

Multilayer Graph Analyse in GraphEngine

Masterarbeit

von

Johannes Thiel

aus

Solingen

vorgelegt an der

Abteilung Betriebssysteme

Prof. Dr. Michael Schöttner

Heinrich-Heine-Universität Düsseldorf

25. August 2020

Gutachter:

Prof. Dr. Michael Schöttner

Zweitgutachter:

Prof. Dr. Gunnar W. Klau

Inhaltsverzeichnis

1	Einleitung	3
2	Grundlagen	5
2.1	Graphen	5
2.1.1	Dichte	5
2.1.2	Knotengrad	5
2.1.3	Multilayer Graphen	5
2.2	Page Rank	7
2.3	Hubs and Authorities	7
2.4	Graph Engine	9
2.4.1	Memory Cloud	9
2.4.2	Graph Model	10
2.4.3	Computation Engine	10
2.4.4	Datenzugriff	12
2.5	Andere Multilayer Graph Systeme	13
2.5.1	MuxViz	13
2.5.2	Multilayer Networks Library for Python	13
3	Architektur	15
3.1	Model	15
3.2	Lib	15
3.3	Client	17
3.4	Proxy	17
3.5	Server	17
3.5.1	Laden	18
4	Implementierung	19
4.1	Graph Engine	19
4.1.1	Einrichtung	19
4.1.2	Datenzugriff	19
4.1.3	Schwierigkeiten	20
4.2	Implementierung	21
4.2.1	Model	21
4.2.2	Lib	23

4.2.3	Client	23
4.2.4	Proxy	25
4.2.5	Server	28
4.2.6	Algorithmen	30
4.2.7	Erweiterbarkeit	31
5	Evaluation	37
5.1	Cluster	37
5.2	Testdaten	37
5.3	Benchmarks	38
5.3.1	Ladezeiten	38
5.3.2	Graph Dichte	38
5.3.3	HITS	39
6	Fazit	41
6.1	Ausblick	42
6.1.1	Benutzeroberfläche	42
6.1.2	Grafische Auswertung	42
6.1.3	Cluster Verwaltung	42
6.1.4	Algorithmen	42
A	Anhang	43
A.1	Einrichtung Graph Engine	43
	Literaturverzeichnis	45
	Abbildungsverzeichnis	47

Zusammenfassung

Die Analyse von Graphen ist in vielen Forschungsfeldern relevant. Viele Sachverhalte lassen sich besonders gut durch Multilayer Graphen darstellen. Die Werkzeuge und Programme zur Analyse von Multilayer Graphen sind jedoch nicht auf Skalierbarkeit für besonders große Graphen ausgerichtet.

In dieser Arbeit wird ein verteiltes und skalierbares System zur Verarbeitung und Analyse von Multilayer Graphen entwickelt. Dabei werden drei Anwendungen, Client, Proxy und Server entwickelt, die zusammen genutzt werden. Das System nutzt das von Microsoft entwickelte Graph Engine als verteilten Key-Value Speicher und zum Nachrichtenaustausch zwischen den einzelnen Komponenten. Das entwickelte System ist um weitere Algorithmen, Eingabe- und Ausgabeformate erweiterbar. Es wird gezeigt, dass das System in der Lage ist große Multilayer Graphen zu laden und Berechnungen auf diesen durchzuführen. Insbesondere wird gezeigt, dass das System skalierbar ist und durch mehr Server die Berechnungen schneller durchführt.

Kapitel 1

Einleitung

Die Untersuchung von komplexen Netzwerken ist für viele verschiedene Forschungsbereiche, wie der Biologie, Soziologie, Transportation, Physik und vielen mehr, relevant [4]. Netzwerke können viele verschiedene reale oder künstliche Systeme darstellen und für die Forschung an diesen genutzt werden. Netzwerke wurden zum Beispiel benutzt, um Beziehungen zwischen Personen, Verlinkungen zwischen Websites, Interaktion zwischen Proteinen und mehr darzustellen.

In traditionellen Netzwerken gibt es in der Regel nur eine Art von Kante, die alle Verbindungen zwischen Knoten beschreiben muss. Diese Einschränkung ist in den meisten Fällen eine Vereinfachung und kann dazu führen, dass gewisse Probleme nur schwer angegangen werden können.

Multilayer Netzwerke besitzen verschiedene Arten von Verbindungen zwischen Knoten und können somit Systeme darstellen, in denen eine Entität verschiedene Nachbarn in verschiedenen Ebenen hat. Diese Ebenen können, je nach Anwendungsfall, verschiedene Kategorien sein. In diesen Multilayer Netzwerken gibt es neben den Verbindungen zwischen Knoten in der gleichen Ebene noch Verbindungen zwischen Knoten unterschiedlicher Ebenen. So lassen sich Systeme wie Transportnetzwerke mit verschiedenen Modi der Transportation und dem Übergang zwischen diesen Modi darstellen.

In vielen Bereichen steigen die entstehenden Datenmengen, die für Netzwerkanalysen benutzt werden, immer weiter an. Ein großes Beispiel hierfür sind soziale Netzwerke, in denen sich die Verbindungen zwischen Nutzern und deren Interaktionen durch Multilayer Netzwerke darstellen lassen.

Zur Analyse von Multilayer Netzwerken wurden bereits verschiedene Anwendungen, wie muxViz [3] entwickelt. Diese Anwendungen können Multilayer Netzwerke laden, verschiedene Statistiken zu ihnen bilden und Algorithmen auf ihnen laufen lassen.

Die verschiedenen Anwendungen haben jedoch keinen verteilten Ansatz und laufen alle auf einer einzelnen Maschine. Dadurch können Graphen, die zu groß für den Arbeitsspeicher einer Maschine sind, nicht verarbeitet werden. Um solch große Graphen zu handhaben bietet sich ein verteilter Ansatz mit mehreren Maschinen an, auf die der Graph aufgeteilt wird.

Es gibt Systeme zur verteilten Verarbeitung von großen Graphen, wie zum Beispiel Pregel [7] oder Apache Giraph. Diese Systeme können große Graphen verteilt auf vielen Ma-

schinen verarbeiten und effizient Berechnungen auf diesen Graphen durchführen. Allerdings sind sie nicht auf Multilayer Netzwerke ausgerichtet und können nur mit klassischen Graphen umgehen.

Ein verteiltes Graphsystem, in welchem die Darstellung des Graphen frei gewählt werden kann, ist das von Microsoft Research Asia entwickelte Graph Engine [10] [9]. In dieser Arbeit wird untersucht, inwiefern mit Graph Engine ein System zur verteilten Verarbeitung von Multilayer Graphen geschaffen werden kann. Dabei wird untersucht, wie die Freiheiten in der Graph Darstellung und Speicherung von Graph Engine genutzt werden können, um Multilayer Graphen in Graph Engine zu speichern. Zudem bietet Graph Engine die Möglichkeit Nachrichten zwischen den einzelnen Maschinen auszutauschen und aufgrund dieser Berechnungen durchzuführen. Auch diese Kommunikation kann frei gestaltet und für die Zwecke der Multilayer Graph Verarbeitung genutzt werden.

In dieser Arbeit soll mithilfe von Graph Engine ein System erstellt werden, welches Multilayer Graphen laden, verändern und Algorithmen auf ihnen ausführen kann. Dafür müssen eine Client-, Proxy- und Serveranwendung entwickelt werden, die miteinander arbeiten. Dabei liegt der Fokus darauf, ein allgemeines System zu erstellen, das erweitert werden kann.

Kapitel 2

Grundlagen

2.1 Graphen

Im Allgemeinen kann ein Graph durch $G = (V, E)$ dargestellt werden, wobei V eine Menge an Knoten und E eine Menge an Kanten ist. Die Kanten sind jeweils ein geordnetes Knotenpaar $(u, v) \subseteq V \times V$.

2.1.1 Dichte

Die Dichte eines Graphen gibt das Verhältnis zwischen der Anzahl an Kanten und der potentiell möglichen Anzahl an Kanten an [5]. Sie lässt sich bestimmen mit:

$$d(G) = \frac{2|E|}{|V|(|V| - 1)}$$

2.1.2 Knotengrad

In einem gerichteten Graphen besitzt jeder Knoten einen Ausgangs- und Eingangsgrad. Dies ist jeweils die Anzahl an Kanten, die bei dem Knoten starten bzw. bei ihm enden [5]. Der Ausgangsgrad für einen Knoten v im Graph G wird mit $d_G^+(v)$ und den Eingangsgrad mit $d_G^-(v)$ bezeichnet.

2.1.3 Multilayer Graphen

Im Folgenden werden Multilayer Graphen betrachtet. Diese Graphen haben neben Knoten und Kanten auch eine Menge an Ebenen. Dabei kann, je nach Art des Graphen, der gleiche Knoten in verschiedene Ebenen vorhanden sein oder auch Verbindungen zwischen den Knoten in verschiedenen Ebenen bestehen. Welche Bedeutung den Ebenen zukommt, hängt vom Graphen und dem Anwendungsfall der Daten ab. Multilayer Graphen können genutzt werden, um Netzwerke in unterschiedlichen Themengebieten abzubilden, z.B. soziale Netzwerke, Transport oder Biologie.

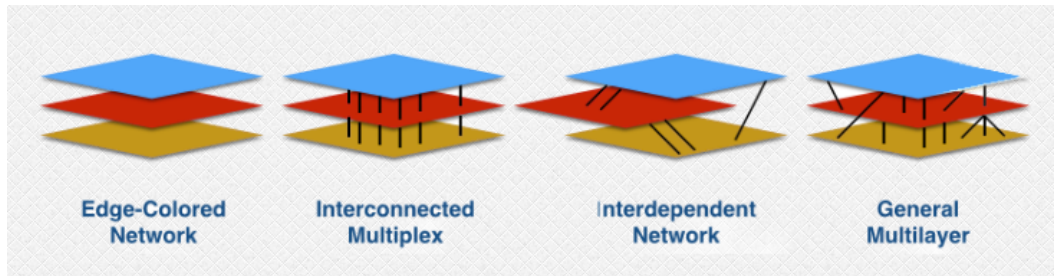


Abbildung 2.1: Multilayer Netzwerkarten [1]

Multilayer Graphen können, abhängig von ihren Verbindungen, in verschiedene Kategorien eingeteilt werden. Die verschiedenen Kategorien sind in Abbildung 2.1 dargestellt.

Edge Colored Network

In Edge Colored Networks gibt es nur Kanten zwischen Knoten im gleichen Layer. Dabei sind die Kanten jeweils mit einem Label versehen das kennzeichnet, zu welchem Layer sie gehören.

Interconnected Multiplex

In Multiplex Netzwerken gibt es hauptsächlich Kanten zwischen Knoten, die sich im gleichen Layer befinden. Die einzigen Kanten, die es zwischen verschiedenen Layern gibt, verbinden jeweils den gleichen Knoten in dem anderen Layer.

Inderdependet Network

In Inderdependet Networks gibt es keine Knoten, die in mehr als einem Layer vorkommen. Kanten können Knoten innerhalb eines Layers, aber auch zwischen zwei Layern verbinden.

Generelle Multilayer

In einem Generellen Multilayer Graph gibt es keine Restriktion, wie die Knoten der verschiedene Ebenen miteinander verbunden sein können. Es kann sowohl Verbindungen zwischen dem gleichen Knoten in verschiedenen Ebenen geben, als auch Verbindungen zu Knoten in anderen Ebenen.

Ein genereller Multilayer Graph kann formal als ein gerichteter Multigraph [4] $G = (\Sigma_V, \Sigma_E, V, E, s, t, l_V, l_E)$, dessen Knoten und Kanten ein Label besitzen, definiert werden, wo

- V die Menge an Knoten und E die Menge an Kanten ist
- Σ_V und Σ_E die Alphabete sind, die als Label für Knoten und Kanten dienen
- $s : E \rightarrow V$ und $t : E \rightarrow V$ zwei Zuordnungen sind, die angeben, von welchem Knoten eine Kante ausgeht und zu welchem sie führt

-
- $l_V : V \rightarrow \Sigma_V$ und $l_E : V \rightarrow \Sigma_E$ zwei Zuordnungen sind, die den Knoten und Kanten ihre Labels zuweisen

2.2 Page Rank

Page Rank wurde von Larry Page, Sergey Brin, R. Motwani und T. Winograd in der Arbeit "The PageRank Citation Ranking: Bringing Order to the Web" [8] vorgestellt. Sie beschreiben, wie PageRank genutzt werden kann, um die Wichtigkeit von Internetseiten zu bewerten.

Neben Internetseiten kann PageRank aber auch auf andere Graphen angewandt werden. PageRank bewertet Knoten in einem Graphen anhand ihrer Kanten. Knoten erhalten eine hohe Bewertung, wenn sie viele eingehenden Kanten besitzen. Auch die Bewertung der Knoten, von denen die eingehenden Kanten ausgehen, wirkt sich auf die Bewertung aus.

Da die Daten verteilt auf mehreren Servern liegen und keine globale Sicht existiert, wird die iterative Variante von PageRank betrachtet. Bei dieser verteilt jeder Knoten in jeder Iteration seinen eigenen PageRank Wert gleichmäßig auf alle Knoten, zu denen er ausgehende Kanten besitzt.

