

# **TODOs (Vor dem Druck entfernen!)**



# **Meine Abschlussarbeit**

Masterarbeit

von

## **Johannes Thiel**

aus

Solingen

vorgelegt an der

Abteilung Betriebssysteme

Prof. Dr. Michael Schöttner

Heinrich-Heine-Universität Düsseldorf

19. November 2015

Gutachter:

Prof. Dr. Michael Schöttner

Zweitgutachter:

Prof. Dr. Stefan Conrad

Betreuer:

Msc. Bernd Mustermann



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Grundlagen</b>	<b>3</b>
2.1	Graphen . . . . .	3
2.1.1	Multi Layer Graphen . . . . .	3
2.2	Page Rank . . . . .	4
2.3	HITS . . . . .	5
2.4	GraphEngine . . . . .	6
2.4.1	Memory Cloud . . . . .	7
2.4.2	Graph Model . . . . .	7
2.4.3	Computation Engine . . . . .	8
2.4.4	Datenzugriff . . . . .	10
2.5	Andere Multi Layer Graph Systeme . . . . .	10
2.5.1	MuxViz . . . . .	10
2.5.2	Multilayer Networks Library for Python . . . . .	11
<b>3</b>	<b>Architektur</b>	<b>13</b>
3.1	Graph Engine . . . . .	13
3.1.1	Einrichtung . . . . .	13
3.1.2	Datenzugriff . . . . .	13
3.1.3	Schwierigkeiten . . . . .	14
3.2	Architektur . . . . .	14
3.2.1	Model . . . . .	15
3.2.2	Lib . . . . .	15
3.2.3	Client . . . . .	15
3.2.4	Proxy . . . . .	15
3.2.5	Server . . . . .	16
<b>4</b>	<b>Implementierung</b>	<b>17</b>
4.1	Implementierung . . . . .	17
4.1.1	Model . . . . .	17
4.1.2	Lib . . . . .	19
4.1.3	Client . . . . .	19
4.1.4	Proxy . . . . .	21

4.1.5	Server . . . . .	22
4.1.6	Erweiterbarkeit . . . . .	23
<b>5</b>	<b>Evaluation</b>	<b>29</b>
5.1	Cluster . . . . .	29
5.2	Testdaten . . . . .	29
5.3	Ladezeiten . . . . .	29
<b>A</b>	<b>Mein Anhang</b>	<b>31</b>
	<b>Literaturverzeichnis</b>	<b>33</b>
	<b>Abbildungsverzeichnis</b>	<b>33</b>
	<b>Tabellenverzeichnis</b>	<b>35</b>
	<b>Algorithmenverzeichnis</b>	<b>37</b>

---

# Kapitel 1

## Einleitung

Die Untersuchung von komplexen Netzwerken ist für viele verschiedene Forschungsbereiche, wie der Biologie, Soziologie, Transportation, Phusik und vielen mehr, relevant. 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 das gewisse Probleme nur schwer angegangen werden können.

Multilayer Netzwerken besitzen verschiedene Arten von Verbindungen zwischen Knoten und können 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 den Übergang zwischen diesen Modi darstellen.

In vielen Bereichen steigen die entstehenden Datenmengen die für Netzwerk Analysen benutzt werden immer weiter an. Ein großes Beispiel hierfür sind soziale Netzwerke, welche sich durch Multilayer Netzwerke darstellen lassen.

Zur Analysen von Multilayer Netzwerken wurden bereits verschiedene Anwendungen, wie MuxViz 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, auf die der Graph aufgeteilt wird an.

Es gibt Systeme zur verteilten Verarbeitung von großen Graphen, wie zum Beispiel Pregel oder Giraph. Diese sind jedoch nicht auf Multilayer Netzwerke ausgerichtet. Ein verteiltes Graph System, in welchem die Darstellung des Graphen frei gewählt werden kann ist das

---

von Microsoft Research Asia entwickelte Graph Engine.

In dieser Arbeit wird untersucht inwiefern mit Graph Engine ein System zur verteilten Verarbeitung von Multilayer Graphen geschaffen werden kann. Dabei wird geschaut, wie die Freiheiten in der Graph Darstellung und Speicherung von GE genutzt werden können um Multilayer Graphen in GE 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.

