

TODOs (Vor dem Druck entfernen!)

tsl von einem algorithmus ausführlich erklären	10
Output refactor abschliesen und dann hier erklären	13
Paper lesen und hier noch einfügen	31
Inline todo	31
Figure: Übersicht Architektur	31



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

1	Grundlagen	1
1.1	Page Rank	1
1.2	HITS	2
1.3	GraphEngine	2
1.3.1	Memory Cloud	3
1.3.2	Graph Model	3
1.3.3	Computation Engine	4
1.3.4	Datenzugriff	6
2	Hauptteil	7
2.1	Architektur	7
2.1.1	Model	7
2.1.2	Lib	7
2.1.3	Client	8
2.1.4	Proxy	8
2.1.5	Server	8
2.2	Implementierung	8
2.2.1	Model	8
2.2.2	Lib	10
2.2.3	Client	11
2.2.4	Proxy	12
2.2.5	Server	14
2.2.6	Erweiterbarkeit	14
3	Evaluation	21
3.1	Cluster	21
3.2	Testdaten	21
4	Einleitung	23
4.0.1	Vor dem Druck/Finale Version	23
5	Gliederung der Arbeit	25
6	Struktur	27

6.1	Mein Abschnitt	27
6.1.1	Unterabschnitt	27
6.1.2	Noch ein Unterabschnitt	27
6.1.3	Wieder eine Subsection	28
6.2	Und eine einfache Section	28
6.3	Zeilenumbrüche und Absätze	28
6.4	Neue Seite erzwingen	28
7	Organisation	31
8	Text	33
9	Zitieren/Referenzen	35
10	Auflistungen	37
10.1	Enumeration	37
10.2	Bullet points	37
11	Bilder und Grafiken	39
12	Tabellen	41
12.1	Einfache Tabelle	41
12.2	Komplexere Tabelle: Multicolumn und Multirow	42
13	Matheumgebung	43
13.1	Indices/Zahlen hochstellen/tiefstellen	43
14	Code	45
14.1	Pseudocode	45
14.2	C-Code	45
14.3	Java-Code	46
A	Mein Anhang	47
	Literaturverzeichnis	49
	Abbildungsverzeichnis	49
	Tabellenverzeichnis	51
	Algorithmenverzeichnis	53

Kapitel 1

Grundlagen

1.1 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 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, 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 ϵ festgelegt werden und der Algorithmus bricht ab sobald die Veränderung aller Werte unter ϵ liegt.

1.2 HITS

Hubs and Authorities (HITS) wurde von John M. Kleinberg in Authoritative Sources in a Hyperlinked Environment¹ 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 besser 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 Iteration so normalisiert dass 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.

HITS aktualisiert zuerst die Authority Werte der Knoten und danach die Hub Werte.

Authority Update:

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

$$x_p = \sum_{q:(q,p) \in E} y_q$$

Danach werden alle Authority Werte wie oben beschrieben normalisiert:

$$x_p = \frac{x_p}{\sum x_q^2}$$

Hub Update:

Genauso

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

1.3 GraphEngine

Graph Engine (GE) ist ein verteiltes in-Memory Datenverarbeitungssystem welches von Microsoft Research Asia entwickelt wurde. Es bietet einen verteilten Key-Value Speicher in dem Daten gespeichert und verarbeitet werden können

¹ [?]

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

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. 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.

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.

1.3.2 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 1.1 dargestellt.

Listing 1.1: Beispiel für einen in TSL definierte Graphenstruktur

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

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

1.3.3 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 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 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 durchzuführen. 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.

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

Ansyncrhone 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 Syncronen 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 Syncrone 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.

1.3.4 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  }
```

Kapitel 2

Hauptteil

2.1 Architektur

Das MultiLayer Graph System setzt sich aus mehren 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 bieten die Basis für die anderen Komponenten.

2.1.1 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 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.

2.1.2 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.

2.1.3 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.

2.1.4 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 werden oder zurück an den Client gesendet werden.

2.1.5 Server

Die Server erfüllen zwei Aufgaben. Sie verwalten sie die Graph Daten in GE. 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 Servern 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.

2.2 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.

2.2.1 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 cell struct Node {
11     long Id;
12     int Layer;
13     PageRankData PageRankData;
14     HITSData HITSData;
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 zurückgiebt Double werte benutzt. Die Phasen werden in einer seperaten 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 Algorithmus.

```
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.