Sei $PR(p_i; t)$ der PageRank Wert des Knoten p_i zu dem Iterationsschritt t und die gesamte Zahl aller Knoten N . Zudem sei $M(p_i)$ die Knoten, die eine ausgehende Kante zu p_i haben und $L(p_i)$ die Anzahl an ausgehenden Kanten von Knoten p_i . Jeder Knoten startet mit einem PageRank Wert von:

$$PR(p_i; 0) = \frac{1}{N}$$

Bei jedem Iterationsschritt wird für jeden Knoten ein neuer Wert mit folgender Formel berechnet:

$$PR(p_i; t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j; t)}{L(p_j)}$$

Dabei ist d ein Dämpfungsfaktor. Er verhindert, dass die Werte nicht nur zu Knoten fließen, die keine ausgehenden Kanten haben. Als Abbruchbedingung für die Iterationen können eine feste Anzahl an Iterationsschritten gewählt werden. Alternativ kann auch ein Grenzwert ϵ festgelegt werden und der Algorithmus bricht ab, sobald die Veränderung aller Werte unter ϵ liegt.

2.3 Hubs and Authorities

Hubs and Authorities (HITS) ist ein Algorithmus zur Bewertung von Knoten in einem Netzwerk, der von John M. Kleinberg in dem Paper "Authoritative Sources a Hyperlinked Environment" vorgestellt wurde [6].

Dabei werden in einem Netzwerk aufeinander verweisende Dokumente beurteilt. Beispiele für solche Netzwerke sind Internetseiten, die aufeinander verlinken oder wissenschaftliche

Veröffentlichungen, die einander referenzieren. Ein Hub sind Dokumente, die auf viele gute Quellen verweisen, während Authorities gute Quellen sind, auf die oft verwiesen wird. Der Algorithmus läuft ähnlich zum PageRank-Algorithmus ab, jedoch wird nicht nur ein einzelner Wert für jeden Knoten bestimmt, sondern jeweils ein Hub und Authority Wert pro Knoten.

Für einen gerichteten Graphen $G = (V, E)$ weisen wir jedem Knoten $v \in V$ einen Hub y_v und Authority x_v Wert zu.

HITS ist ein iterativer Algorithmus, der für jeden Knoten eines Graphen jeweils einen Hub und einen Authority Wert bestimmt. Hierfür werden die Hub oder Authority Werte der benachbarten Knoten genutzt. Damit die Authority und Hub Werte sich nicht gegen unendlich aufschaukeln, werden sie in jeder Iterationen so normalisiert, dass die Summe der Quadrate 1 ergibt:

$$\sum x_i^2 = 1 \quad \sum y_i^2 = 1$$

Die Werte für jeden Knoten konvergieren im Laufe der Iterationen.

Zu Beginn werden Hub und Authority Werte aller Knoten auf 1 gesetzt. Danach wird in jeder Iteration ein Authority und ein Hub Update durchgeführt.

Authority Update

Für jeden Knoten $v \in V$ wird der neue Authority Wert aus der Summe der Hub Werte der eingehenden Kanten gebildet:

$$x_v = \sum_{v:(u,v) \in E} y_u$$

Danach werden alle Authority Werte wie oben beschrieben normalisiert:

$$x_v = \frac{x_v}{\sum_{u \in V} x_u^2}$$

Hub Update

Für jeden Knoten $v \in V$ wird der neue Hub Wert aus der Summe der Authority Werte der Knoten gebildet, auf die v zeigt.

$$y_v = \sum_{v:(v,u) \in E} x_u$$

Danach werden alle Hub Werte wie oben beschrieben normalisiert:

$$y_v = \frac{y_v}{\sum_{u \in V} y_u^2}$$

Konvergenz

Um den Algorithmus konvergieren zu lassen, wähle ein ε und prüfe nach jeder Iteration, ob die gesamte Änderung der Authority und Hub Werte kleiner als das ε ist. Ist dies nicht der Fall, werden weitere Iterationen durchgeführt.

Alternativ kann auch eine feste Anzahl an Iterationen festgelegt werden. Sobald diese erreicht ist, endet der Algorithmus.

2.4 Graph Engine

Graph Engine ist ein verteiltes in-Memory Datenverarbeitungssystem, welches von Microsoft Research Asia entwickelt wurde [10]. Der Quellcode von Graph Engine ist quelloffen auf Github verfügbar und in C# geschrieben.

Es bietet einen verteilten Key-Value Speicher, in dem Daten gespeichert und verarbeitet werden können. Dabei ist Graph Engine sehr flexibel darin, wie die Daten gespeichert werden. Der Anwender muss selbst Schemata für die zu speichernden Daten erstellen. Dazu bietet Graph Engine die Möglichkeit, die Kommunikation unter den verschiedenen Servern mit selbst erstellten Protokollen zu koordinieren. Im Folgenden werden die verschiedenen Komponenten von Graph Engine besprochen.

2.4.1 Memory Cloud

Im Kern von Graph Engine steht die sogenannte Memory Cloud. Diese stellt einen verteilten Key-Value Speicher dar, der im Arbeitsspeicher der Maschinen liegt, um einen schnellen Zugriff zu gewährleisten.

Graph Engine verwaltet sogenannte Memory Trunks, in denen die Key-Value Paare gespeichert werden. Jeder der Trunks ist auf einer Maschine gespeichert. In der Regel gibt es mehr Trunks als Maschinen, sodass eine Maschine mehrere Trunks hält. Durch die Aufteilung in mehrere Trunks kann parallel auf Daten aus verschiedenen Trunks zugegriffen werden, ohne dass ein weiterer Lock-Mechanismus benötigt wird. Die Größe der Trunks kann manuell gewählt werden, liegt aber bei den Standardeinstellungen bei 2GB. Um zu gewährleisten, dass die Daten wiederhergestellt werden können, werden die Memory Trunks in dem verteilten Dateisystem Trinity File System (TFS) gespeichert. Das Design von TFS ist ähnlich zu Googles HDFS.

Als Schlüssel für die Werte werden 64-Bit Werte verwendet. Die Werte selbst sind beliebig große Datenblobs, die direkt im Arbeitsspeicher der Maschinen verwaltet werden.

Um einen Wert anhand des Schlüssels zu finden, werden zwei Schritte durchgeführt. Zuerst wird die Maschine gefunden, die für den jeweiligen Schlüssel verantwortlich ist. Danach wird der Schlüssel in den Memory Trunks dieser Maschine gefunden.

Im ersten Schritt wird der Schlüssel auf einen p -Bit Wert gehasht, um ein $i \in [0, 2^p - 1]$ zu erhalten. Der Schlüssel liegt demnach in Memory Trunk i . Jede Maschine besitzt eine Adressierungstabelle, die festhält, welcher Trunk auf welcher Maschine liegt.

Auf dieser Maschine muss nun der Schlüssel gefunden werden. Dafür besitzt jeder Memory Trunk eine Hashtabelle, die zu jedem Schlüssel ein Offset und die Größe des Wertes im Speicher angibt.

2.4.2 Graph Model

Graph Engine bietet ein flexibles Model, mit dem die Graphdaten modelliert werden können. Es gibt keine festgelegte Struktur und es ist den Entwicklern überlassen, Schemata für die Daten festzulegen. Hierbei hat man die Möglichkeit das Graphmodell genau an das zu lösende Problem anzupassen. Dies bietet die Chance Optimierungen zu finden und gibt eine sehr feine Kontrolle über die gespeicherten Daten.

Trinity Specification Language (TSL)

Um das Datenmodell zu definieren, benutzt Graph Engine eine eigene Sprache, die Trinity Specification Language (TSL). Mit dieser werden sowohl die Schemas für Daten, als auch Server Protokolle und Schnittstellen erstellt.

TSL bietet die Möglichkeit Zellen zu definieren, welche im Betrieb als Werte im Key-Value Speicher abgelegt werden können. Zellen können Grunddatentypen wie int, float, string etc. sowie Listen von Werten speichern. Um komplexere Werte darzustellen, können auch in TSL erstellte structs verwendet werden. Mit diesen Möglichkeiten lassen sich sehr viele Datenstrukturen in TSL modellieren.

Ein Beispiel für einen simplen Graphen ist in 2.1 dargestellt. In diesem Beispielgraph hat jeder Knoten einen Wert und eine Liste an Kanten. Die Kanten selbst haben ein Gewicht sowie einen Verweis auf die ID des Knoten, auf den sie zeigen.

Listing 2.1: Beispiel für eine in TSL definierte Graphenstruktur

```
1 struct Edge {
2     float Weight;
3     long Link;
4 }
5
6 cell struct GraphNode {
7     int Value;
8     List<Edge> Edges;
9 }
```

Graph Engine hat einen eigenen Compiler für TSL, der die TSL Dateien in C# Quellcode umwandelt. So werden aus den Definitionen Schnittstellen generiert, um die entsprechenden Zellen in Graph Engine zu erstellen, zu verändern oder zu löschen.

2.4.3 Computation Engine

Um Berechnungen durchzuführen, besteht ein Graph Engine Cluster aus drei verschiedenen Komponenten, die unterschiedliche Aufgaben übernehmen.

Server

Die Server in einem Graph Engine Cluster haben zwei Aufgaben: Zum einen speichern sie die Memory Trunks in ihrem Arbeitsspeicher, zum anderen führen sie Berechnungen auf den gespeicherten Daten durch. Um diese Berechnungen durchzuführen, werden in der Regel Nachrichten mit anderen Graph Engine Komponenten ausgetauscht. Insbesondere die Kommunikation zwischen den Servern selbst ist oft notwendig, da jeder Server nur eine Sicht auf seine lokal gespeicherten Daten hat.

Proxy

Proxies speichern selber keine Daten, können aber Nachrichten austauschen und Berechnungen durchführen. Sie können als Bindeglied zwischen Client und Server genutzt werden. So können sie z.B. von Clients geschickte Anfragen auf die Server aufteilen und deren Berechnungen koordinieren oder die einzelnen Ergebnisse aggregieren. Insbesondere in aufwändigeren Algorithmen kann eine Proxy den Ablauf kontrollieren und Entscheidungen wie Abbruchsbedingungen prüfen.

Client

Clients laufen auf der Maschine des Benutzers, der mit dem Graph Engine Cluster interagieren will. Clients senden Anfragen an die Server oder Proxies und erhalten die entsprechenden Ergebnisse zurück. Sie halten keine Daten und führen in der Regel auch keine Berechnungen durch, womit es keine großen Hardwareanforderungen an die Maschine gibt, auf der ein Client läuft.

Protokolle

Server, Proxies und Clients kommunizieren über Nachrichten, die sie einander schicken. Graph Engine unterstützt hierbei drei verschiedene Arten von Protokollen.

Synchrone Protokolle sind ähnlich zu synchronen Funktionsaufrufen, die auf einer anderen Maschine stattfinden. Sie blockieren die weitere Ausführung, bis eine Antwort erhalten wurde. Ein Synchrones Protokoll kann sowohl in der Anfrage, als auch in der Antwort Daten mitsenden. So kann beispielsweise die Liste von relevanten Schlüsseln übergeben werden und mit deren Gesamtsumme der Werte geantwortet werden.

Asynchrone Protokolle blockieren die Ausführung des Absenders nicht. Der Empfänger antwortet beim Erhalten der Anfrage sofort mit der Bestätigung, dass diese erhalten wurde. In einem Synchronen Protokoll können lediglich in der Anfrage Werte mitgegeben werden. Der Empfänger startet einen Thread, der die Anfrage abarbeitet. Der Absender erfährt nicht, wann die Anfrage vollständig bearbeitet wurde.

HTTP Protokolle bieten Clients die Möglichkeit eine RESTful Version der Synchronen Protokolle über HTTP zu nutzen. Graph Engine erstellt automatisch die Endpunkte, an denen auf Anfragen gewartet wird. So wird z.B. für ein Protokoll `MyHTTPProtocol` am Endpunkt

`http://example.com/MyHttpProtocol` gewartet. Die Anfrage und Antwort werden jeweils in JSON Strukturen übergeben. HTTP Protokolle werden nicht für Server zu Server Kommunikation genutzt, da sie deutlich weniger effizient sind als die Synchronen und Asynchronen Protokolle.

TSL

Die Kommunikationschemata von Servern und Proxies werden in TSL definiert. In 2.2 ist ein Beispiel für einen Server, der ein Synchrones Ping Protokoll unterstützt, dargestellt.

Listing 2.2: In TSL definiertes Ping Protokoll

```
1 struct PingMessage {
2     string Content;
3 }
4
5 protocol SynEchoPing {
6     Type: Syn;
7     Request: PingMessage;
8     Response: PingMessage;
9 }
10
11
12 server PingServer {
13     protocol SynEchoPing;
14 }
```

Wie schon bei den Zellen werden die Server und Protokolldefinitionen von dem TSL Compiler in C# Code übersetzt. Für Server und Proxy Definitionen werden abstrakte Klassen erstellt, in denen jeweils Methoden für die benötigten Protokolle implementiert werden müssen. Es werden zudem Methoden generiert, um die definierten Anfragen an den Server oder die Proxy zu erstellen und diese zu senden.