Es wird ein System mithilfe von Graph Engine erstellt, welches Multilayer Graphen laden, verändern und Algorithmen auf ihnen ausführen kann. Dabei steht 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 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 Multi Layer Graphen

Im folgendem werden Multi Layer 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 verschiedenen 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. Multi Layer Graphen können genutzt werden um Netzwerke in unterschiedlichen Themengebieten abzubilden z.B. Soziale Netzwerke, Transport oder Biologie.

Die Abbildung 2.1 stellt verschiedene Arten von Multi Layer Graphen dar.

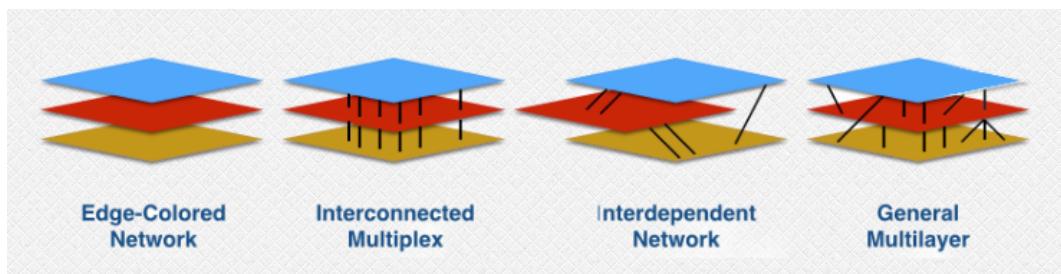


Abbildung 2.1: Multi Layer Netzwerkarten

---

### **Edgde Colored Network**

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

### **Interconnected Multiplex**

In Multiplex Netzwerken gibt es zum einem 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. Knoten können Knoten innerhalb eines Layers aber auch zwischen zwei Layern verbinden.

### **Generelle Multi Layer**

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

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

- $V$  die Menge an Knoten und  $E$  die Menge an Kanten ist
- $\Sigma_V$  und  $\Sigma_E$  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 aussgeht und zu welchem sie führt
- $l_V : V \rightarrow \Sigma_V$  und  $l_E : E \rightarrow \Sigma_E$  zwei Zuordnungen sind, die den Knoten und Kanten ihre Labels zuweisen

Da mit den Generellen Multilayer Graphen auch alle anderen Arten von MultiLayer Graphen dargestellt werden kann, werden im folgendem nur noch Genrelle Multigraphen betrachtet.

## **2.2 Page Rank**

PageRank bewertet Knoten in einem Graphen anhand von ihren 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 Variant 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, der verhindert das einzelne Knoten, die keine ausgehenden Knoten besitzen die einzigen sind die keinen Wert von 0 am Ende haben.

Als Abbruchbedingung für die Iterationen können eine feste Anzahl an Iterationsschritten gewählt werden. Alternativ kann auch ein Grenzwert  $\epsilon$  festgelegt werden und der Algorithmus bricht ab sobald die Veränderung aller Werte unter  $\epsilon$  liegt.

## 2.3 HITS

Hubs and Authorities (HITS) wurde von John M. Kleinberg in Authoritative Sources in dem Paper a Hyperlinked Environment vorgestellt.

Dabei werden in einem Netzwerk aus aufeinander verweisenden Dokumenten beurteilt. Beispiele für solche Netzwerke sind Internetseiten die aufeinander verlinken oder Wissenschaftliche Veröffentlichung die einander referenzieren. Ein Hub sind Dokumente die auf viele gute Quellen verweisen, während Authorities Dokumente sind die gute Quellen sind und auf welche 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. Umso höher der Wert umso ein besserer Hub oder Authority ist ein Knoten.

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

HITS ist ein iterativer Algorithmus der für jeden Knoten eines Graphen jeweils einen Hub und einen Authority Wert bestimmt. Hierfür werden jeweils 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 das die Summe der Quadrate 1 ergibt:  $\sum x_i^2 = 1$  und  $\sum y_i^2 = 1$ . 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  $\epsilon$  und prüfe nach jeder Iteration ob die gesammte Änderung der Authority und Hub Werte kleiner als das  $\epsilon$  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 GraphEngine**

Graph Engine (GE) ist ein verteiltes in-Memory Datenverarbeitungssystem, welches von Microsoft Research Asia entwickelt wurde. Der Quellcode von GE 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 GE sehr flexibel darin, wie die Daten gespeichert werden. Der Anwender muss selbst Schemata für die zu Speichernden Daten erstellen. Dazu bietet GE die Möglichkeit die Kommunikation unter den verschiedenen Servern mit selbst erstellten Protokollen zu koordinieren.

