Speichereffiziente Berechnung des Clustering-Koeffizienten

Bachelor-Thesis von Johannes Decher
Januar 2013

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
P2P Networks Group

Speichereffiziente Berechnung des Clustering-Koeffizienten

Vorgelegte Bachelor-Thesis von Johannes Decher

1. Gutachten: Prof. Dr. Thorsten Strufe
2. Gutachten: Benjamin Schiller

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30. Januar 2013

(J. Decher)

Abstrakt

Der Clustering-Koeffizient ist eine Metrik, die bei der Analyse von Graphen benutzt wird. Durch diese Metrik wird ausgedrückt, wie viele der Nachbarn eines Knotens untereinander verbunden sind. Speziell bei Graphen aus sozialen Netzwerken wird diese Metrik sehr häufig eingesetzt, da hierbei der Clustering-Koeffizient Aussagen über die Beziehungen der Benutzer untereinander ermöglicht.
Die Berechnung des Clustering-Koeffizienten unter Verwendung bekannter Verfahren, ist allerdings sehr speicher- und rechenintensiv und kann dementsprechend nur auf Maschinen mit viel Speicher berechnet werden.
Das Ziel dieser Arbeit ist es, ein möglichst speichereffizientes Verfahren zu finden, mit welchem die Berechnung des Clustering-Koeffizienten auch auf Maschinen mit wenig Speicher möglich ist.
Da es auf den Maschinen mit wenig Arbeitsspeicher nicht möglich ist den gesamten Graphen im Speicher zu halten, benutzen wir für unseren Ansatz einen Cache. Dieser besitzt allerdings eine beschränkte Kapazität und somit können nur eine begrenzte Anzahl an Einträgen zwischengespeichert werden. Deswegen ist es wichtig, diesen möglichst effizient zu organisieren. Für diese Organisation verwenden wir verschiedene Caching-Strategien. Die Berechnung erfolgt dabei auf Graphen aus sozialen Netzwerken. Graphen aus sozialen Netzwerken besitzen die Eigenschaft, dass Nachbarn eines Knotens stärker miteinander verbunden sind als bei Graphen aus anderen Netzwerken. Demnach besitzen Knoten untereinander viele gemeinsame Nachbarn. Durch verschiedene Abarbeitungsreihenfolgen wird deshalb die Reihenfolge, wie die Knoten bearbeitet werden, so verändert, dass die nacheinander berechneten Knoten eine möglichst große und gemeinsame Nachbarschaft besitzen. Dadurch kann die begrenzte Kapazität des Caches besser ausgenutzt werden.
Insgesamt haben wir 5 Abarbeitungsreihenfolgen und 5 Caching-Strategien auf Graphen aus sozialen Netzwerken getestet. Durch die Kombination dieser verschiedenen Strategien und einer Erweiterung für 3 Abarbeitungsreihenfolgen, haben wir entsprechend 40 verschiedene Kombinationen aus Abarbeitungsreihenfolge und Caching-Strategie getestet. Dabei konnte eindeutig nachgewiesen werden, dass die Abarbeitungsreihenfolge einen deutlichen Einfluss auf die Effektivität des Caches und die Laufzeit der Berechnung hat.

