden

Kräfte zu vergleichenden Knoten kann also, anders als beim AFDL-Algorithmus, ausgeschlossen werden. Es stellen sich hier allerdings auch eine Fragen, denen nachzugehen lohnenswert wäre. So wäre für das SVL-Verfahren zu klären, ob sich die Fläche, die den Knoten eines jeden logischen Graphen für die Positionierung zugewiesen wird, dynamisch an die größe des jeweiligen logischen Graphen anpassen lässt. Bislang liegt dem Entwurf des Algorithmus die Annahme zugrunde, dass jeder logische Graph aus etwa gleich vielen Knoten besteht. Doch dies muss offensichtlich nicht der Fall sein, und so wäre eine dynamischere Zuteilung der Layoutfläche je logischer Graph interessant. Eine andere Richtung in die die Erforschung des SVL-Ansatzes gehen könnte wäre die Frage, ob sich nicht anderen Möglichkeiten finden lassen die Knoten der Graphmenge mit einem logischen Graphen zu assoziieren. Das Konzept des Kopfgraphen mag für einige Graphmengen ausreichen, es gibt aber sicherlich auch solche bei denen dieser Ansatz fehlschlägt. So könnte man untersuchen ob es sinnvoll ist, einen Knoten mit dem kleinsten logischen Graphen zu assoziieren, dessen Teil er ist. Auch die Frage, ob nicht *Cooling Schedules* existieren, die sich besser für das Layouten von Graphmengen eignen als der hier verwendete wäre von interesse. Ein besserer *Cooling Schedule* könnte zu einer Verringerung der notwendigen Iterationen sowohl für das Asymmetric Force Directed Layout als auch für das Super Vertex Layout führen. Zuletzt muss natürlich auch der Frage weiter nachgegangen werden, wie sich die logischen Graphen einer Graphmenge überhaupt als eigenständige Strukturen innherhalb eines Layouts herausstellen lassen. Die im Rahmen dieser Arbeit entwickelten Ansätze gingen beide von der Annahme aus, dass die relative Nähe der Knoten zueinander die eindeutigste Möglichkeit darstellen. Ob das jedoch für alle Domänen der Fall ist, ist bislang noch offen.

Literatur

[1] *Amazon Neptune - Fast, Reliable Graph Database built for the cloud*. Amazon Web Services, Inc. Library Catalog: aws.amazon.com. URL: https://aws.amazon.com/neptune/ (besucht am 31. 07. 2020).

[2] Renzo Angles. „The property graph database model.“ In: *AMW*. 2018.

[3] *Apache Accumulo*. URL: https://accumulo.apache.org/ (besucht am 04. 07. 2020).

[4] *Apache Hadoop*. URL: https://hadoop.apache.org/ (besucht am 04. 07. 2020).

[5] *Apache HBase – Apache HBase™ Home*. URL: https://hbase.apache.org/ (besucht am 05. 08. 2020).

[6] *Apache Spark™ - Unified Analytics Engine for Big Data*. URL: https://spark.apache.org/ (besucht am 21. 10. 2019).

[7] Daniel Baumgarten. „Verteiltes Graph-Layouting mit Gradoop“. Master Thesis. Leipzig: Leipzig University, 2019. 87 S.

[8] *Boost Graph Library: Fruchterman-Reingold Force-Directed Layout - 1.63.0*. URL: https://www.boost.org/doc/libs/1_63_0/libs/graph/doc/fruchterman_reingold.html (besucht am 07. 08. 2020).

[9] *dbs-leipzig/gradoop-generators*. original-date: 2018-10-09T12:00:39Z. 23. Okt. 2018. URL: https://github.com/dbs-leipzig/gradoop-generators (besucht am 30. 07. 2020).

[10] Peter Eades. „A Heuristic for Graph Drawing“. In: *Congressus Numerantium* 42 (1984), S. 149–160.

[11] Thomas M. J. Fruchterman und Edward M. Reingold. „Graph drawing by force-directed placement“. In: *Software: Practice and Experience* 21.11 (Nov. 1991), S. 1129–1164. ISSN: 00380644, 1097024X. DOI: 10.1002/spe.4380211102. URL: http://doi.wiley.com/10.1002/spe.4380211102 (besucht am 11. 10. 2019).