Im Folgendem werden die verschiedenen Komponenten von GE besprochen.

---

### 2.4.1 Memory Cloud

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

GE 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 das 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, welche direkt im Arbeitsspeicher der Maschinen verwaltet werden.

Um einen Wert anhand des Schlüssels zu finden werden zwei Schritte durchgeführt. Erst 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 fest hält 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

GE bietet ein flexibles Model mit denen 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 GE 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. speichern sowie Listen von Werten. 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 einen 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 }
```

GE hat einen eigenen Compiler für TSI, der die TSI Dateien in C# Quellcode umwandelt. So werden aus den Definitionen für Zellen, Schnittstellen generiert um diese in GE zu Erstellen, Verändern oder Löschen.

### 2.4.3 Computation Engine

Um Berechnungen durchzuführen besteht ein GE Cluster aus drei verschiedenen Komponenten die unterschiedliche Aufgaben übernehmen.

1. Server
2. Proxy
3. Client

#### Server

Die Server in einem GE Cluster haben zwei Aufgaben. Zum einem 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 GE 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 einzelne 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 GE 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. GE unterstützt hierbei drei verschiedene Arten von Protokollen.

Synchrone Protokolle sind ähnlich zu syncronen Funktionaufrufen, die auf einer anderen Maschinen stattfinden. Sie blockieren die weitere Ausführung bis eine Antwort erhalten wurde. Ein Synchroenes 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.

Asynchronre Protokolle blockieren die Ausführung des Absenders nicht. Der Empfänger antwortet beim erhalten der Anfrage sofort mit der Bestätigung das 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 Syncronen Protokolle über HTTP zu nutzen. GE 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 Synrchosnen und Asynchronen Protokolle.

## TSL

Die Kommunikationschemas von Servern und Proxies werden in TSL definiert. Der folgende Block bietet ein Beispiel für einen Server der ein Synchroenes Ping Protokoll unterstützt.

```
1 struct PingMessage {
2     string Content;
3 }
4
5 protocol SynEchoPing {
6     Type: Syn;
7     Request: PingMessage;
8     Response: PingMessage;
9 }
10
11 server PingServer {
12     protocol SynEchoPing;
13 }
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 muss und entsprechende Zeiger Arithmetik betreiben muss. GE 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 arbeiten sowohl komfortabel als auch effizient arbeiten.

```
1  using (var node = Global.LocalStorage.UseNode(cellId, accessOptions)) {  
2      int value = node.Value;  
3      node.Value = 5;  
4  }
```

### 2.5 Andere Multi Layer Graph Systeme

Im folgendem werden verschiedenen 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. MuxViz ist ein open-source Projekt, welches es ermöglicht Multi Layer 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, dass 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 das 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.