Inhaltsverzeichnis

1 Einleitung

Eine der häufigen Datenrepräsentationen von Objekten, die in Beziehung zueinander stehen, sind Graphen. Sie bestehen aus einer Menge von Knoten, die untereinander durch Kanten verbunden sein können. Bekannte Beispiele von Datenstrukturen, die als Graph ausgedrückt werden können, sind zum Beispiel das World Wide Web, die Internet-Topologie oder das Telefonnetz [KNT10]. Es können allerdings auch verschiedene Strukturen aus der nicht Technikwelt als Graphen modelliert werden. Als Beispiel kann hier der Liniennetzplan eines öffentlichen Verkehrsunternehmens oder die Nahrungskette in der Natur genannt werden.
Ein weiteres Beispiel für Graphen sind soziale Netzwerke. Unter sozialen Netzwerken versteht man Akteure, die über eine Beziehung zueinander verbunden sind (z.B. Freundschafts- oder Geschäftsbeziehungen). Bei der Darstellung eines sozialen Netzwerkes als Graph, werden die Benutzer als Knoten dargestellt und die Beziehungen als Kanten. Vor allem die sozialen Netzwerke sind in den letzten Jahren sehr stark gewachsen. Dienste wie Facebook, Google+, Twitter oder YouTube haben mittlerweile Mitgliederzahlen von mehreren Milliarden.
Eine Metrik die häufig bei der Analyse von Graphen beutzt wird, ist der Clustering-Koeffizient. Durch diesen wird ermittelt, wie stark die Nachbarn eines Knotens untereinander verbunden sind.
Die Berechnung dieser Metrik unter Verwendung bekannter Verfahren, ist allerdings sehr speicher- und rechenintensiv und somit für große Graphen nur auf Maschinen mit viel Speicher möglich. Unser Ziel ist es nun ein Verfahren zu finden, mit welchem die Berechnung auch auf Maschinen mit wenig Speicher möglich ist.
Durch den begrenzten Speicher dieser Maschinen ist es nicht möglich den gesamten Graphen in diesen zu laden. Deswegen wird ein Cache benutzt, in dem eine begrenzte Anzahl an Einträgen gespeichert wird. Wegen der geringen Anzahl an Einträgen ist es allerdings wichtig, den Cache möglichst effizient zu nutzen. Um dies zu erreichen, kommen verschiedene Caching-Strategien zum Einsatz. Bekannte Strategien hierfür sind beispielsweise die LRU- oder die FIFO-Strategie.
Die Berechnung des Clustering-Koeffizienten wird dabei auf Graphen aus sozialen Netzwerken durchgeführt. Ein Phänomen, was vor allem bei den sozialen Netzwerken beobachtet werden kann, ist die starke Verbundenheit der Freunde bzw. der Nachbarn untereinander. Ist ein Knoten beispielsweise mit zwei Nachbarn verbunden, ist die Wahrscheinlichkeit in einem sozialen Netzwerk, dass zwischen diesen Nachbarn ebenfalls eine Verbindung besteht höher als bei anderen Typen von Netzwerken bzw. Graphen [NP03].
Bei der Berechnung des Clustering-Koeffizienten für einen Knoten müssen ebenfalls dessen Nachbarn betrachtet und dementsprechend in den Cache geladen werden. Hierbei wird nun das beschriebene Phänomen der sozialen Netzwerke ausgenutzt. Wird nachdem der Clustering-Koeffizient für einen Knoten berechnet wurde, einer der Nachbarknoten als nächstes berechnet, sind bereits viele dessen Nachbarn bei der Berechnung des vorherigen Knoten geladen worden und müssen deswegen nicht noch einmal geladen werden. Durch die Reihenfolge, wie die Knoten abgearbeitet werden, kann somit eine effizientere Nutzung des Caches erreicht werden. Als Abarbeitungsreihenfolgen verwenden wir beispielsweise verschiedene Graphen-Traversierungen wie BFS oder DFS.
In mehreren Tests auf sozialen Graphen haben wir diese Abarbeitungsreihenfolgen und Caching-Strategien getestet. Dabei zeigten sich einige Reihenfolgen und Caching-Strategien als gut, aber auch als weniger gut geeignet. Generell hat sich gezeigt, dass die Reihenfolge einen starken Einfluss auf die Effektivität des Caches und Laufzeit der Berechnung hat.
Der Aufbau dieser Arbeit gliedert sich wie folgt:
In Kapitel 2 werden wir Hintergrundinformationen und einige Definitionen, die für diese Arbeit wichtig sind, einführen. Es folgt in Kapitel 3 die Beschreibung einiger verwandter Arbeiten zu dem Thema: Berechnung des Clustering-Koeffizienten. Kapitel 4 beinhaltet unseren eigenen Ansatz zur Berechnung des Clustering-Koeffizienten gefolgt von Kapitel 5, wo wir die Implementierung unseres Ansatzes beschreiben. Danach diskutieren wir in Kapitel 6 die Ergebnisse, die wir bei den Tests unseres Ansatzes ermitteln konnten. Abschließend folgt in Kapitel 7 eine Zusammenfassung unserer Arbeit und einen Ausblick, wie diese Arbeit als Grundlage für weitere Tests benutzt werden kann.