Die Server definition zieht alle diese Protokolle zusammen.

on einem algo-
us ausführlich
ren

2.2.2 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 kenne. 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-ID 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

2.2.3 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
```

2.2.4 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 einen 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 'MultiLayer-BaseProxy' 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

Ausgabe

Die Proxy stellt den Algorithmen

Output refactor a schließen und da hier erklären

2.2.5 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 '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.

2.2.6 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 müssen beschrieben. Im Fall der Beispieldfunktion 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
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 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.NEdgesCount);
7     long[] nodeCount = new long[phaseResults[0].Count];
8
9     // Sum up the results from all the servers.
10    foreach(List<long> result in phaseResults) {
11        for (int i = 0; i < result.Count; i++) {
12            nodeCount[i] += result[i];
13        }
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].ToString());
24         output.Add(outputRow);
25
26         totalNodeCount += nodeCount[i];
27     }
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
5      PhaseFinished(Phases.NEdgesCount, Util.ToStringList(result));
6  }
```

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 gewunschten 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 3

Evaluation

3.1 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, bezitzen 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 wird gelesen werden.

3.2 Testdaten

Kapitel 4

Einleitung

Nachfolgend einige Punkte, die es zu beachten gilt:

1. Dieses Template dient als Vorlage, damit diverse Anforderungen wie allgemeines Layout der Arbeit, Formatierung etc. einheitlich sind. Weiterhin bietet es einen etwas leichteren Einstieg in LaTeX und sollte somit nicht zu viel Zeit in Anspruch nehmen, damit die eigentliche Arbeit nicht darunter leidet.
2. Einzelne Kapitel dieser Vorlage geben einige grundlegende Konstrukte vor, die zum größten Teil per Copy/Paste übernommen werden können. Dennoch ist es unvermeidbar, sich mit LaTeX (bis zu einem gewissen Maß) zu beschäftigen. Mit Google kann man die meisten Probleme sehr leicht lösen und bekommt oft direkt den passenden Code geliefert.

4.0.1 Vor dem Druck/Finale Version

Checkliste, was es vor dem Druck der finalen Version noch zu beachten gibt (keine Garantie auf Vollständigkeit!):

1. Das Dokument ist auf doppelseitigen Druck ausgelegt, d.h. unbedingt doppelseitig drucken lassen.
2. Vor dem Druck **alles** nochmal kontrollieren.
3. Sind alle Kapitel vorhanden?
4. Ist das Inhaltsverzeichnis vollständig?
5. Sind alle Bestandteile der Arbeit vorhanden (Titelblatt, Inhaltsverzeichnis, Anhang, Erklärung, etc.)
6. Sind leere Abschnitte entfernt (z.B. keine Algorithmen im Algorithmusverzeichnis)?
7. Sind alle Todos aus dem Dokument entfernt und erledigt?

-
8. Ist das Todo Verzeichnis aus dem Dokument entfernt?
 9. Steht das richtige Datum auf der Arbeit?
 10. Das Thema muss **exakt** so formuliert sein, wie bei der Anmeldung der Arbeit.
 11. Die Arbeit ist gebunden in dreifacher Ausführung beim Prüfungsamt abzugeben.
 12. Jegliche digitalen Dokumente, Quellcode, Programme, Messergebnisse und Anleitungen für den Aufbau von Testumgebungen sind jeder gedruckten Arbeit auf CD/DVD hinzuzufügen.

Kapitel 5

Gliederung der Arbeit

Eine sinnvolle und korrekte Unterteilung der Arbeit ist nicht nur wichtig für den Leser, sondern hilft auch dem Verfasser bei der Anfertigung.

Allgemeiner grundlegender Aufbau mit den wichtigsten inhaltlichen Aspekten:

1. Einleitung: Hinführung an die Thematik, Problemstellung, Ziel der Arbeit, Aufbau der Arbeit
2. Grundlagen: Erläuterung von Begriffen, Definitionen, Algorithmen, Programmiertechniken, etc. die im Kontext der Arbeit Verwendung finden und welche der Leser für das Verständnis benötigt. Hierbei muss nicht ein ganzer Themenbereich der Informatik ausgearbeitet und erklärt werden. Referenzen auf Literatur sind hier sehr wichtig.
3. Hauptteil: Der Hauptteil der Arbeit besitzt keine vorgegebene Gliederung. Hier muss eine dem Thema angemessene Unterteilung in ein oder mehrere Kapitel und Unterkapitel ausgearbeitet werden. Oft bietet sich hier eine Trennung von Design/Architektur und Implementierung an.
4. Evaluation: Messungen von selbstgeschriebenen oder bereits vorhanden Benchmarks müssen in diesem Abschnitt mithilfe von Tabellen und Grafiken dokumentiert werden. Weiterhin muss der Versuchsaufbau festgehalten werden. Dabei sind alle in den Messungen verwendeten Systeme mit ihren (relevanten) Hardware- und Softwarekonfiguration aufzuführen. Die Ergebnisse müssen bewertet und ausführlich erklärt werden. Ausreisser oder Anomalien sind nicht außer Acht zu lassen!
5. Zusammenfassung, Fazit und Ausblick: Zum Schluss werden die vorgestellten Konzepte und die Implementierung zusammengefasst. Ein Bezug auf die erfüllten/nicht erfüllten Ziele der Arbeit sowie die Ergebnisse der Evaluation dürfen hier nicht fehlen. Weiterführende Ideen und Konzepte, die über die Themenstellung hinausgehen und Verbesserungspotential sind mögliche Punkte für einen Ausblick.