2.4.4 Datenzugriff

Die Daten der Zellen liegen im Arbeitsspeicher der Maschinen als Datenblobs. Um auf diese komfortabel zuzugreifen, können die Daten in ein C# Objekt serialisiert werden; das ist jedoch sehr langsam. Schnelleren Zugriff hat man, indem man die Daten direkt im RAM manipuliert. Das ist jedoch deutlich schwieriger, da man das Speicherlayout der jeweiligen Daten kennen und entsprechende Zeiger-Arithmetik betreiben muss. Graph Engine löst diesen Konflikt, indem es aus der TSL Zellendefinition eine Zugriffsklasse erzeugt. Diese übersetzt die Lese- und Schreibzugriffe auf die Werte der Zelle auf die entsprechenden Operationen im Arbeitsspeicher. So lässt sich mit der Zugriffsklasse sowohl komfortabel, als auch effizient arbeiten.

Listing 2.3: Bearbeitung einer Zelle mithilfe der Zugriffsklasse

```
1 using (var node = Global.LocalStorage.UseNode(cellId, accessOptions)) {
2     int value = node.Value;
```

```
3 |     node.Value = 5;
4 | }
```

2.5 Andere Multilayer Graph Systeme

Im Folgendem werden verschiedene andere Systeme betrachtet, mit denen Multilayer Graphen analysiert werden können.

2.5.1 MuxViz

Das Analysetool MuxViz wurde von De Domenico, M. und Porter, M. A. und Arenas, A. in ihrer Arbeit "MuxViz: a tool for multilayer analysis and visualization of networks vorgestellt" [3]. MuxViz ist ein open-source Projekt, welches es ermöglicht Multilayer Graphen mit verschiedenen Algorithmen zu analysieren und visualisieren. Es nutzt für die Berechnungen R sowie GNU Octave und bietet mit einem modularen Aufbau die Möglichkeit, das Nutzer eigene Funktionalität hinzufügen.

MuxViz bietet eine grafische Nutzeroberfläche, welche genutzt werden kann, um Graphen zu laden, Algorithmen auszuführen und Graphen sowie Ergebnisse zu visualisieren. Die Benutzeroberfläche ist Webbasiert, was ermöglicht, dass die tatsächlichen Berechnung entweder lokal oder auf einem entfernten Server durchgeführt werden. Dabei gibt es eine große Auswahl an Algorithmen und Statistiken, die auf den Graphen angewandt werden können.

2.5.2 Multilayer Networks Library for Python

Die Multilayer Networks Library for Python (Pymnet) wurde von Mikko Kivelä 2015 veröffentlicht. Die in Python geschriebene Bibliothek ermöglicht es mit Multilayer Netzwerke in Python zu arbeiten. Sie unterstützt das Laden und Manipulieren von Multilayer Netzwerken und bietet eine Reihe an Algorithmen zur Analyse der Netzwerke. Zudem können Netzwerke mithilfe von Matplotlib und D3 visualisiert werden.

Kapitel 3

Architektur

Das Multilayer Graph System setzt sich aus mehreren Komponenten zusammen, die miteinander arbeiten, um die gewünschten Anforderungen zu gewährleisten. Dabei wird GE genutzt, um die Graph Daten effizient im Arbeitsspeicher der Server zu verwalten und die Kommunikation der einzelnen Komponenten zu steuern. Das TSL-Model und eine geteilte Bibliothek stellen die Basis für die anderen Komponenten.

In der Architektur ist der Client für das Senden von Anfragen, sowie das Ausführen eines Algorithmus oder das Laden eines Graphen an die Proxy zuständig. Die Proxy verarbeitet die Anfrage und koordiniert ggf. die Berechnungen der Server. Die Server kommunizieren, falls nötig, für ihre Berechnungen untereinander, um Daten über entfernte Knoten anzufragen oder zu senden. Sobald die Anfrage fertig bearbeitet ist, schickt die Proxy eine Antwort mit den entsprechenden Ergebnissen an den Client. Eine Darstellung dieser Arbeitsweise ist in Abbildung 3.1 dargestellt.

Der Client, Proxy und Server werden als C# Anwendungen umgesetzt.

3.1 Model

Das TSL Model dient als Basis für den Rest der Anwendung. Es definiert, wie Knoten gespeichert werden und wie Client, Server und Proxy miteinander kommunizieren können. Die einzelnen Komponenten müssen die von GE automatisch generierten Klassen implementieren.

Es wird die grundlegende Graphenstruktur definiert, die genutzt wird, um Multilayer Graphen darzustellen. Knoten speichern hierbei jeweils ihre ID, zu welchem Layer sie gehören und ihre Liste an ausgehenden Kanten. Dazu kommen Daten, die gebraucht werden, um Algorithmen auszuführen. Diese können z.B. der aktuelle PageRank Wert des Knoten sein.

3.2 Lib

Die geteilte Bibliothek besteht aus zwei Komponenten. Sie enthält den von TSL generierten Code und macht es so dem Client/Proxy/Server möglich darauf zuzugreifen. Der generier-

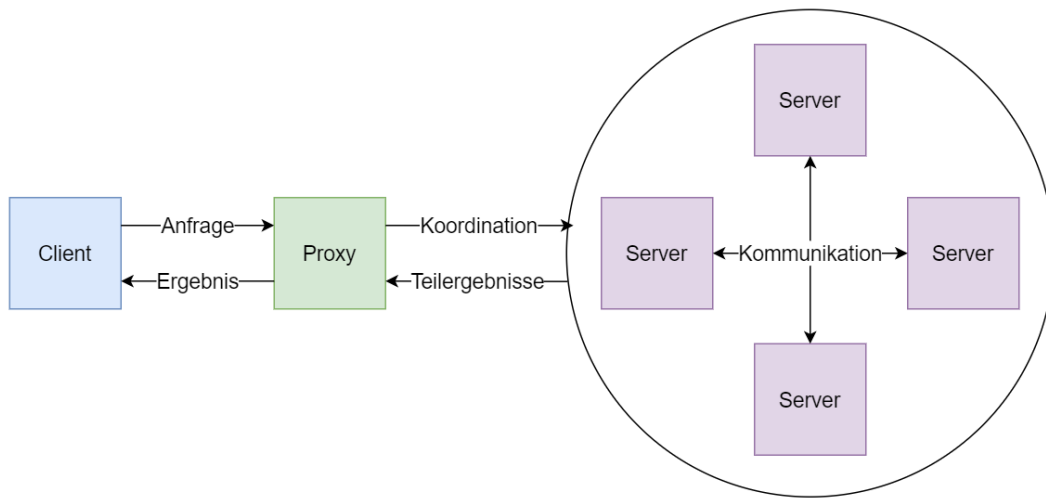


Abbildung 3.1: Aufbau des Multilayer Clusters

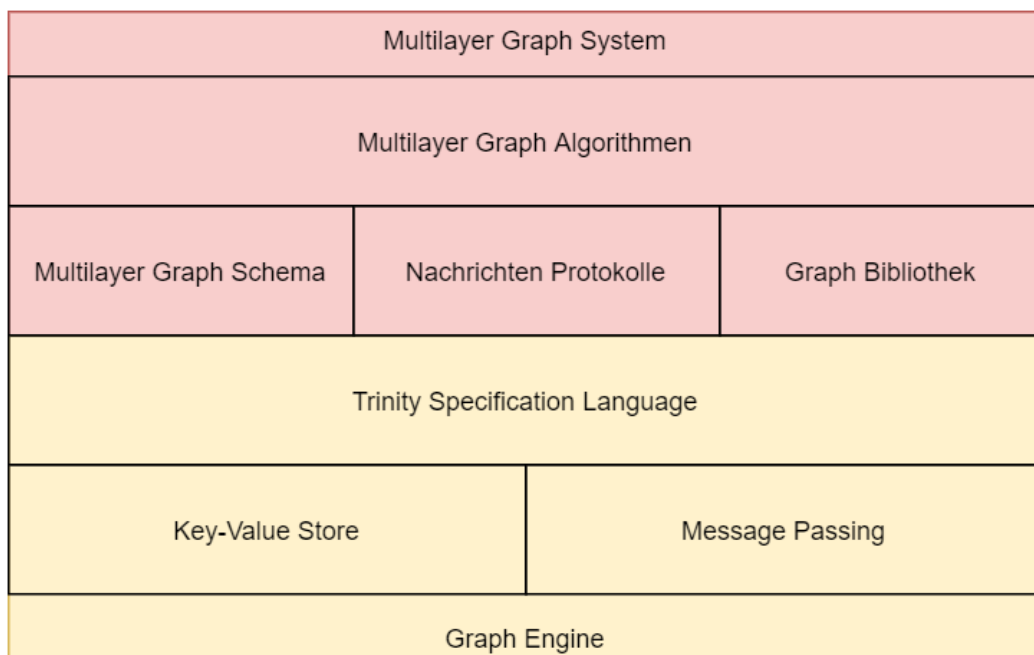


Abbildung 3.2: Schichten des Multilayer Graph Systems.

te Code enthält auch die abstrakten Klassen für Proxy und Server. Diese werden in den entsprechenden Komponenten implementiert.

Außerdem enthält sie eine Sammlung von Funktionen, die alle Projekte nutzen können. Insbesondere besitzt sie ein Interface, um mit dem Graphen zu interagieren und z.B. neue Knoten anzulegen oder einem Knoten neue Kanten hinzuzufügen. Dazu kommt die Funktionalität, Ergebnisse von Algorithmen ausgeben zu lassen, da dies sowohl auf dem Client als auch auf der Proxy möglich ist.

3.3 Client

Der Client stellt die Schnittstelle zwischen dem Anwender und dem Multilayer System dar. Der Client kann Anweisungen des Anwenders auf zwei Arten empfangen. Zum einen wird ein Kommandozeilen-Interface bereitgestellt, mit dem der Anwender interagieren kann. Zum anderen kann dem Client eine Batch-Datei mit Anweisungen übergeben werden, welche nacheinander ausgeführt werden. Hierbei besteht auch die Möglichkeit zwischen dem Interaktiven und dem Batch Modus zu wechseln.

Die Anweisungen werden interpretiert und in Anfragen an die Proxy übersetzt, welche die Anfragen dann abarbeitet.

3.4 Proxy

Die Proxy dient als Bindeglied zwischen dem Client und den Servern. Sie nimmt Anfragen vom Client an und sorgt dafür, dass diese ausgeführt werden. Dabei koordiniert sie die Ausführung der verschiedenen Algorithmen und sendet die nötigen Anfragen an die Server. Wenn es nötig ist, kann gewartet werden, bis alle Server die Anfrage abgearbeitet haben. Die Server können dabei auch ein Ergebnis zurücksenden, welches die Proxy weiter verwenden kann. Ein häufiger Fall ist hierbei, dass die Ergebnisse der einzelnen Server aggregiert werden.

Abhängig von der Client Anfrage ist die Proxy auch für das Messen der Laufzeit und das Bilden der Ergebnisse verantwortlich. Die Ergebnisse können im gewünschten Format entweder direkt auf der Proxy ausgegeben oder zurück an den Client gesendet werden.

3.5 Server

Die Server erfüllen zwei Aufgaben. Sie verwalten die Graph Daten in GE und warten darauf, dass sie Anweisungen von der Proxy bekommen und führen auf ihre Anweisung Berechnungen durch. Bei diesen Berechnungen kümmert sich jeder Server um die eigenen lokal gespeicherten Knoten. Die Servern können aber auch miteinander kommunizieren, wenn sie die Daten entfernter Knoten benötigen oder die Daten entfernter Knoten aktualisieren müssen. Hat ein Server die angeforderte Aufgabe beendet, kann er dies der Proxy melden. Dabei kann, falls nötig, auch ein Ergebnis mitgesendet werden.

3.5.1 Laden

Das Laden des Graphen findet verteilt über alle Server statt. Jeder Server liest die Kantendatei Zeile für Zeile und lädt die Kanten und Knoten, für die er verantwortlich ist. Um herauszufinden, ob ein Server für einen Knoten verantwortlich ist wird aus der ID und dem Layer des Knoten ein 64-Bit Hash-Wert gebildet. Dieser wird als Schlüssel für den Key-Value Store von GE verwendet. Nun prüft der Server, wie in 2.4.1 beschrieben, ob er für den Knoten verantwortlich ist.

Kapitel 4

Implementierung

4.1 Graph Engine

Da Graph Engine eine wichtige Rolle im MultiLayer Graph System spielt, werden im Folgenden einige Dinge betrachtet, die wichtig für die Verwendung von Graph Engine sind. Dies ist insbesondere interessant, da die aktuelle Version 2 von GraphEngine sich zurzeit noch in Entwicklung befindet und dadurch die Verwendung aufgrund fehlender Dokumentation erschwert wird.

4.1.1 Einrichtung

Graph Engine wurde in C# entwickelt und verwendet den dafür verbreiteten Paket-Manager NuGet, mit dem Pakete verwaltet werden können. Die aktuellste Version des GraphEngine.Core Pakets auf NuGet ist 1.0.8467 vom 19.08.2016 [2]. Da in dieser Arbeit die Version 2.0.9912 verwendet wird, ist es nötig, dass GraphEngine.Core Paket selber vom Quellcode aus zu bauen. Die genauen Schritte zum Einrichten von Graph Engine sind im Anhang A.1 zu finden.