2 Hintergrund

In diesem Kapitel führen wir Hintergrundinformationen und einige Definitionen ein, die für diese Arbeit notwendig sind, beginnend mit der Defintion eines Graphen und dessen Eigenschaften. Nachfolgend sind die Definition des Clustering-Koeffizienten und abschließend eine kurze Beschreibung eines Caches aufgeführt.

2.1 Graphen-Definition

Ein Graph besteht aus einem Tupel $G = (V, E)$, wobei V eine endliche Menge von Knoten und E eine Menge von Kanten ist. Bei einem gerichteten Graphen zeigt jede Kante $(i,j) \in E$ von Knoten i zu Knoten j. Eine Abbildung eines gerichteten Graphen findet sich in Abbildung 1. Bei einem ungerichteten Graphen zeigt jede Kante in beide Richtungen. Das heisst, wenn $(i,j) \in E$ eine Kante von Knoten i zu Knoten j ist, so gilt auch $(j,i) \in E$. Eine Abbildung eines ungerichteten Graphen findet sich in Abbildung 2.
Der Grad eines Knotens $d(i)$ ist definiert als die Anzahl der Kanten, die ein Knoten besitzt. Beispielsweise besitzt der Knoten 3 in Abbildung 2 einen Grad von 3. Bei einem gerichteten Graphen kann der Grad zusätzlich in einen Eingangsgrad und Ausgangsgrad unterteilt werden. Der Eingangsgrad $d_{in}(i)$ gibt an, wieviele Kanten bei einem Knoten enden. Analog dazu beschreibt der Ausgangsgrad $d_{out}(i)$, wieviele Kanten bei diesem Knoten beginnen. Knoten 2 aus Abbildung 1 hat beispielsweise einen Eingangsgrad von 3 und einen Ausgangsgrad von 1. Dies ergibt zusammen den Grad von 4.
Von einem Nachbarn j eines Knotens i spricht man, wenn eine Verbindung $(i,j) \in E$ existiert.
Ein Dreieck in einem Graphen besteht aus drei Knoten, wobei die drei Knoten untereinander vollständig miteinander verbunden sind. Beispielsweise bilden die Knoten 1, 2 und 3 in Abbildung 2 ein Dreieck. In Abbildung 1 hingegen gibt es keine Dreiecke, da hier keine 3 Knoten komplett miteinander verbunden sind.

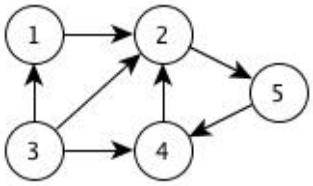

Abbildung 1: Gerichteter Graph

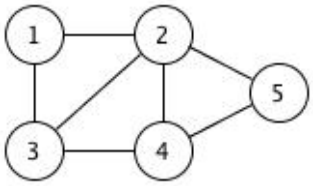

Abbildung 2: Ungerichteter Graph

2.2 Clustering-Koeffizient

Wie beschrieben, besitzen Graphen aus sozialen Netzwerken eine höhere Verbundenheit der Freunde bzw. der Nachbarn untereinander. Dieses Verhältnis kann mit dem Clustering-Koeffizienten ausgedrückt werden.
Der Clustering-Koeffizient gibt dabei an, wieviele der Nachbarknoten eines Knotens untereinander verbunden sind. Für ein soziales Netzwerk bedeutet das, ob die Freunde eines Benutzer selbst auch Freunde sind.