Kapitel 6

Struktur

Die Arbeit kann mit den nachfolgenden Befehlen sehr einfach in Kapitel mit Unterkapiteln eingeteilt werden:

1. `\chapter{Name}`
Beginne ein neues Kapitel
2. `\section{Name}`
Beginne einen neuen Abschnitt eines Kapitels
3. `\subsection{Name}`
Beginne einen Unterabschnitt eines Abschnitts in einem Kapitel
4. `\subsubsection{Name}`
Beginne einen Unterabschnitt in einem Unterabschnitt

Mithilfe von Labels lassen sich Referenzen erstellen:

`\label{LabelName}`

Um auf ein Label zu linken:

`\ref{LabelName}`

6.1 Mein Abschnitt

6.1.1 Unterabschnitt

In diesem Abschnitt wird beschrieben...

6.1.2 Noch ein Unterabschnitt

Noch tiefere Hierarchie

Im Unterabschnitt 6.1.1 wurde erläutert...

Noch ein Unterpunkt vom Unterpunkt

6.1.3 Wieder eine Subsection

6.2 Und eine einfache Section

6.3 Zeilenumbrüche und Absätze

Manchmal ist es ganz gut (hauptsächlich am Ende der Arbeit), die Strukturierung etwas in die Hand zu nehmen, da LaTeX nicht immer eine optimale Aufteilung des Textes mit Grafiken und Tabellen erzeugt. Ein Zeilenumbruch lässt sich mit

\\\

erzeugen.

Falls ein Umbruch auf einen weiteren folgt, so erzeugt dies einen Absatz.

6.4 Neue Seite erzwingen

Auch das erzwingen einer neuen Seite kann nützlich sein, um eine bessere Formatierung zu bekommen. Mit

\newpage

lässt sich dies erzwingen.

Text auf neuer Seite.

Kapitel 7

Organisation

Einige Kommandos, die bei der Ausarbeitung sehr hilfreich sein können:

1. `\todo{Text}`

Auffällige Markieren für TODOs

2. `\begin{verbatim}Text\end{verbatim}`

Text in diesem Block wird von Latex nicht ausgewertet (Sonderzeichen, Latex-Kommandos)

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor
invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est
Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam
voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren,
no sea takimata sanctus est Lorem ipsum dolor sit amet.

Paper lesen und
noch einfügen

Inline todo

Missing
figure

Übersicht Architektur

Dies ist ein Text mit Sonderzeichen, der noch nicht passend für Latex formatiert wurde.
Jegliche Formatierung wird hier ignoriert. Copy/Paste von Code etc. ist hier auch möglich.
`__?!//\\{}{}{<><`

Kapitel 8

Text

Einige allgemeine Punkte zum Schreiben von Text:

- *kursiv*
- **fett**
- unterstrichen
- „deutsche Anführungszeichen“
- Umlaute ä, ö, ü und ß
- “englische Anführungszeichen”
- Wörter_durch_Unterstriche_getrennt

Kapitel 9

Zitieren/Referenzen

Referenzen aus dem Literaturverzeichnis immer mit

`\cite{CiteKey}`

kenntlich machen. Der CiteKey wird bei jedem BibTex-Eintrag festgelegt, z.B.: [?]
Falls dieser Key fehlt, so wird anstatt einer Zahl in eckigen Klammern ein [?] angezeigt.
Nicht verwendete/referenzierte Literatur wird automatisch nicht im Literaturverzeichnis aufgeführt.