Konfigurationsdatei

Graph Engine benutzt eine .xml Konfigurationsdatei, um das Cluster aus Proxy und Servern zu definieren. In der Datei können zudem weitere Einstellungen, wie Dateipfade oder Logging, für die Server gemacht werden (siehe 4.1).

4.1.2 Datenzugriff

Graph Engine bietet mehrere Arten auf die Daten von Zellen zuzugreifen. Dabei ist es wichtig zu verstehen, wie die verschiedenen Arten funktionieren und wie deren Performance ist. Die erste Möglichkeit ist auf Zellen von entfernten Maschinen zuzugreifen. Dabei werden die Funktionen der `Global.CloudStorage` verwendet. Dies ist die langsamste Art des

Listing 4.1: Beispiel Konfigurationsdatei für ein Cluster mit einer Proxy und zwei Servern

```
1 <Trinity ConfigVersion="2.0">
2   <Cluster>
3     <Proxy Endpoint="node52:8133" LogDirectory="D:\log-dir" LoggingLevel="Info" />
4     <Server Endpoint="node55:8133" />
5     <Server Endpoint="node63:8133" />
6   </Cluster>
7 </Trinity>
```

Zugriffs, da für jeden Zellenzugriff eine Netzwerknachricht an die entfernte Maschine geschickt werden muss.

Es gibt zwei weitere Möglichkeiten auf lokale Zellen mittels `Global.LocalStorage` zuzugreifen. Die erste ist `LoadCell()`. Diese Methode lädt die Daten einer Zelle aus dem von Graph Engine verwalteten Arbeitsspeicher in ein Objekt des C# Heaps. Dabei wird jedes Mal ein neues Objekt erzeugt und die Werte in dieses kopiert. Hiermit kann nur auf die Daten der Zelle zugegriffen, diese aber nicht verändert werden.

Die andere Methode ist der direkte Zugriff über `UseCell()`. Dabei wird direkt auf die im Arbeitsspeicher vorhandene Zelle zugegriffen, ohne dass diese kopiert werden muss (siehe 2.4.4). So können auch die Werte der Zelle direkt verändert werden. Diese Methode ist damit von der Performance her die beste Art auf Zellen zuzugreifen und sollte, wenn möglich, immer verwendet werden.

4.1.3 Schwierigkeiten

Da sich die Version 2 von GraphEngine noch in der Entwicklung befindet, gibt es einige Schwierigkeiten, die bei der Entwicklung von Anwendungen mit Graph Engine entstehen. Hauptsächlich ist hierbei die kaum vorhandene Dokumentation für Version 2 problematisch. Viele Änderungen zwischen den beiden Versionen sind nicht dokumentiert. So wurden zum Beispiel das Attribut jeder Zelle, welches die ID der Zelle beinhaltet, von `cell.CellID` zu `cell.CellId` umbenannt. Ebenfalls wurde die Bezeichnung für entfernte Server von `Server` zu `Partition` geändert. Solche Änderungen machen das Entwickeln von Anwendungen mit GraphEngine zeitaufwändig, da immer wieder im Quellcode nachgeschaut werden muss wie bestimmte Dinge heißen.

Dazu sind viele Funktionen auch wenig oder gar nicht dokumentiert. So zum Beispiel die Funktion `Global.CloudStorage.BarrierSync(int)`, die es erlaubt, die Server an einer Barriere warten zu lassen, bis alle diese erreicht haben. Dies macht es schwierig alle Möglichkeiten von GraphEngine auszunutzen und erfordert regelmäßiges Nachforschen im Quellcode, ob gewisse Funktionen vorhanden sind und was diese im Detail machen.

Zudem sind alle Beispielanwendungen, die es im GraphEngine Repository gibt, noch auf Version 1 und einige dadurch nicht kompatibel mit der neueren Version.

4.2 Implementierung

Im Folgendem wird auf die Implementierungsdetails der einzelnen Komponenten eingegangen. Hierbei wird der grundlegende Aufbau der Komponenten erklärt. Zudem wird auf einige Optimierungen und Designentscheidungen eingegangen.

4.2.1 Model

Um Multilayer Graphen in Graph Engine abzubilden wird pro Knoten und Layer eine Zelle erstellt. Die Zellen speichern hierbei ihre ID, ihren Layer, eine Liste der ausgehenden Kanten und die Daten die für die Algorithmen notwendig sind. Die Kanten müssen dabei speichern, auf welche ID in welchem Layer sie zeigen.

Listing 4.2: TSI Definition von Multilayer Knoten und Kanten.

```
1 struct Edge {
2     long StartId;
3     int StartLayer;
4     long DestinationId;
5     int DestinationLayer;
6     float Weight;
7 }
8
9 // We create a cell for each layer a node is on
10 cell struct Node {
11     long Id;
12     int Layer;
13     PageRankData PageRankData;
14     HITSDData HITSDData;
15     DegreeData DegreeData;
16     List<Edge> Edges;
17 }
```

Die Proxy wird mit allen unterstützten Protokollen definiert.

Die Proxy muss von den Servern benachrichtigt werden können, wenn diese asynchrone Aufgaben erfüllt haben. Dabei kann auch ein Ergebnis zurückgegeben werden. Dafür wird ein PhaseFinished Protokoll erstellt. Die Server nutzen das Protokoll, um der Proxy eine PhaseFinishedMessage zu senden.

Diese Nachricht enthält die Phase, die beendet wurde und eine Liste an Strings, welche das Ergebnis der jeweiligen Phase darstellt. Es wird hier aus zwei Gründen eine Liste an Strings verwendet. Zum einem gibt es Algorithmen, die ein Ergebnis pro Layer des Graphen zurück gegeben wollen. Dabei stellt jedes Element der Liste einen Layer des Graphen dar. Zum anderen werden Strings verwendet, da die Ergebnisse verschiedenen Datentypen haben können, welche allerdings alle in einem String dargestellt werden können. So gibt ein Algorithmus, der Knoten zählt nur Integer Werte zurück, wohingegen ein Algorithmus der PageRank Werte zurückgibt Double werte benutzt. Die Phasen werden in einer separaten TSL Datei verwaltet und als Enum abgespeichert.

Um von der Proxy Ergebnisse zurück an den Client geben zu können, gibt es das Konstrukt AlgorithmResult. Dieses beinhaltet ein Ergebnis und die Laufzeit des Algorithmus.

Listing 4.3: Definition für die Proxy Protokolle.

```

1
2 struct AlgorithmResult {
3     string Name;
4     DateTime StartTime;
5     DateTime EndTime;
6     List<List<string>> ResultTable;
7 }
8
9 struct StandardAlgorithmMessage {
10     AlgorithmOptions AlgorithmOptions;
11     OutputOptions OutputOptions;
12 }
13
14 struct PhaseFinishedMessage {
15     List<string> Result;
16     Phases Phase;
17 }
18
19 protocol PhaseFinished {
20     Type: Asyn;
21     Request: PhaseFinishedMessage;
22     Response: void;
23 }
24
25 proxy MultiLayerProxy {
26     // Base Protocols
27     protocol PhaseFinished;
28     // Data Load Protocols
29     protocol LoadGraphProxy;
30     ...
31     // EgoNetwork
32     protocol EgoNetworkProxy
33 }

```

Listing 4.4: Definition der einzelnen Phasen.

```

1 // This is a list of all the phases for the different algorithms
2 // These will be used in the proxies PhaseFinished protocol to identify
3 // which phase has been finished.
4 enum Phases {
5     // DataLoad Phases
6     DataLoad = 0,
7     // Stats Phases
8     NodeCount = 1,
9     EdgeCount = 2,
10    ...
11    // EgoNetwork
12    EgoNetwork = 17
13 }

```

Die einzelnen Algorithmen definieren ihre benötigten Protokolle und Nachrichten in eigenen Dateien. Diese sind jeweils auf den jeweiligen Algorithmus zugeschnitten. Die Server Definition zieht alle diese Protokolle zusammen, sodass die entsprechenden abstrakten Methoden in der Serverklasse generiert werden.

4.2.2 Lib

Die geteilte Bibliothek stellt eine statische Klasse `Graph` zur Verfügung, die von den anderen Komponenten genutzt wird. Die Klasse ermöglicht es mit dem in Graph Engine gespeicherten Graph zu interagieren, ohne die Implementierungsdetails zu kennen. So kann auf Knoten zugegriffen werden, ohne zu wissen, wie ID und Layer in eine Graph Engine ZellenID übersetzt werden. Dazu kommen Funktionen, um Knoten zu erstellen und diese zu verändern.

Ausgabe

Die Bibliothek stellt die Funktionalität zur Ausgabe von Ergebnissen zur Verfügung. Die verschiedenen Arten Ergebnisse auszugeben implementieren alle das Interface `IOutputWriter`, welches die Funktion `void WriteOutput(AlgorithmResult algorithmResult)` hat. Es gibt drei verschiedene Ausgabearten:

- `None`, gibt nichts aus
- `Console`, gibt das Ergebnis in der Konsole aus
- `CSV`, gibt das Ergebnis in einer `.csv` Datei aus

4.2.3 Client

Der Client kann über die Kommandozeile gestartet und bedient werden. Mit dem Argument `'interactive'` wird eine interaktive Sitzung gestartet, in der der Nutzer Befehle eingeben kann, die die Proxy dann ausführt. Im `'batch'` Modus wird zudem eine Datei mitgegeben, die Anweisungen erhält.

Der Client verwaltet eine Liste an ausführbaren Kommandos, die vom Nutzer abgerufen werden können. Dazu speichert er die aktuellen Einstellungen zur Ausführung von Funktionen und Ausgabe der Ergebnisse. Eine Übersicht über die Struktur ist in Abbildung 4.1 gezeigt.

Kommandos

Die einzelnen Funktionen des Clients sind in Kommandos aufgeteilt. Diese implementieren alle das Interface `ICommand`. Der Client verwaltet ein Dictionary mit allen `ICommands` die er kennt. Das Interface bietet Zugang zu Informationen über das Kommando, wie z.B. das Keyword, um es aufzurufen oder welche Argumente das Kommando benötigt. Dazu gibt es drei Interface Funktionen:

- `VerifyArguments(string[] arguments)`, prüft ob die Liste der Argumente die der Nutzer dem Kommando gegeben hat legitime Argumente für das Kommando sind.

-
- `ApplyArguments(string[] arguments)`, wendet die Argumente auf das Kommando an sodass es wenn es danach ausgeführt wird die Werte benutzt.
 - `Run()`, führt das jeweilige Kommando aus.

Da die Methoden, die nur Informationen zurückgeben, sowie das Verifizieren der Argumente für alle Kommandos gleich funktioniert, gibt es eine abstrakte Klasse `Command`, welche diese bereits implementiert. Die Kommandos erben von dieser Klasse und müssen nur noch `ApplyArguments(string[] arguments)` und `Run()` selbst implementieren. Die Kommandos füllen die Informationen über sich selbst jeweils in ihrem Konstruktor aus. Diese sind:

- `Name`, der Name des Kommandos
- `Keyword`, das Keyword um es aufzurufen
- `Description`, eine kurze Beschreibung was das Kommando macht
- `Arguments`, eine Liste welche die festhält welche Datentypen die Argumente haben
- `ArgumentsDescription`, eine Beschreibung für die einzelnen Argumente

Die Verifizierung der Argumente findet in zwei Schritten statt. Zuerst wird die Anzahl der gegebenen Argumente mit der Anzahl der Elemente von 'Arguments' verglichen. Im zweiten Schritt wird für jeden Eintrag in `Arguments` geprüft, ob das gegebene Argument dem jeweiligen Typ entspricht.

Interaktiver Modus

Der Batch Modus wird mit dem Argument 'interactive' gestartet. Sobald der Client die Verbindung zum Graph Engine Cluster aufgebaut hat, kann der Nutzer Kommandos eingeben.

```
1 $ dotnet run interactive
```

Batch Modus

Der Batch Modus wird mit dem Argument 'batch' gestartet. Es muss zudem ein weiteres Argument gegeben werden, welches der Pfad zu der batch Datei ist.