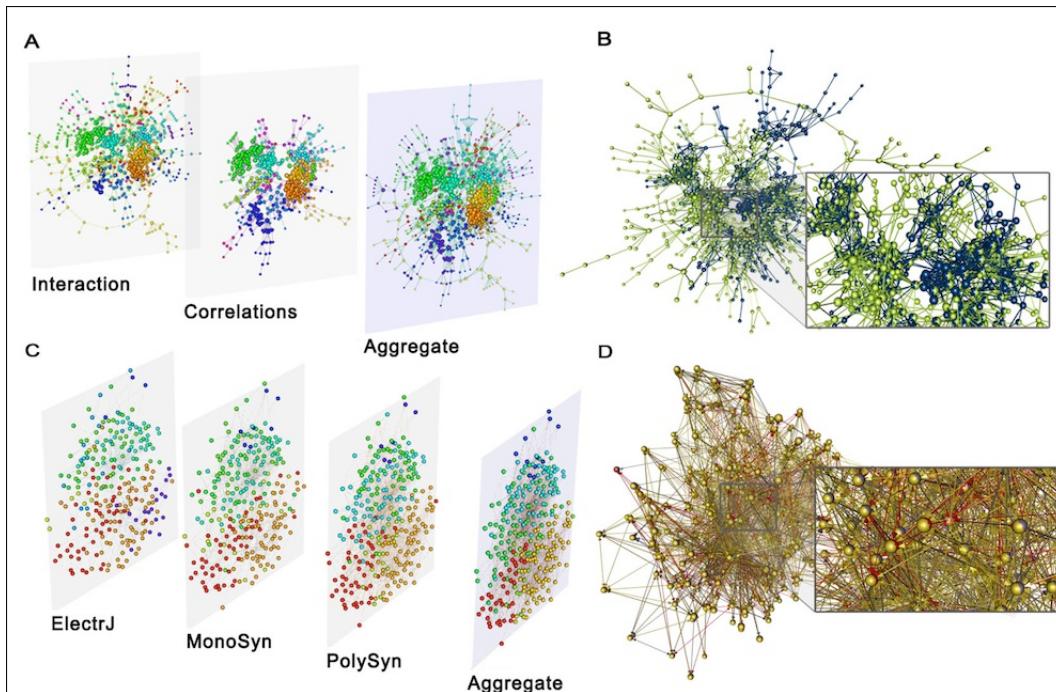


Abbildung 2.2: MuxVix

### 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 biete eine Reihe an Algorithmen zur Analyse der Netzwerke. Zudem können Netzwerke mithilfe von Matplotlib und D3 visualisiert werden.



---

# **Kapitel 3**

## **Architektur**

### **3.1 Graph Engine**

Da Graph Engine eine wichtige Rolle in dem MultiLayer Graph System spielt werden im folgendem 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 durch fehlende Dokumentation erschwert wird..

#### **3.1.1 Einrichtung**

Graph Engine wurde in C# entwickelt und verwendet den dafür verbreitet Paket-Manager NuGet, mit dem Pakete verwaltet werden können. Die aktuellste Version von dem GraphEngine.Core Paket auf NuGet ist 1.0.8467 vom 19.08.2016. Da in dieser Arbeit die Version 2.0.9912 verwendet wird, ist es nötig das GraphEngine.Core Paket selber vom Quellcode aus zu bauen.

Der Quellcode ist auf Github in dem Repository <https://github.com/microsoft/GraphEngine> zu finden.

#### **3.1.2 Datenzugriff**

Graph Engine bietet mehrere Arten auf die Daten von Zellen zuzugreifen. Dabei ist es wichtig zu verstehen, wie die verschiedenen Arten funktionieren und welche 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, da für jeden Zellenzugriff eine Netzwerknachricht an die entfernte Maschine geschickt werden muss.

Dazu gibt es zwei weitere Möglichkeiten auf Lokale Zelle mittels `Global.LocalStorage` zuzugreifen. Die erste ist `LoadCell()`. Diese Methode lädt die Daten einer Zelle aus dem von GE 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 das diese kopiert werden muss. Dazu können so auch die Werte der Zelle direkt verändert werden. Diese Methode ist damit von der Performance her beste Art auf Zellen zuzugreifen und sollte, wenn möglich immer verwendet werden.

### 3.1.3 Schwierigkeiten

Da sich die Version 2 von GraphEngine noch in Entwicklung befindet gibt es einige Schwierigkeiten die bei der Entwicklung von Anwendungen mit GraphEngine 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 beinhaltete von `cell.CellID` zu `cell.CellId` umbenannt. Auch 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)` welche 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 Beispieldokumentationen, die es im GraphEngine Repository gibt noch auf Version 1 und einige dadurch nicht kompatibel mit der neuen Version.

## 3.2 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-Modell und eine geteilte Bibliothek bieten die Basis für die anderen Komponenten.

In der Architektur ist der Client für das Senden von Anfragen, wie 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. Sobald die Anfrage fertig bearbeitet ist schickt die Proxy eine Antwort mit den entsprechenden Ergebnissen an den Client.

---

### **3.2.1 Model**

Das TSL Model dient als Basis für den Rest der Anwendung. Es wird 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. Die können z.B. der aktuelle PageRank Wert des Knoten sein.

### **3.2.2 Lib**

Die geteilte Bibliothek besteht aus zwei Komponenten. Sie enthält den von TSI generierten Code und macht es so dem Client/Proxy/Server möglich darauf zuzugreifen. Der generierte Code enthält auch die abstrakten Klassen für Proxy und Server. Diese werden in den entsprechenden Komponenten implementiert.

Außerdem enthält sie eine Sammlung aus Funktionen die alle Projekte nutzen können. Insbesondere eine 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 der Proxy möglich ist.