Kapitel 10

Auflistungen

10.1 Enumeration

1. Erster wichtiger Punkt der Liste
2. Ein weiterer
3. Noch einer

10.2 Bullet points

- Apfel
- Birne
- Gurke

Kapitel 11

Bilder und Grafiken

Bilder bzw. Grafiken lassen sich auf verschiedene Arten mit LaTeX einbinden. Es gilt zu beachten, dass im Text ein inhaltlicher Bezug mithilfe von

```
\ref{label_name_grafik}
```

hergestellt ist.

Beispiel: In Darstellung 11.1 ist das Logo der HHU zu sehen, welches ...

Viele Dateiformate werden unterstützt, jedoch ist die Nutzung von jpg, png, gif oder pdf (Vektorgrafiken) zu empfehlen.



Abbildung 11.1: Einzelne Grafik (centered)



Abbildung 11.2: Einzelne Grafik (verkleinert)



(a) HHU Logo links



(b) HHU Logo rechts

Abbildung 11.3: Zwei Grafiken horizontal angeordnet



(a) HHU Logo oben



(b) HHU Logo unten

Abbildung 11.4: Zwei Grafiken vertikal angeordnet

Kapitel 12

Tabellen

Tabellen sind vor allem im Abschnitt der Evaluation sehr wichtig. Die Messergebnisse aus der Tabelle müssen im Text erklärt und referenziert werden.

12.1 Einfache Tabelle

Anzahl der Iterationen	100	250	500	1.000
Implementierung 1 (ms)	0,375738	0,51265	0,77477	1,201552
Implementierung 2 (ms)	0,397062	0,397987	0,411314	0,415611

Tabelle 12.1: Durchschnittliche Zeit pro Frame aus 10.000 Frames in Millisekunden

Die Ergebnisse aus Tabelle 12.1 zeigen, dass...

12.2 Komplexere Tabelle: Multicolumn und Multirow

Datensatz1	Datensatz2		Typ	
	$t_{total}(ms)$	#Pakete	$t_{total}(ms)$	#Pakete
0	45,654	2	113,692	8
1	60,385	4	138,214	16
2	78,785	6	247,939	24
3	121,979	11	389,466	44
4	241,632	20	653,795	80
5	412,281	39	1.088,248	156
6	678,701	77	1.921,394	308
7	1.119,715	152	3.666,934	608
8	2.029,649	303	7.048,150	1.212
9	3.896,919	604	13.504,028	2.416
10	7.526,054	1.207	26.443,533	4.828

Tabelle 12.2: Gesamtzeiten der Übertragungen der Datensätze 1 und 2 in Millisekunden der Typen 0 bis 10

Die Messergebnisse der Testreihe sind in Tabelle 12.2 festgehalten und zeigen...

Kapitel 13

Matheumgebung

13.1 Indices/Zahlen hochstellen/tiefstellen

1. $Matrix_{4 \times 4}$
2. x^{a+b}

Kapitel 14

Code

14.1 Pseudocode

Input: A[0..N-1], value
Output: value if found, otherwise not_found

```
// Lokale Variablen
let low = 0 let high = N - 1
// Es gibt auch noch weitere Loops wie ForEach
while low <= high do
    // Invariante: value > A[i] for all i < low
    // Invariante: value < A[i] for all i > high
    mid = (low + high) / 2;
    if A[mid] > value then
        high = mid - 1;
    else
        if A[mid] < value then
            low = mid + 1;
        else
            return mid;
        end
    end
    return not_found;
end
```

Algorithmus 1: Binärsuche in Pseudocode

14.2 C-Code

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
```

```
4 | {  
5 |     printf("Hello World!\n");  
6 |     return 0;  
7 | }
```

14.3 Java-Code

```
1 | public class HelloWorld  
2 | {  
3 |     public static void main(String[] args)  
4 |     {  
5 |         System.out.println("Hello World!");  
6 |     }  
7 | }
```

Anhang A

Mein Anhang

Klassendiagramme und weitere Anhänge sind hier einzufügen.

Abbildungsverzeichnis

11.1 Einzelne Grafik (centered)	39
11.2 Einzelne Grafik (verkleinert)	40
11.3 Zwei Grafiken horizontal angeordnet	40
11.4 Zwei Grafiken vertikal angeordnet	40

Tabellenverzeichnis

12.1 Durchschnittliche Zeit pro Frame aus 10.000 Frames in Millisekunden	41
12.2 Gesamtzeiten der Übertragungen der Datensätze 1 und 2 in Millisekunden der Typen 0 bis 10	42

Algorithmenverzeichnis

1	Binärsuche in Pseudocode	45
---	--------------------------	----

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