Diese Datei ist eine normale Textdatei, die pro Zeile ein Kommando enthält. Der Client arbeitet die Datei Zeile für Zeile ab und führt die Kommandos aus. Die Verarbeitung verhält sich genau so, als hätte ein Nutzer das Kommando im interaktiven Modus eingegeben. Wird das Ende der Datei erreicht, wird das Client Programm beendet.

```
1 $ dotnet run batch path\to\batch\file
```

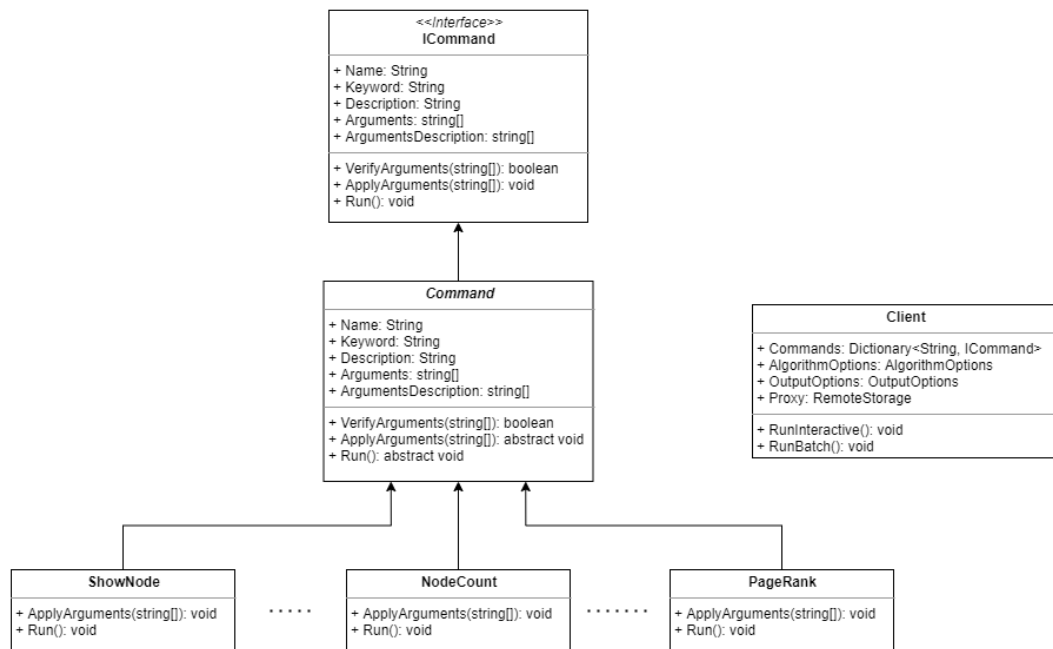


Abbildung 4.1: Klassendiagramm für die Client-Anwendung

4.2.4 Proxy

Die in den TSL Dateien definierte Proxy `MultiLayerProxy` wird vom TSL Compiler in eine abstrakte Klasse `MultiLayerProxyBase` mit abstrakten Request Handlern kompiliert. Diese wird hier von 'MultiLayerProxyImpl' implementiert.

Da die `MultiLayerProxyImpl` alle Requests für alle Protokolle handhaben muss, wird sie über mehrere Dateien als partielle Klasse implementiert. Dies vermag die sonst sehr groß werdende Klasse in kleine Teile aufzuteilen, sodass eine Datei jeweils nur für einen Algorithmus verantwortlich ist. Die Abbildung 4.2 gibt eine Übersicht über den Aufbau der Proxy-Anwendung.

BaseProxy

In 'MultiLayerBaseProxy.cs' wird die grundlegende Funktionalität der Proxy definiert, die nicht zu einem einzelnen Algorithmus gehört. Dies ist zum einen die Möglichkeit Algorithmen zu starten und deren Ergebnisse auszugeben, zum anderem wird hier die Möglichkeit geboten auf Antworten der einzelnen Server, sobald diese eine Phase beendet haben, zu warten. Hat ein Server eine Phase beendet, sendet er an die Proxy einen `PhaseFinished` Request, der die Phase enthält, die er beendet hat und wenn nötig sein lokales Ergebnis. Die `MultiLayerBaseProxy` handhabt diesen Request und zählt wieviele Server eine jeweilige Phase schon beendet haben und sammelt deren Ergebnisse. Hierfür werden drei Elemente verwendet:

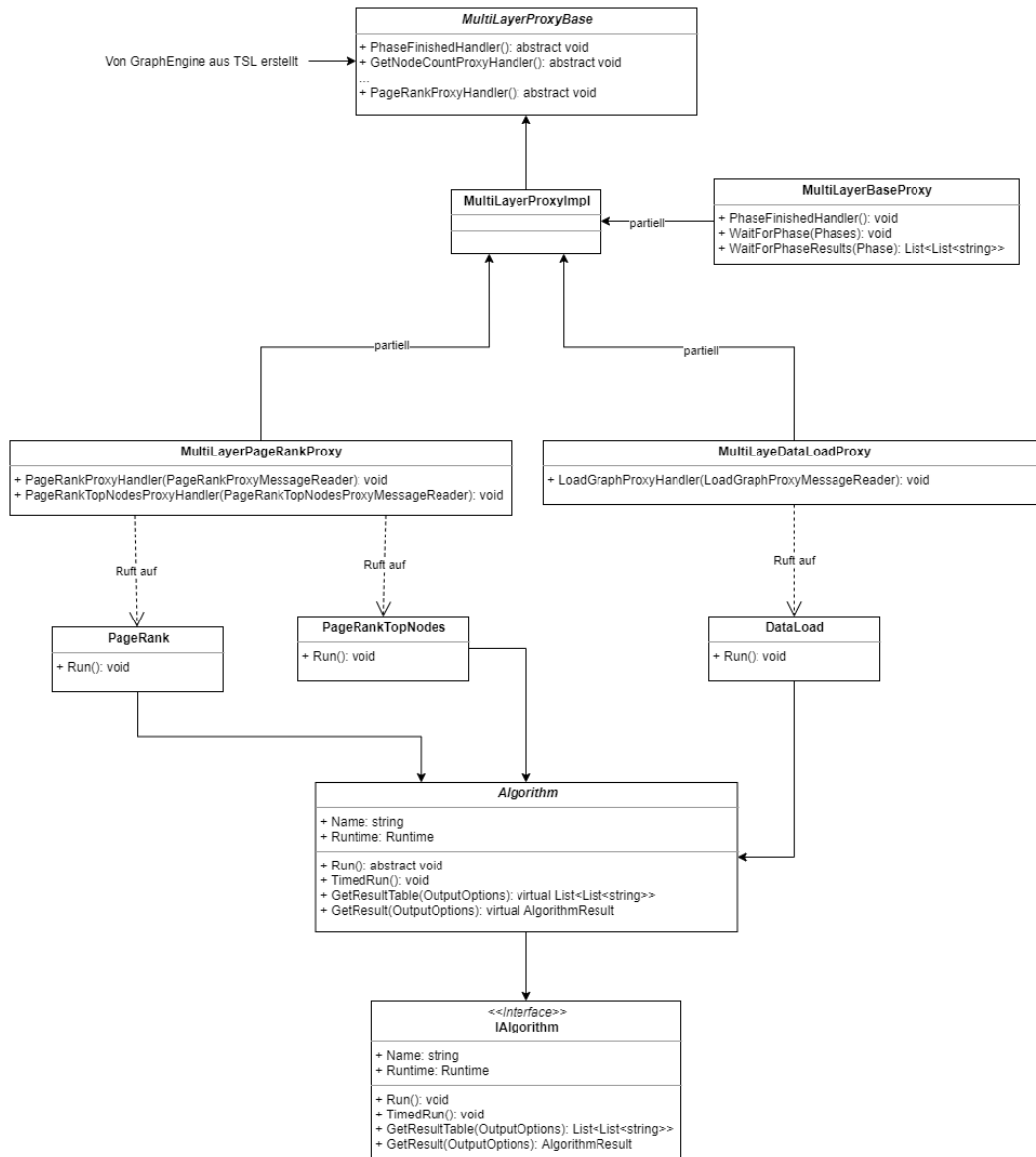


Abbildung 4.2: Klassendiagramm für die Proxy-Anwendung

- Dictionary<Phases, int> phaseFinishedCount, um zu zählen ,wieviele Server bereits eine bestimmte Phase abgeschlossen haben.
- List<List<string>> phaseResults, um die Ergebnisse der einzelnen Server zu aggregieren
- object phaseFinishedCountLock, dient als Lock, um die Operationen an den anderen beiden Variablen zu schützen

Es wird solange gewartet, bis die Anzahl an Servern, die eine Phase als beendet gemeldet haben, der Anzahl aller Server entspricht.

Listing 4.5: Implementierung des Mechanismus, um auf Phasen zu warten.

```

1 public override void PhaseFinishedHandler(PhaseFinishedMessageReader request) {
2     // Lock the phaseFinishedCount to avoid lost updates.
3     lock (phaseFinishedCountLock) {
4         phaseResults.Add(request.Result);
5         phaseFinishedCount[request.Phase]++;
6     }
7 }
8
9 private void WaitForPhaseAnswers(Phases phase) {
10     SpinWait wait = new SpinWait();
11
12     while (phaseFinishedCount[phase] != Global.ServerCount) {
13         wait.SpinOnce();
14     }
15
16     lock (phaseFinishedCountLock) {
17         phaseFinishedCount[phase] = 0;
18     }
19 }

```

Die Proxy bietet den einzelnen Algorithmen damit die Möglichkeit, auf Ergebnisse zu warten. Außerdem gibt es mehrere Methoden, die es erlauben, die Ergebnisse direkt im gewünschten Datentyp zu erhalten.

Algorithmen

Um die Ausführung, Performance Messung und Ausgabe von Ergebnissen einheitlich zu halten, gibt es das Interface IAlgorithm, welches die Grundfunktionen der Algorithmen definiert(siehe Abbildung 4.2). Es besitzt vier Methoden:

- Run(), führt den Algorithmus aus
- TimedRun(), führt den Algorithmus aus und misst die Laufzeit
- List<List<string>> GetResultTable(OutputOptions outputOptions), gibt das Ergebnis in Tabellenform zurück

-
- `AlgorithmResult GetResult(OutputOptions outputOptions)`, gibt das komplette Ergebnis inklusive Laufzeit zurück

Das Interface wird von der abstrakten Klasse `Algorithm` implementiert. In dieser Klasse bleibt die `Run()` Methode abstrakt. Die Klassen für die einzelnen Algorithmen erben von `Algorithm` und implementieren entsprechend ihrer Anforderungen die `Run()` Methode. Erhält die Proxy eine Anfrage einen bestimmten Algorithmus auszuführen ruft der entsprechende Requesthandler den angefragten Algorithmus auf.

4.2.5 Server

Wie schon bei der Proxy wird der in TSL definierte Server `MultiLayerServer` zu der abstrakten Klasse `MultiLayerServerBase` kompiliert. Diese hat abstrakte Methoden für die Protokollhandler. Die `MultiLayerServerBase` wird von der Klasse `MultiLayerServerImpl` implementiert, welche auch wieder zur besseren Übersicht in einzelne partielle Klassen aufgeteilt wird. Diese partiellen Teile sind in die jeweiligen Algorithmen und eine Basisklasse `MultiLayerBaseServer` aufgeteilt. Der `MultiLayerBaseServer` bietet den Algorithmen Methoden, um der Proxy mitzuteilen, dass sie eine Phase beendet haben. Der Aufbau der Server-Anwendung ist in Abbildung 4.3 dargestellt.

Laden

Das Laden des Graphen findet verteilt über alle vorhandenen Server statt. Die Kanten werden von einer Kantendatei geladen, die je eine Kante pro Zeile speichert. Das genaue Format, wie eine Kante in dieser Kantendatei vorhanden ist, kann verschieden sein. Wichtig ist das jede Kante die Informationen enthält, von welchem Knoten und Layer zu welchem Knoten und Layer sie zeigt.

Jeder Server geht die Kantendatei Zeile für Zeile durch und puffert diese in einer Liste. Die Liste wird solange gefüllt, bis der Knoten wechselt, von dem die Kanten ausgehen. Sobald der Knoten wechselt, wird die gepufferte Liste von Kantenzeilen an einen Threadpool übergeben, der die Kantenzeilen lädt und den Knoten in Graph Engine speichert.

Um mehrere Formate an Kantenzeilen zu unterstützen, gibt es ein Interface `IEdgeLoader`. Dieses stellt drei Methoden zur Verfügung die implementiert werden müssen.

- `Edge LoadEdge(string line)`, lädt eine Kante aus einer Zeile
- `long GetId(string line)`, liest die Knoten ID aus einer Zeile aus
- `int GetLayer(string line)`, liest den Layer aus einer Zeile aus

Die Methoden `GetId` und `GetLayer` sind nötig, damit während des Pufferens der Zeilen festgestellt werden kann, wann der Ursprungsknoten der Kanten wechselt. Für alle Formate, die unterstützt werden sollen, muss nun eine Klasse erstellt werden, die das Interface implementiert.