[12] Pawel Gajer, Michael T. Goodrich und Stephen G. Kobourov. „A multi-dimensional approach to force-directed layouts of large graphs“. In: *Computational Geometry* 29.1 (Sep. 2004), S. 3–18. ISSN: 09257721. DOI: 10.1016/j.comgeo.2004.03.014. URL: https://linkinghub.elsevier.com/retrieve/pii/S092577210400046X (besucht am 21. 10. 2019).

[13] *Graphzeichnen*. In: *Wikipedia*. Page Version ID: 190961402. 1. Aug. 2019. URL: https://de.wikipedia.org/w/index.php?title=Graphzeichnen&oldid=190961402 (besucht am 11. 10. 2019).

[14] Ronny Hadany und David Harel. „A multi-scale algorithm for drawing graphs nicely“. In: *Discrete Applied Mathematics* (2001), S. 19.

[15] Antoine Hinge und David Auber. „Distributed Graph Layout with Spark“. In: *2015 19th International Conference on Information Visualisation*. 2015 19th International Conference on Information Visualisation (iV). Barcelona: IEEE, Juli 2015, S. 271–276. ISBN: 978-1-4673-7568-9. DOI: 10.1109/iV.2015.56. URL: https://ieeexplore.ieee.org/document/7272614/ (besucht am 11. 10. 2019).

[16] Martin Junghanns u. a. „Analyzing extended property graphs with Apache Flink“. In: *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics - NDA '16*. the 1st ACM SIGMOD Workshop. San Francisco, California: ACM Press, 2016, S. 1–8. ISBN: 978-1-4503-4513-2. DOI: 10.1145/2980523.2980527. URL: http://dl.acm.org/citation.cfm?doid=2980523.2980527 (besucht am 18. 10. 2019).

[17] Martin Junghanns u. a. „GRADOOP: Scalable Graph Data Management and Analytics with Hadoop“. In: *arXiv:1506.00548 [cs]* (1. Juni 2015). arXiv: 1506.00548. URL: http://arxiv.org/abs/1506.00548 (besucht am 16. 10. 2019).

[18] Stephen G. Kobourov. „Spring Embedders and Force Directed Graph Drawing Algorithms“. In: *arXiv:1201.3011 [cs]* (14. Jan. 2012). arXiv: 1201.3011. URL: http://arxiv.org/abs/1201.3011 (besucht am 22. 09. 2020).

[19] Eric Miller. „An introduction to the resource description framework“. In: *Bulletin of the American Society for Information Science and Technology* 25.1 (1998), S. 15–19. DOI: 10.1002/bult.105. URL: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/bult.105.

[20] *Neo4j Graph Platform – The Leader in Graph Databases*. Neo4j Graph Database Platform. Library Catalog: neo4j.com. URL: https://neo4j.com/ (besucht am 31. 07. 2020).

[21] André Petermann u. a. „FoodBroker - Generating Synthetic Datasets for Graph-Based Business Analytics“. In: *Big Data Benchmarking*. Hrsg. von Tilmann Rabl u. a. Bd. 8991. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, S. 145–155. ISBN: 978-3-319-20232-7 978-3-319-20233-4. DOI: 10.1007/978-3-319-20233-4_13. URL: http://link.springer.com/10.1007/978-3-319-20233-4_13 (besucht am 30. 04. 2020).

[22] Michael Stonebraker. „The case for shared nothing“. In: *IEEE Database Eng. Bull.* 9.1 (1986), S. 4–9.

[23] *Titan: Distributed Graph Database*. URL: https://titan.thinkaurelius.com/ (besucht am 31. 07. 2020).

[24] Chris Walshaw. „A Multilevel Algorithm for Force-Directed Graph-Drawing“. In: *Journal of Graph Algorithms and Applications* 7.3 (), S. 33.

Akronyme

AFDL Asymmetric Force Directed Layout. 14, 15, 17, 20, 23, 24, 25, 29, 30

EPGM Extended Property Graph Model. 4, 9, 13, 14, 16, 17, 18

FR Fruchterman und Reingold. 5, 11, 12, 13, 14, 15, 16, 17, 19, 24, 25, 29

PGM Property Graph Model. 8, 9

RDF Resource Description Framework. 8

SN Shared-Nothing. 4, 22

SVG Super Vertex Graph. 16, 17, 20, 25

SVL Super Vertex Layout. 16, 17, 21, 25, 29, 30

VIDS verteiltes In-Memory Dataflow System. 8, 14, 29

Erklärung

„Ich versichere, dass ich die vorliegende Arbeit selbständig 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“.

Ort	Datum	Unterschrift