### **3.2.3 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.

### **3.2.4 Proxy**

Die Proxy dient als Bindeglied zwischen dem Client und den Servern. Sie nimmt Anfragen vom Client an und sorgt dafür das diese ausgeführt werden. Dabei koordiniert sie die Ausführung der verschiedenen Algorithmen und sendet sie 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 Ergebniss zurücksenden, welches die Proxy weiter verwenden kann. Ein häufiger Fall ist hierbei das 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.

### 3.2.5 Server

Die Server erfüllen zwei Aufgaben. Sie verwalten sie die Graph Daten in GE und sie warten darauf das 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 Serven können aber auch miteinander kommunizieren, wenn sie die Daten entfernter Knoten benötigen oder die Daten entfernter Knoten aktualisieren müssen. Ist ein Server mit einer angeforderten Aufgabe fertig kann er dies der Proxy melden. Dabei kann, falls nötig, auch ein Ergebniss mitgesendet werden.

---

# Kapitel 4

## Implementierung

### 4.1 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.

#### 4.1.1 Model

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

```
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 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 unterstützen 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 Ergebniss der jeweiligen Phase darstellt. Es wird hier aus zwei Gründen eine Liste an Strings verwendet. Zum einem gibt es Algorithmen die ein Ergebniss pro Layer des Graphen zurückgegeben 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 während einer 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 sowohl ein Ergebnis in Tabellenform als auch die Laufzeit des Algorithmusses.

```
1 struct AlgorithmResult {
2     string Name;
3     DateTime StartTime;
4     DateTime EndTime;
5     List<List<string>> ResultTable;
6 }
7
8 struct StandardAlgorithmMessage {
9     AlgorithmOptions AlgorithmOptions;
10    OutputOptions OutputOptions;
11 }
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 }
```

```
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     // DataLoader Phases
6     DataLoader = 0,
```

---

```

7   // Stats Phases
8   NodeCount = 1,
9   EdgeCount = 2,
10  ...
11  // EgoNetwork
12  EgoNetwork = 17
13 }
```

Die einzelnen Algorithmen definieren ihre benötigen 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.

#### 4.1.2 Lib

Stellt eine statische Klasse 'Graph' zur Verfügung die von den anderen Komponenten genutzt wird. Die Klasse ermöglicht es mit dem in GE gespeicherten Graph zu interagieren ohne die Implementierungsdetails zu kennen. So kann auf Knoten zugreiffen werden wenn nur die ID und Layer bekannt sind ohne wissen zu müssen wie man diese in eine GE Zelle-nID übersetzt.

#### 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 Ergebniss in der Konsole aus
- CSV, gibt das Ergebniss in einer .csv Datei aus

#### 4.1.3 Client

Der Client kann über die Kommandozeile gestartet und bedient werden. Entweder mit dem Argument 'interactive' um eine interaktive Sitzung zu starten in der der Nutzer Befehle eingeben kann die die Proxy dann ausführt. Oder im 'batch' Modus, wo zudem eine Datei mitgegeben werden muss die Anweisungen erhält.

#### 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 braucht. 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 wenn es danach ausgeführt wird die Werte benutzt.
  - Run(), führt das jeweilige Kommando aus.

Da die Methoden die nur Informationen zurückgegeben 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

```

1 public ShowNode (): base (null) {
2     Name = "Show Node";
3     Keyword = "showNode";
4     Description = "Shows information about a single node";
5     Arguments = new string[] {"long", "int"};
6     ArgumentsDescription = new string[] {"Id", "Layer"};
7 }
```

Die Verifizierung der Argumente findet in zwei Schritten statt. Erst 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 Type entspricht.

## Interaktiver Modus

Der Batch Modus wird mit dem Argument 'interactive' gestartet. Sobald der Client die Verbindung zum GE 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 welche pro Zeile ein Kommando enthält. Der Client arbeitet die Datei Zeile für Zeile ab und führt die Kommandos aus. Die Verarbeitung ist genau die gleich 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
```

### 4.1.4 Proxy

Die in den TSL Dateien definierte Proxy 'MultiLayerProxy' wird vom TSL Compiler in eine abstrakte Klasse 'MultiLayerProxyBase' mit abstrakten Request Handlern compiliert. Diese wird hier von 'MultiLayerProxyImpl' implementiert. Da die 'MultiLayerProxyImpl' alle Request für alle Protokolle handhaben muss wird sie über mehrere Dateien als partielle Klasse implementiert. Dies ermöglicht die sonst sehr groß werdende Klasse in kleine Teile aufzuteilen sodass eine Datei jeweils nur für einen Algorithmus verantwortlich ist.