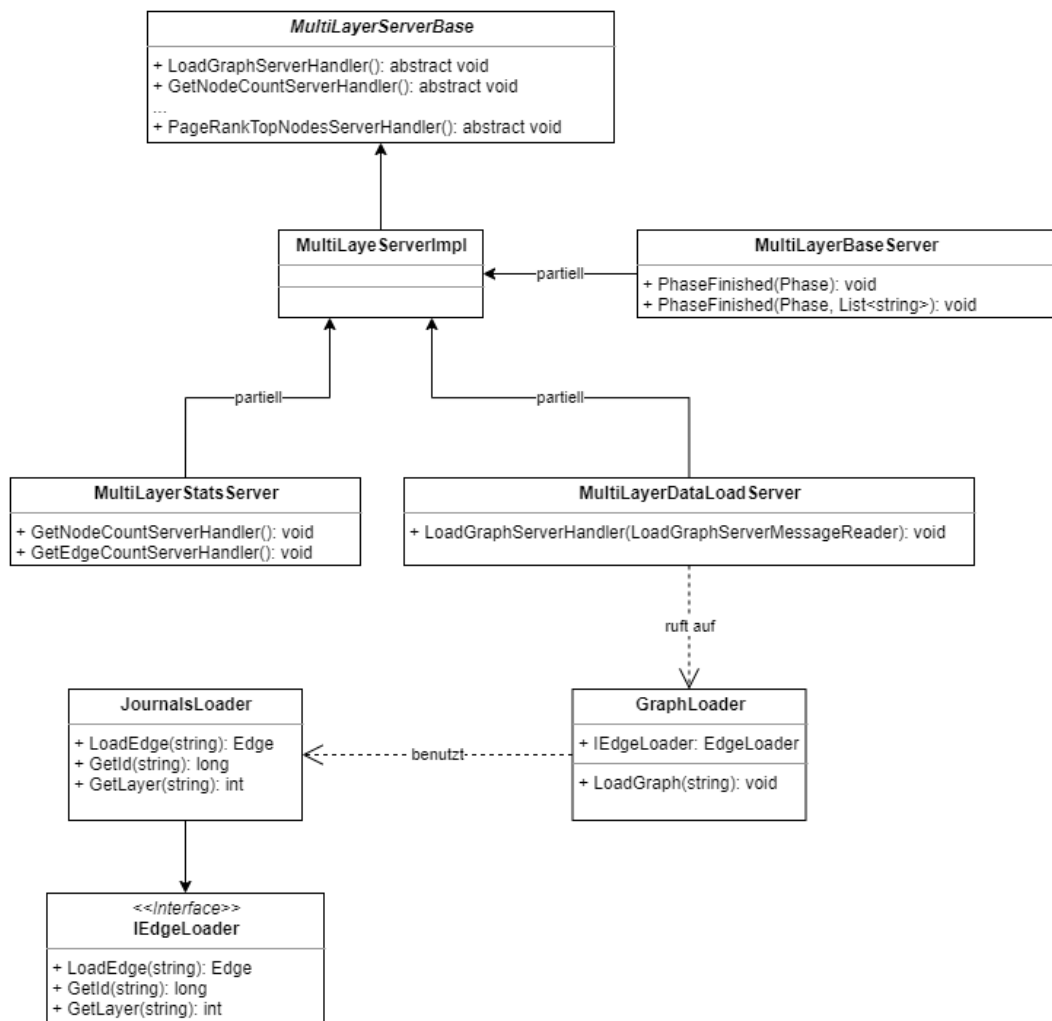


Abbildung 4.3: Klassendiagramm für die Server-Anwendung

4.2.6 Algorithmen

Für das Multilayer Graph System wurden bereits Algorithmen bzw. Metriken implementiert. In diesem Abschnitt wird grob die Funktionsweise der einzelnen Algorithmen und deren Optimierungen erläutert.

Knoten- und Kantenanzahl

Zum bestimmen der Knoten- und Kantenanzahl zählt jeder Server die Knoten bzw. deren Kanten für die er verantwortlich ist. Dabei kann bei der Kantenanzahl festgelegt werden, ob Kanten die von einem Layer in einen anderen gehen mitgezählt werden. Die Knoten und Kanten werden pro Layer gezählt und das Ergebnis an die Proxy gesendet. Diese summiert die Ergebnisse aller Server zusammen und bildet das Endergebnis.

Graph Dichte

Die Graph Dichte wird berechnet, indem die Proxy einmal die Knoten und Kanten, wie zuvor beschrieben zählt. Danach wird für jeden Layer die Dichte wie in 2.1.1 beschrieben berechnet.

Knotengrad

Der Knotengrad für jeden Knoten wird in zwei Schritte ermittelt. Da jeder Knoten seine ausgehenden Kanten speichert kann der Ausgangsgrad direkt bestimmt werden. Jeder Server zählt für seine Knoten die Anzahl der ausgehenden Kanten.

Um den Eingangsgrad zu bestimmen geht jeder Server alle seine Knoten durch. Für jede Kante wird geprüft, ob der Zielknoten auf dem gleichen Server liegt. Ist dies der Fall wird der Eingangsgrad des Knoten direkt um 1 erhöht. Liegt der Knoten auf einem anderen Server wird diesem eine entsprechende Nachricht zugesandt und dieser erhöht den Eingangsgrad des Knoten.

PageRank

Das Berechnen der PageRank Werte ist in mehrere Phasen aufgeteilt, deren Ablauf von der Proxy gesteuert wird.

In einer ersten Phase werden die Werte für alle Knoten auf den gleichen initial Wert gesetzt. Dieser kann vom Client gewählt werden.

Danach beginnt die Update Phase, in der die Werte neu berechnet werden. Hier wird, wie in 2.2 beschrieben, jeweils der neue Wert für jeden Knoten berechnet. Dabei geht jeder Server alle seine Knoten und deren Kanten durch. Falls der Zielknoten der Kante auf dem gleichen Server liegt kann der Wert sofort entsprechend geändert werden. Liegt der Zielknoten auf einem anderen Server wird sich gemerkt, um wieviel der Wert sich ändern muss und dies dem entfernten Server per Nachricht mitgeteilt.

Sind alle Updates abgeschlossen wird die Normalisierung der Werte durchgeführt und bestimmt, wie groß die Gesamtänderung aller PageRank Werte ist. Liegt die Gesamtänderung

unter einem vom Client gewählten ϵ wird der Algorithmus beendet. Ansonsten wird eine weitere Update Runde durchgeführt.

HITS

Zum bestimmen der Hub und Authority Werte läuft der Algorithmus in mehreren Phasen ab, die von der Proxy gesteuert werden.

In der ersten Phase werden die Hub und Authority Werte auf einen vom Client festgelegten initial Wert gesetzt.

Danach werden erst die Authority Werte neu berechnet. Dabei wird über jeden Knoten und jede Kante iteriert. Zeigt die Kante auf einen lokalen Knoten kann dessen Authority Werte direkt angepasst werden. Handelt es sich um einen entfernten Knoten auf einem anderen Server wird diesem eine Nachricht mit der Wertänderung geschickt.

Sind die Authority Werte neu berechnet werden diese in einer weiteren Phase normalisiert. Dabei wird die Gesamtänderung der Authority Werte bestimmt und der Proxy mitgeteilt.

Danach werden die neuen Hub Werte berechnet. Dazu iteriert jeder Server über alle seine Knoten und deren Kanten. Zeigt die Kante auf einen lokalen Knoten wird dessen Authority Wert genutzt, um den Hub Wert des aktuellen Knotens zu aktualisieren. Handelt es sich um einen entfernten Knoten wird sein Authority Wert von dem anderen Server angefragt.

Anschließend werden auch die Hub Werte normalisiert und deren Gesamtänderung berechnet. Dieser wird der Proxy zugesandt.

Die Proxy prüft nun, ob die Gesamtänderung der Hub und Authority Werte unter einem vom Client gewählten Schwellwert ϵ liegt. Ist dies nicht der Fall wird wieder mit dem Update der Authority Werte begonnen.

Optimierungen

Algorithmen, wie PageRank oder HITS müssen auf Werte von Knoten, die auf anderen Servern gespeichert sind, zugreifen. Entweder müssen die entfernten Werte aktualisiert oder gelesen werden. Diese Zugriffe direkt zu machen ist, wie in 4.1.2 erläutert, nicht sehr schnell und erfordert jedes mal Kommunikation zwischen den Servern. Da diese Kommunikation jedes mal mit einem gewissen Overhead an Aufwand verbunden ist und für große Graphen mehrere Millionen mal auftritt, muss versucht werden dies zu vermeiden.

Als Lösung für das Problem unterstützt das Multilayer Graph System das Versenden von einer Menge an Schlüssel und Werten Paaren in einer Anfrage. Damit wird der Overhead für die vielen einzelnen Anfragen vermieden. Die Implementierung von PageRank, HITS und der Berechnung des Knotengrads verwenden diese Lösung, um entfernte Werte anzufragen oder zu aktualisieren.

4.2.7 Erweiterbarkeit

Das System ist um weitere Funktionen erweiterbar. In diesem Abschnitt wird erläutert, welche Schritte notwendig sind, um neue Funktionalität hinzuzufügen. Dabei muss die gewünschte Funktionalität im Model, Client, Proxy und Server hinzugefügt werden.

Um den Ablauf zu veranschaulichen, wird dies für eine Beispielfunktion erklärt. Diese zählt die Anzahl an Knoten, die eine vom Client bestimmte Anzahl an ausgehenden Kanten besitzt.

Model

Im Model muss eine .tsl Datei für den neuen Algorithmus erstellt werden. In dieser werden die Daten und Protokolle die Graph Engine unterstützen muss beschrieben. Im Fall der Beispielfunktion sind zwei Protokolle notwendig. Eines damit der Client die Anfrage für die Ausführung an die Proxy senden kann und ein weiteres, damit die Proxy die einzelnen Server nach ihrer Anzahl an Knoten mit der gewünschten Menge an Kanten fragen kann. Dabei müssen bei beiden Protokollen die Anzahl der Kanten mitgegeben werden können. In der Anfrage an die Proxy sollen zudem die Optionen für die Ausführung von Algorithmen, sowie die Option für die Ausgabe von Ergebnissen dabei sein.

```
1 // Proxy Protocol
2 struct GetNEdgeNodesProxyMessage {
3     AlgorithmOptions AlgorithmOptions;
4     OutputOptions OutputOptions;
5     int NumberOfEdges;
6 }
7
8 protocol GetNEdgeNodesProxy {
9     Type: Syn;
10    Request: GetNEdgeNodesProxyMessage;
11    Response: void;
12 }
13
14 // Server Protocol
15 struct GetNEdgeNodesServerMessage {
16     int NumberOfEdges;
17 }
18
19 protocol GetNEdgeNodesServer {
20     Type: Syn;
21     Request: GetNEdgeNodesServerMessage;
22     Response: void;
23 }
```

Diese Protokolle müssen nun in `MultiLayerProxy` und `MultiLayerServer` eingetragen werden, damit die von TSL erzeugte Server/Proxy Basisklasse abstrakte Methoden für diese besitzen.

Damit die Server der Proxy ihre lokalen Ergebnisse zusenden können, muss die 'Phases.tsl' Datei um eine Phase für diesen Algorithmus erweitert werden.

```
1 enum Phases {
2     // DataLoad Phases
3     DataLoad = 0,
4     ...
5     NEdgesCount = 18
6 }
```

Client

Für den neuen Algorithmus muss im Client ein Kommando hinzugefügt werden, um diesen auszuführen. Das Kommando muss dabei die Anzahl der Kanten als Argument nehmen und das zuvor definierte Protokoll nutzen, um die Anfrage an die Proxy zu senden.

Es wird eine neue Klasse `NEdgesNodeCount` erstellt, welche von der abstrakten Klasse `Command` erbt. Im Konstruktor müssen nun die bereits erwähnten Daten eingegeben werden. Dabei ist wichtig, dass es ein Argument des Typs `int` gibt, welches die Anzahl der gewünschten Kanten darstellt. Nun müssen die Methoden `ApplyArguments` und `Run` implementiert werden. In `ApplyArguments` wird das übergebene Argument in einer lokalen Variable gespeichert, sodass es beim Aufruf von `Run` verwendet werden kann. In `Run` wird die im Model bereits definierte Nachricht erstellt und an die Proxy gesendet.

```
1 using MultiLayerLib;
2 using MultiLayerLib.MultiLayerProxy;
3
4 namespace MultiLayerClient.Commands {
5
6     class NEdgesNodeCount: Command {
7
8         private int NumberOfEdges { get; set; }
9
10        public NEdgesNodeCount (Client client): base (client) {
11            Name = "NEdge Node Count";
12            Keyword = "nEdgeNodeCount";
13            Description = "Counts the number of nodes that have a certain amount of ↵
14                           edges.";
15            Arguments = new string[] { "int" };
16            ArgumentsDescription = new string[] { "NumberOfEdges" };
17        }
18
19        public override void ApplyArguments(string[] arguments) {
20            NumberOfEdges = int.Parse(arguments[0]);
21        }
22
23        public override void Run() {
24            using (var msg = new GetNEdgeNodesProxyMessageWriter(Client.↵
25                AlgorithmOptions, Client.OutputOptions, NumberOfEdges)) {
26                MessagePassingExtension.GetNEdgeNodesProxy(Client.Proxy, msg);
27            }
28        }
29    }
30 }
```

In der Client Klasse muss im Konstruktor nun noch das erstellte Kommando registriert werden.

Proxy

Für die Proxy müssen zwei Funktionen implementiert werden. Zum einem der Algorithmus der Anfragen an alle Server sendet, um ihre Anzahl an lokalen Knoten mit N Kanten zu senden und diese Ergebnisse aggregiert. Zum anderem muss für das in TSL definierte

Proxy Protokoll `GetNEdgeNodesProxy` ein entsprechender Handler erstellt werden, der die Anfrage verarbeitet und den Algorithmus startet.

Der Algorithmus muss die abstrakte Klasse `Algorithm` implementieren, insbesondere die abstrakte Methode `Run()` und `AlgorithmResult GetResult(OutputOptions outputOptions)`. In `Run()` wird die im Model erstellte Anfrage an alle Server gesendet, welche auf diese mit ihrer lokalen Anzahl an Knoten mit der gewünschten Kantenanzahl antworten. Die Proxy wartet bis alle Server Ergebnisse gesendet haben und zählt diese zu einem Gesamtergebnis zusammen.