2.2.1 Lokaler Clustering-Koeffizient

Watts und Strogatz [JH98] führen den lokalen Clustering-Koeffizienten für einen Knoten i mit k_i Nachbarn ein. Für gerichtete Graphen ist dieser definiert als:

$$C_i = \begin{cases} \frac{d(i)}{k_i(k_i-1)}, & \text{falls } k_i > 1 \\ 0, & \text{sonst } 0. \end{cases}$$

In unserem Fall eines gerichteten Graphen werden als Nachbarn k_i nur jene Nachbarn j gezählt, die bidirektional mit dem Knoten i verbunden sind. Das bedeutet, es müssen Kanten $(i,j) \in E$ und $(j,i) \in E$ exisitieren.
Der lokale Clustering-Koeffizient berechnet sich demnach aus der Anzahl der existierenden Kanten zwischen den Nachbarn des Knotens i, dividiert durch die maximal mögliche Anzahl an Kanten. Der Wert kann also höchstens den Wert 1 annehmen, nämlich dann, wenn jeder Nachbar des Knotens auch zu allen anderen Nachbarn desselben Knotens verbunden ist. Der kleinst mögliche Wert ist demzufolge 0 und tritt dann auf, wenn kein Nachbar Kanten zu einem anderen Nachbarn des Knotens besitzt.
Der Clustering-Koeffizient drückt demnach die mittlere Wahrscheinlichkeit aus, dass 2 Nachbarn eines Knotens ebenfalls

miteinander verbunden sind.
Betrachtet man stattdessen ungerichtete Graphen, so können nur halb so viele Kanten zwischen Nachbarn existieren wie im gerichteten Fall. Dies erfordert entsprechend eine Halbierung des Nenners in der Definition des lokalen Clustering-Koeffizienten.

2.2.2 Durchschnittlicher Clustering-Koeffizient

Nachdem für jeden Knoten der lokale Clustering-Koeffizient bestimmt wurde, lässt sich nach Watts und Strogatz der durchschnittliche Clustering-Koeffizient C für einen Graphen G wie folgt bestimmen:

$$C = \frac{1}{|V|} \sum_{i=1}^{|V|} C_i.$$

Der durchschnittliche Clustering-Koeffizient für einen Graphen ist demnach die Summe der einzelnen lokalen Clustering-Koeffizienten dividiert durch die Anzahl an Knoten.

2.2.3 Transitivität

Eine alternative Definition des Clustering-Koeffizienten, die Transitivität, wurde von Newman, Watts und Strogatz eingeführt [NWS02]. Die Transitivität stellt den Anteil der verbundenen Trippel dar, die Dreiecke bilden. Ein Trippel ist dabei ein Teilgraph, bestehend aus drei Knoten, bei welchem jeder Knoten mindestens zu einem seiner Nachbarn eine Verbindung besitzt. Die Transitivität lässt sich wie folgt ausdrücken:

$$T = \frac{\text{3 x Anzahl der Dreiecke im Graphen}}{\text{Anzahl der Knotentrippel}}.$$

Der Faktor 3 lässt sich dadurch erklären, dass ein Dreieck aus drei solchen Knotentrippeln besteht. Wie bei der Definition des Clustering-Koeffizienten kann die Transitivität auch nur einen Wert zwischen 0 und 1 annehmen. Sie drückt somit, wie der Clustering-Koeffizient, die Wahrscheinlichkeit aus, dass zwei Nachbarn eines Knotens ebenfalls benachbart sind. Ein Unterschied im Gegensatz zu der vorherigen Definition ist, dass hier Knoten mit niedrigem Grad weniger stark ins Gewicht fallen [New03].
Analog für den gerichteten Fall definiert Newman die Transitivität wie folgt [New03]:

$$T = \frac{\text{6 x Anzahl der Dreiecke im Graphen}}{\text{Anzahl Wege der Länge 2}}.$$

Für den Nenner werden hierbei nicht mehr alle Knotentrippel gezählt, sondern alle gerichteten Wege der Länge 2. Ein Knotentrippel kann nun doppelt so viele Kanten aufweisen, wie bei dem ungerichteten Fall, was den Faktor 6 erklärt.