#### BaseProxy

In 'MultiLayerBaseProxy.cs' wird grundlegende Funktionalität der Proxy definiert die nicht zu einem einzelnen Algorithmus gehört. Dies ist zum einem die Möglichkeit Algorithmen zu starten und deren Ergebnisse auszugeben zum anderem wird hier die Möglichkeit geboten auf Antworten der einzelnen Servern, 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:

- 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

Um nun auf eine Phase zu warten wird solange gewartet bis die Anzahl der Server die eine Phase als beendet gemeldet haben der Anzahl aller Server entspricht.

```
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 stellt 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

### 4.1.5 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 'MultiLayerBaserServer' aufgeteilt. Der 'MultiLayerBaserServer' bietet den Algorithmen Methoden um der Proxy mitzuteilen das sie eine Phase beendet haben.

## 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 GE 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), list 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 pufferns der Zeile festgestellt werden kann sobald der Ursprungsknoten der Kanten wechselt. Für alle Formate die unterstützt werden sollen muss nun eine Klasse erstellt werden die das Interface implementiert.

#### 4.1.6 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 Datein für den neuen Algorithmus erstellt werden. In dieser werden die Daten und Protokolle die GE 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 bereit in [verweis auf implementierung der kommands] erwähnten Daten eingegeben werden. Dabei ist wichtig das 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 übergeben 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    }
```

```

20     }
21
22     public override void Run() {
23         using (var msg = new GetNEdgeNodesProxyMessageWriter(Client.←
24             AlgorithmOptions, Client.OutputOptions, NumberOfEdges)) {
25             MessagePassingExtension.GetNEdgeNodesProxy(Client.Proxy, msg);
26         }
27     }
28 }
```

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 ihre Anzahl an lokalen Knoten mit N Kanten zu senden und diese Ergebnisse aggregiert. Zum anderem muss für das in TSL defineirte 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 Gesamtergebniss zusammen.

Die andere Methode dient dazu die Ergebnisse in einer Tabellenform darzustellen. Dazu wird sich die Anzahl der Knoten pro Layer gemerkt und jeder Layer als eine Reihe in der Tabelle dargestellt. Die erste Spalte ist die ID des Layers während die zweite die Anzahl der Knoten in diesem Layer ist. 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("Total", totalNodeCount.ToString()));
31
32     return output;
33 }
```

Der Request Handler muss Teil der partellen 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, den Algorithmus starten und dann die Ergebnisse ausgeben.

```

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
13            RunAlgorithm(nEdgesnodeCount, request.AlgorithmOptions);
14            OutputAlgorithmResult(nEdgesnodeCount, request.OutputOptions);
15        }
16
17    } // Rest of the class omitted
18 }
```

## 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 Ergebniss 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 vorhanden Datei

---

'MultiLayerStatsServer' der eine parteille Klasse der Server Implementierung ist. So wird der Handler hier hinzugefügt. Es wird das Parameter 'NumberOfEdges' aus der Anfrage gelesen und der Funktion zu 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 vorhanden 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 Ergebniss 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 Lehrstuhl 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 16Gigabyte Arbeitsspeicher und eine 240 Gigabyte SSD. Alle Knoten im Cluster sind mit Gigabit-Ethernet verbunden und können miteinander kommunizieren.

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

### **5.2 Testdaten**

Als Testdatensatz wird ein Teil des Microsoft Academic Graph genutzt. Der Graph enthält wissenschaftliche Publikation 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 5 mal verwiesen haben. Der Datensatz besteht aus einer Kantendatei, in die die Verweise von einem Journal auf ein anderes darstellen. Dabei speichert jede Kante welches Journals auf welches verweist, in welchem Forschungsgebiet, in welchem Jahr und die Anzahl der Verweise.

Die Kantendatei besteht aus X Kanten die zwischen Y Journals liegen.

### **5.3 Ladezeiten**



---

## **Anhang A**

### **Mein Anhang**

Klassendiagramme und weitere Anhänge sind hier einzufügen.



---

# **Abbildungsverzeichnis**

2.1	Multi Layer Netzwerkarten . . . . .	3
2.2	MuxVix . . . . .	11



---

## **Tabellenverzeichnis**



---

## **Algorithmenverzeichnis**



# **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, 19. November 2015