Die andere Methode dient dazu, die Ergebnisse in einer Tabellenform darzustellen. Dazu wird die Anzahl der Knoten pro Layer vermerkt und jeder Layer als eine Reihe in der Tabelle dargestellt. Bei der ersten Spalte handelt es sich um die ID des Layers, während die zweite die Anzahl der Knoten in diesem Layer darstellt. Es wird eine weitere Zeile mit der Gesamtanzahl der Knoten hinzugefügt.

```
1 public override void Run() {
2     foreach(var server in Global.CloudStorage) {
3         MessagePassingExtension.GetNEdgeNodesServer(server);
4     }
5
6     List<List<long>> phaseResults = Proxy.WaitForPhaseResultsAsLong(Phases.↵
7         NEdgesCount);
8     long[] nodeCount = new long[phaseResults[0].Count];
9
10    // Sum up the results from all the servers.
11    foreach(List<long> result in phaseResults) {
12        for (int i = 0; i < result.Count; i++) {
13            nodeCount[i] += result[i];
14        }
15    }
16 }
17
18 public override List<List<string>> GetResultTable(OutputOptions options) {
19     List<List<string>> output = new List<List<string>>();
20     long totalNodeCount = 0;
21
22     for (int i = 0; i < nodeCount.Length; i++) {
23         List<string> outputRow = ResultHelper.Row("Layer" + (i + 1), nodeCount[i].↵
24             ToString());
25         output.Add(outputRow);
26
27         totalNodeCount += nodeCount[i];
28     }
29
30     output.Add(ResultHelper.Row("Toal", totalNodeCount.ToString()));
31
32     return output;
33 }
```

Der Request Handler muss Teil der partiellen Klasse `MultiLayerProxyImpl` sein. Dafür kann entweder eine weitere Datei erstellt werden oder eine bereits vorhandene genutzt werden die thematisch zu dem Handler passt. Da Knotenzählen bereits in `MultiLayerStatsProxy`

implementiert ist wird dort auch der Handler für `GetNEdgeNodesProxy` hinzugefügt. Dieser muss die Anfrage erhalten, um den Algorithmus zu starten und dann die Ergebnisse auszugeben.

```
1 using MultiLayerProxy.Algorithms;
2 using MultiLayerLib;
3
4 namespace MultiLayerProxy.Proxy {
5
6     partial class MultiLayerProxyImpl: MultiLayerProxyBase {
7
8         public override void GetNEdgeNodesProxyHandler(↵
9             GetNEdgeNodesProxyMessageReader request) {
10             NodeCount nEdgesnodeCount = new NEdgesnodeCount(this, request.NumberOfEdges↵
11                 );
12             RunAlgorithm(nEdgesnodeCount, request.AlgorithmOptions);
13             OutputAlgorithmResult(nEdgesnodeCount, request.OutputOptions);
14         }
15         // Rest of the class ommitted
16     }
17 }
```

Server

Auf der Serverseite müssen zwei Funktionen implementiert werden. Der Request im Model definierte Handler `GetNEdgeNodesServer`, welcher von der Proxy aufgerufen wird. Dieser muss die Anfrage verarbeiten und die Funktion zur Berechnung der Kantenanzahl starten. Das Ergebnis dieser Funktion muss wieder zurück an die Proxy gesendet werden. Dazu muss natürlich die Funktion implementiert werden, welche die Kantenanzahl zählt.

Der Request Handler passt auch hier thematisch wieder zu der bereits vorhandenen Datei `MultiLayerStatsServer`, der eine partielle Klasse der Server Implementierung ist. So wird der Handler hier hinzugefügt. Es wird das Parameter `NumberOfEdges` aus der Anfrage gelesen und der Funktion zur Berechnung übergeben. Da die Ergebnisse als String Liste an die Proxy übergeben werden müssen, werden diese zu String konvertiert.

```
1 public override void GetNEdgeNodesServerHandler(GetNEdgeNodesServerMessageReader ↵
2     request) {
3     List<long> result = Stats.GetNEdgeNodes(request.NumberOfEdges);
4     PhaseFinished(Phases.NEdgesCount, Util.ToStringList(result));
5 }
```

Die Funktion zur Berechnung selbst wird als statische Funktion der bereits vorhandenen Klasse `Stats` realisiert. Gäbe es keine passende Klasse, kann diese natürlich nach Bedarf erstellt werden. In der Funktion selbst wird über alle lokalen Knoten iteriert und die Anzahl der Knoten mit der gewünschten Kantenanzahl pro Layer gezählt. Das Ergebnis wird zurück an die Proxy gesendet.

```
1 public static List<long> GetNEdgeNodeCount(int numberOfEdges) {
2     long[] nodeCount = new long[Graph.LayerCount];
3
4     foreach(Node_Accessor node in Graph.NodeAccessor()) {
5         // Only count nodes with the correct amount of edges
6         if (node.Edges.Count == numberOfEdges) {
7             nodeCount[node.Layer - 1]++;
8         }
9     }
10
11     List<long> result = new List<long>(nodeCount);
12     return result;
13 }
```

Kapitel 5

Evaluation

5.1 Cluster

Die Evaluation wurde auf dem Rechencluster des Lehrstuhls für Betriebssysteme der Heinrich-Heine Universität Düsseldorf durchgeführt. Das Cluster besitzt mehrere Knoten, die Nutzer reservieren und benutzen können.

Die zur Evaluation verwendeten Knoten haben den Xeon E3-1220 Prozessor, besitzen 16 Gigabyte Arbeitsspeicher und eine 240 Gigabyte SSD. Alle Knoten im Cluster sind mit Gigabit-Ethernet verbunden und können miteinander kommunizieren.

Das home-Verzeichnis eines Nutzers ist in einem verteilten Dateisystem gespeichert. Damit können Dateien im home-Verzeichnis, unabhängig davon welcher Knoten genutzt wird, gelesen werden.

5.2 Testdaten

Als Testdatensatz wird ein Teil des Microsoft Academic Graph genutzt [11]. Der Graph enthält wissenschaftliche Publikationen und die Zitierungsbeziehungen zwischen diesen Publikation, sowie Autoren, Institutionen, Journals, Konferenzen und Forschungsgebieten.

In dem Testdatensatz werden nur die Zitierungen zwischen Journals betrachtet. Dabei sind nur Journals enthalten, in denen zwischen 2007 und 2011 mehr als 100 Paper veröffentlicht wurden und die mindestens auf andere Journals fünf mal verwiesen haben. Der Datensatz besteht aus einer Kantendatei, die Verweise von einem Journal auf ein anderes darstellt. Dabei speichert jede Kante welches Journal auf welches verweist, in welchem Forschungsgebiet, in welchem Jahr und die Anzahl der Verweise.

Die verschiedene Forschungsgebiete stellen dabei die Layer des Graphen dar.

Die Kantendatei besteht aus 34715307 Kanten, die zwischen 12670 Journals in 12611 Layern liegen.

5.3 Benchmarks

Um die Performance des Multilayer Systems zu testen, wurden für verschiedenen Funktionen Benchmarks durchgeführt. Um zu prüfen, wie sich die Performance mit unterschiedlicher Anzahl von Servern verändert, wurde jeder der Benchmarks mit 1 bis 8 Servern durchgeführt. Bei allen Benchmarks ist die Proxy für das Messen der Laufzeit verantwortlich.

Es wird das Laden der Kantendatei, das Berechnen der Graph Dichte und das Berechnen der Hub und Authority Werte untersucht. Die Funktionen für die Benchmarks wurden ausgewählt, um die Performance für verschiedene Arten der Berechnungen zu testen.

Bei der Berechnung der Dichte handelt es sich um eine Funktion, bei der die Server nicht untereinander kommunizieren müssen. Jeder Server berechnet sein lokales Ergebnis für die Knoten, die er speichert und sendet dieses an die Proxy. Diese aggregiert am Ende das Gesamtergebnis. Hier lässt sich prüfen, wie sich die Laufzeit verhält, wenn die Server für die Ausführung nicht miteinander kommunizieren müssen.

Bei der Berechnung der Hub und Authority Werte müssen die Server viel untereinander kommunizieren. Sie führen, soweit es geht, die Aktualisierungen der Werte lokal durch, müssen aber, falls sie entfernte Knoten aktualisieren müssen oder die Werte von entfernten Knoten benötigt werden, diese von den anderen Servern erhalten. Die Proxy koordiniert hierbei nur die Ausführung der einzelnen Aktualisierungen und Normalisierung der Wert. Hiermit wird geprüft, wie sich die Laufzeit verhält, wenn einiges an Server zu Server Kommunikation nötig ist.

5.3.1 Ladezeiten

Um die Ladezeit zu messen, wurde der Ladevorgang jeweils mit 1-8 Servern durchgeführt. Dabei wurden für jede Konfiguration 5 Durchläufe durchgeführt und der Mittelwert gebildet.

Die Ladezeit verbessert sich mit einer steigenden Anzahl an Servern von knapp 8 Minuten auf nur 1,5 Minuten. Dabei ist zu beobachten, dass die Ladezeit bei 5 Servern stagniert und sich nur minimal verändert.

Dies ist wie folgt zu begründen: Beim Laden liest jeder Server die Kantendatei komplett, während er parallel die Knoten, für die er verantwortlich ist, in GE speichert. Hat man nur einen Server, ist dieser für alle Knoten verantwortlich und das Speichern der Knoten dauert deutlich länger, als das Lesen der Kantendatei. Bei einer höheren Serveranzahl bleibt die Zeit zum Lesen der Kantendatei gleich, während die Anzahl der Knoten die jeder Server speichern muss sinkt. Ab dem Punkt, wo das Lesen der Kantendatei länger dauert, als das Speichern der Knoten, lässt sich keine Verbesserung durch mehr Server erreichen. Für eine noch größere Kantendatei wäre zu erwarten, dass es mehr Server benötigt, um den Punkt zu erreichen, wo die Verbesserung stagniert.

5.3.2 Graph Dichte

Bei der Graphdichte wurden für jede Konfiguration an Servern jeweils 10 Durchläufe gemessen und der entsprechende Mittelwert gebildet. Die Ergebnisse zeigt Abbildung 5.2.

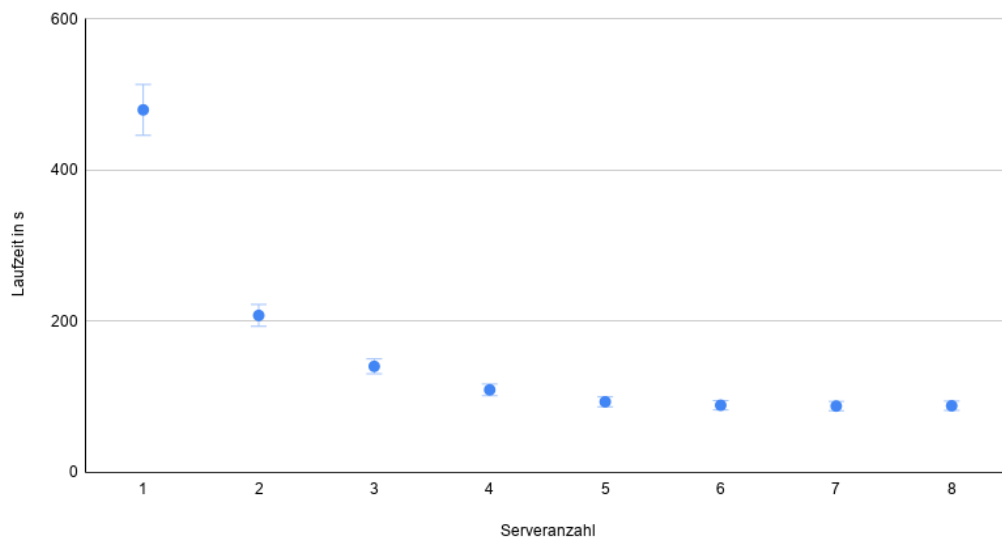


Abbildung 5.1: Dauer der Ladezeit abhängig von der Serveranzahl

Die Zeit, um die Graphdichte zu berechnen, sinkt von 11.47 Sekunden auf 2.19 Sekunden. Je größer die Anzahl der Server desto geringer ist die Anzahl an Knoten, für die ein einzelner Server verantwortlich ist. Damit sinkt auch der Aufwand für jeden einzelnen Server, umso mehr Server beteiligt sind. Diese Entwicklung lässt sich in den Werten gut beobachten. Besonders der Sprung von einem zu zwei Servern halbiert fast die Laufzeit zur Berechnung der Graphdichte.

5.3.3 HITS

Bei HITS wurde für jede Konfiguration jeweils die Dauer von 5 Update Runden für Hub und Authority Werte gemessen und der Mittelwert dieser gebildet. Die Ergebnisse sind in Abbildung 5.3 gezeigt.

Die Zeit für eine Update Runde wird mit steigender Serveranzahl immer geringer. Auch hier sinkt für jeden Server mit der Anzahl an Knoten, für die er verantwortlich ist die Arbeit, die er bewältigen muss.

Jedoch sorgen mehr Server auch dafür, dass mehr Netzwerknachrichten zwischen ihnen ausgetauscht werden müssen. Die Server müssen bei jeder Update Runde miteinander kommunizieren, um entweder die Werte von entfernten Knoten anzufragen oder diese an andere Server zu senden. Diese Kommunikation ist allerdings optimiert, indem die Anfragen für die Werte von entfernten Knoten immer nur gebündelt verschickt werden und so das Senden vieler kleiner Nachrichten vermieden wird.