2.3 Cache

Ein Cache ist eine Art Pufferspeicher, durch den Zugriffe auf ein langsames Hintergrundmedium verringert werden. Ein Cache ist üblicherweise ein schnelles Speichermedium mit einer allerdings begrenzten Kapazität. Es können also nur eine begrenzte Anzahl an Einträgen gespeichert werden.
Von einem Cache-Hit spricht man, wenn ein Objekt aus einem Cache angefragt wird und sich dieses Objekt bereits in dem Cache befindet. So muss dieses Objekt nicht von dem langsameren Hintergrundmedium geladen werden, wodurch sich die Zugriffszeit deutlich verringert. Ein Cache-Miss bezeichnet demnach eine Abfrage eines Objektes, welches sich nicht in dem Cache befindet. Tritt ein Cache-Miss auf, muss das Objekt von dem Hintergrundmedium geladen werden. Deshalb ist das primäre Ziel beim Einsatz eines Caches, eine maximale Anzahl an Cache-Hits zu erreichen.
Ein weiterer wichtiger Aspekt, der die Performance des Caches sehr stark beeinflusst, ist die Strategie zur Ersetzung eines Cache-Eintrags. Da der Cache nur eine begrenzte Anzahl an Einträgen besitzt, müssen Einträge entfernt werden, falls der Cache voll belegt ist und Platz für einen neuen Eintrag gemacht werden muss. Einige dieser Strategien beschreiben wir in Kapitel 4.
Dabei können Caches sowohl in Hardware als auch in Software, wie in Kapitel 5 zu sehen, realisiert werden. Ein Beispiel für einen Hardware-Cache ist der Cache der Festplatte. Dieser Cache wird dazu benutzt, um bestimmte Daten der Festplatte zwischenzuspeichern und diese bei erneuter Abfrage gegebenenfalls schneller verfügbar zu haben. Ein weiterer Cache ist der Disk-Cache [Smi85]. Dieser wird vom Betriebssystem benutzt, um ähnlich dem Festplatten-Cache bestimmte Daten, die von der Festplatte gelesen wurden, zwischenzuspeichern. Dafür wird ein kleiner Teil des Arbeitsspeichers benutzt. Nachfolgend verwenden wir den Begriff Disk-Cache als Oberbegriff für den eigentlichen Disk-Cache und den Festplatten-Cache.

3 Verwandte Arbeiten

Es existieren zwei Kategorien von Ansätzen, um den Clustering-Koeffizient für einen Graphen zu berechnen. Die exakte Berechnung und die approximative Berechnung. Ein Problem der exakten Berechnung ist, dass sie sehr speicher- und rechenaufwendig ist. Mit der approximativen Berechnung wird deswegen versucht, eine möglichst gute Schätzung des Clustering-Koeffizienten für einen Graphen zu finden.