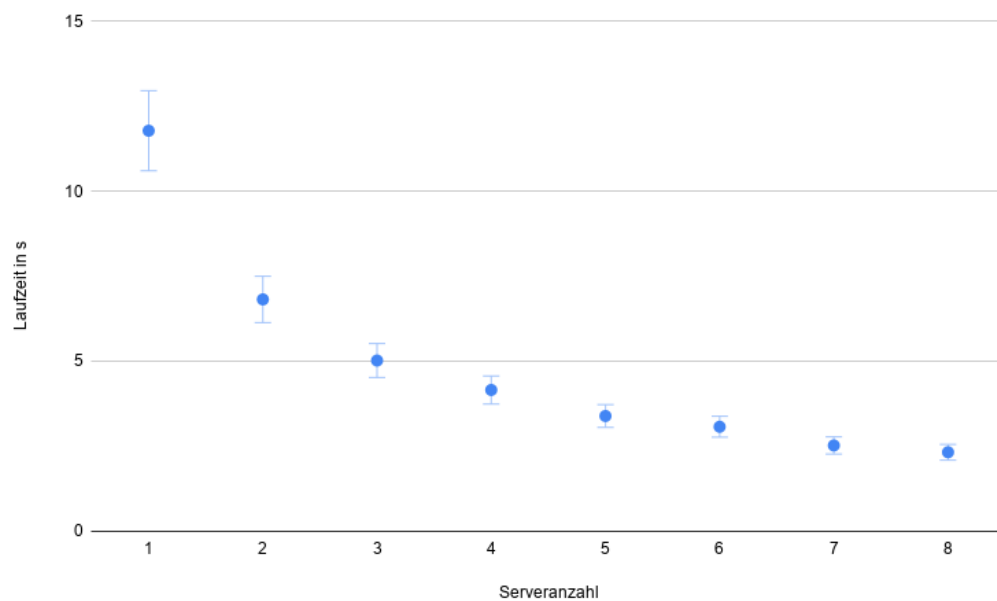


Abbildung 5.2: Dauer der Berechnung der Graph Dichte abhängig von der Serveranzahl

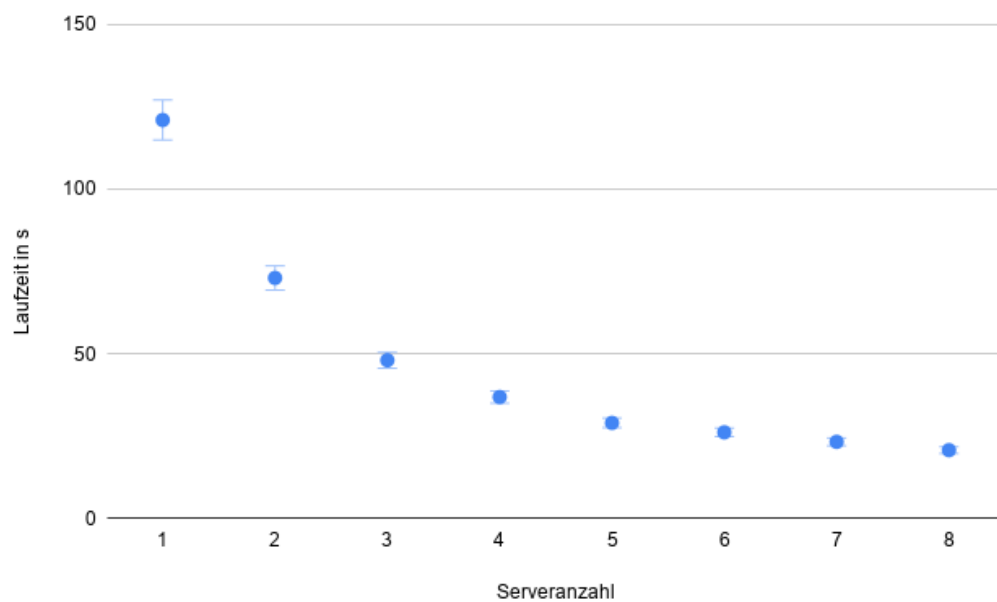


Abbildung 5.3: Dauer einer HITS Update Runde abhängig von der Serveranzahl

Kapitel 6

Fazit

In dieser Arbeit wurde mithilfe von Graph Engine ein erweiterbares System gebaut, welches genutzt werden kann, um Multilayer Graphen verteilt und skalierbar zu verarbeiten. Dazu wurden in C# drei Anwendungen erstellt, die als Client, Proxy und Server miteinander agieren. Danach wurde die Performance von verschiedenen Funktionen und Algorithmen in Hinsicht auf ihre Skalierbarkeit mit mehreren Servern gemessen.

Es konnte gezeigt werden, dass mit der Freiheit, die Graph Engine bei der Gestaltung der Graphstruktur ermöglicht, Multilayer Graphen dargestellt werden können. Dazu konnten die Funktionen von Graph Engine zum Austausch von Nachrichten zwischen einzelnen Komponenten genutzt werden, um eine effiziente Kommunikation zwischen Client, Proxy und Server zu gewährleisten. Diese ermöglicht es eine auf die einzelnen Algorithmen und Funktionen zugeschnittene Kommunikation zu haben.

Es wurde eine Client-Anwendung entwickelt, die auf der Maschine des Benutzer läuft. Diese kann über die Kommandozeilen bedient werden und bietet die Möglichkeit interaktiv Kommandos auszuführen oder in einem Batch Modus eine Reihe an Kommandos aus einer Datei heraus auszuführen.

Eine Proxy Anwendung wurde erstellt, die als Bindeglied zwischen Client und Server fungiert. Sie kann Anfragen vom Client verarbeiten und die Server anweisen die Anfrage entsprechend zu bearbeiten. Sie koordiniert die Server und kann aus deren Zwischenergebnissen ein Gesamtergebnis bilden, welches an den Client zurückgegeben wird.

Der entwickelte Server kann unter der Koordinierung der Proxy verschiedene Algorithmen ausführen. Dabei sind die Server in der Lage untereinander zu kommunizieren und nötige Daten auszutauschen.

Es wurde gezeigt, dass das System in der Lage ist einen große Multilayer Graphen zu laden und zu verarbeiten. Dabei konnte beobachtet werden, wie sich die Performance verhält, wenn man die Anzahl der Server im Cluster erhöht. Sowohl beim Laden, als auch bei den beiden geprüften Algorithmen konnte gezeigt werden, dass durch die parallele Implementierung eine größere Serveranzahl zu deutlich besserer Performance führt. Hierbei wurde insbesondere beobachtet, dass ab einer bestimmten Serverzahl der Overhead der Berechnungen den Großteil der Laufzeit ausmacht, sodass weitere Server nur zu kleineren Verbesserungen führen.

6.1 Ausblick

In diesem Abschnitt werden einige mögliche Verbesserungen für das Multilayer System vorgestellt. Sie richten sich an die Nutzbarkeit des Systems für einen Endanwender.

6.1.1 Benutzeroberfläche

Aktuell kann der Client nur über die Kommandozeile bedient werden. Dies ist umständlich und macht die Benutzung gerade für technisch weniger versierte Nutzer schwierig. Eine grafische Benutzeroberfläche, ähnlich zu der von muxViz, würde die Bedienung um einiges erleichtern.

6.1.2 Grafische Auswertung

Viele andere Graph Analyse Tools bieten die Möglichkeit sich Ergebnisse visuell darstellen zu lassen. Momentan können die Ergebnisse nur in der Kommandozeile oder als .csv Datei ausgegeben werden. In Verbindung mit einer grafischen Benutzeroberfläche könnten die .csv Dateien genutzt werden, um die Ergebnisse visuell darzustellen.

6.1.3 Cluster Verwaltung

Noch gibt es keine Möglichkeit die Proxy und Server automatisch zusammen starten zu lassen. Die Anwendungen müssen aktuell händisch auf jeder Maschine gestartet werden. Das ist gerade für Cluster mit einer hohen Serveranzahl ein erhöhter Arbeitsaufwand. Es müsste eine kleine Anwendung entworfen werden, welche die Cluster Konfiguration liest und automatisch auf den entsprechenden Maschinen die Server bzw. Proxy startet.

6.1.4 Algorithmen

In der aktuellen Form bietet das Multilayer Graph System noch nicht viele Algorithmen von Haus aus an. Die zwingt Nutzer dazu selber die benötigten Algorithmen zu implementieren. Damit das System einfacher zu benutzen ist, müssten weitere, häufig verwendete, Algorithmen und Metriken hinzugefügt werden.

Anhang A

Anhang

A.1 Einrichtung Graph Engine

Im Folgendem wird die Einrichtung von Graph Engine auf dem Rechencluster (siehe 5.1) erläutert. Die Einrichtung ist für Ubuntu 16.04/18.04/20.04 vorgesehen und wurde nicht auf anderen Versionen getestet.

Zum kompilieren von Graph Engine müssen die Pakete `libunwind8`, `g++`, `cmake` und `libssl-dev` installiert werden:

```
1 $ sudo -P apt install libunwind8 g++ cmake libssl-dev
```

Anschließend muss das .NET.Core Framework von Microsoft installiert werden. Dafür muss das entsprechende Pakete zur Paketverwaltung hinzugefügt werden. Danach kann es

```
1 $ wget -q https://packages.microsoft.com/config/ubuntu/16.04/packages-microsoft-prod.deb
2 $ sudo -P dpkg -i packages-microsoft-prod.deb
```

Das Paket `apt-transport-https` und das .NET.Core Framework können nun installiert werden.

```
1 $ sudo -P apt-get -y install apt-transport-https
2 $ sudo -P apt-get update
3 $ sudo -P apt-get -y install dotnet-sdk-2.2
```

Nun muss noch das Graph Engine Repository heruntergeladen und das Skript zum kompilieren des Projektes ausgeführt werden.

```
1 $ git clone https://github.com/Microsoft/GraphEngine.git
2 $ sed -i 's/make -j/make -j `nproc`/g' ./GraphEngine/tools/build.sh
3 $ ./GraphEngine/tools/build.sh
```

Dabei ist zu beachten, dass das kompilieren sehr viel Arbeitsspeicher benötigt und dies auf Maschinen mit geringem Arbeitsspeicher zu Probleme führen kann. Der Speicherbedarf kann verringert werden, indem die Parallelität beim Kompilieren verringert wird. Dies ist möglich indem der `make` Befehl in `build.sh` zu `make -j 'nproc' — exit -1—` geändert wird. Dies reduziert die Anzahl an parallelen Prozessen die beim kompilieren genutzt wird auf die Anzahl der vorhanden Prozessoren.

Nun können Graph Engine Anwendungen auf der Maschine ausgeführt werden. Eine Reihe an Beispielanwendungen, die in Graph Engine 2 übersetzt wurden, findet sich unter <https://github.com/ToxicJojo/graph-engine-samples>. Es ist auch ein kleines Skript vorhanden, das ein neues Graph Engine Projekt mit einem Client und Server erstellt.

Literaturverzeichnis

- [1] muxviz features. <http://muxviz.net/features.php>. Accessed: 2020-08-23.
- [2] Nuget graphengine.core version. <https://www.nuget.org/packages/GraphEngine.Core/>. Accessed: 2020-08-18.
- [3] M. De Domenico, M. A. Porter, and A. Arenas. Muxviz: a tool for multilayer analysis and visualization of networks. Journal of Complex Networks, 3(2):159–176, Oct 2014.
- [4] Manlio De Domenico, Albert Solè-Ribalta, Emanuele Cozzo, Mikko Kivelä, Yamir Moreno, Mason Porter, Sergio Gomez, and Alex Arenas. Mathematical formulation of multi-layer networks. Physical Review X, 3, 07 2013.
- [5] R. Diestel. Graphentheorie. Springer-Lehrbuch Masterclass. Springer Berlin Heidelberg, 2010.
- [6] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. Journal of the ACM, 46:668–677, 1998.
- [7] Grzegorz Malewicz, Matthew Austern, Aart Bik, James Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. page 48, 01 2009.
- [8] Larry Page, Sergey Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1998.
- [9] Bin Shao, Yatao Li, Haixun Wang, and Huanhuan Xia. Trinity graph engine and its applications. Bulletin of the Technical Committee on Data Engineering, 40(3):18–29, 9 2017.
- [10] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. pages 505–516, 06 2013.
- [11] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In WWW - World Wide Web Consortium (W3C), May 2015.

Abbildungsverzeichnis

2.1	Multilayer Netzwerkkarten [1]	6
3.1	Aufbau des Multilayer Clusters	16
3.2	Schichten des Multilayer Graph Systems.	16
4.1	Klassendiagramm für die Client-Anwendung	25
4.2	Klassendiagramm für die Proxy-Anwendung	26
4.3	Klassendiagramm für die Server-Anwendung	29
5.1	Dauer der Ladezeit abhängig von der Serveranzahl	39
5.2	Dauer der Berechnung der Graph Dichte abhängig von der Serveranzahl . .	40
5.3	Dauer einer HITS Update Runde abhängig von der Serveranzahl	40

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Thiel, Johannes

Düsseldorf, 25. August 2020