3.1 Exakte Berechnung

Eine Möglichkeit, den Clustering-Koeffizienten exakt zu bestimmen, benutzt die Matrixmultiplikation [CW90]. Bei diesem Verfahren wird der Graph als Adjazenzmatrix $A(G)$ aufgefasst. Wird nun für diese Matrix $A(G)^3$ bestimmt, kann anhand der Diagonalen der resultierenden Matrix die Anzahl der Dreiecke für jeden Knoten abgelesen werden [SW04]. Dieses Verfahren besitzt eine Laufzeit von $O(|V|^{2.37})$, wobei 2.37 den Matrixmultiplikationsexponenten ω beschreibt. Der Speicherverbrauch ist mit $O(|V|^2)$ allerdings sehr hoch, weshalb dieses Verfahren für große Graphen ungeeignet ist.
Ein weiteres Verfahren zur exakten Bestimmung des Clustering-Koeffizienten ist die Iteration über die gesamten Knoten [SW04][TKM09]. Bei dieser Methode wird für jeden Knoten i überprüft, wie viele Kanten zwischen seinen Nachbarn existieren. Dieses entspricht dann der Anzahl an Dreiecken, an denen i beteiligt ist. Die Laufzeit für diese Methode ergibt sich dabei aus der maximalen Anzahl an Kanten, die zwischen den Nachbarn eines Knotens existieren können. Für einen Knoten i ergibt sich für den ungerichteten Fall $d(i)(d(i)-1)/2$ Kanten und für den gerichteten Fall $d(i)(d(i)-1)$ Kanten. Somit ergibt sich für die Laufzeit $O(|V| \cdot max(d(i))^2)$, was wiederum $O(|V|^3)$ entspricht. Durch den Einsatz verschiedener Techniken (z.B. Hashing) für die Überprüpfung, ob zwei Nachbarknoten verbunden sind, lässt sich die Laufzeit für diese Methode weiter verbessern [SW05]. Befinden sich jedoch keine Knoten mit sehr hohen Graden in dem Graphen, zeigt sich diese Methode oft schneller als die Matrixmultiplikation und benötigt vor allem weniger Speicherplatz, da bei dieser Methode keine Matrixdarstellung nötig ist. Hierbei genügt es, die Daten als Adjazenzlisten zu speichern [SW04]. In Algorithmus 1 befindet sich diese Methode in Pseudocode.
Analog zu der Iteration über die Knoten ist es ebenfalls möglich, über die Kanten zu iterieren. Dabei wird für jede Kante überprüft, ob die Start- und Zielknoten einen gemeinsamen Nachbarn besitzen [SW05]. Die Laufzeit dieser Methode ist entsprechend $O(|E| \cdot max(d(i))^2)$, was $O(|E|^3)$ entspricht.
In [AYZ97] stellen Alon, Yuster und Zwick ein schnelles und exaktes Verfahren vor, um die Anzahl der Dreiecke in einem Graphen zu bestimmen. Dazu werden die Knoten in zwei Gruppen eingeteilt. Gruppe Eins beinhaltet all die Knoten, deren Grad $d(i)$ kleiner als ein definiertes Δ ist. Δ wird entsprechend des Exponenten der Matrixmultiplikation ω mit $\Delta = |E|^{\frac{\omega-1}{\omega+1}}$ gewählt. Gruppe Zwei beinhaltet dementsprechend alle Knoten, deren Grad größer ist als Δ. Es werden zunächst alle Dreiecke bestimmt, an denen ein Knoten aus Gruppe Eins beteiligt ist. Dazu werden alle Wege der Länge zwei untersucht, bei denen ein Knoten der ersten Gruppe den Mittelpunkt bildet. Anschließend wird überprüft, ob zwischen dem Start- und Endknoten des Weges der Länge zwei eine Kante besteht. Ist dies der Fall, besteht ein Dreieck. Anschließend müssen noch die Dreiecke gefunden werden, die aus Knoten der zweiten Gruppe bestehen. Die Laufzeit für diese Methode ist $O(|E|^{\frac{2\omega}{\omega+1}})$.

Nach [TKM09] und [SW04] ist der AYZ-Algorithmus aktuell die schnellste Möglichkeit, den exakten Clustering-Koeffizienten für einen Graphen zu berechnen.
Es zeigt sich, dass die schnellen Verfahren, wie der AYZ-Algorithmus, eine Matrixdarstellung des Graphen benutzen, um den exakten Clustering-Koeffizienten zu berechnen. Diese Darstellung hat allerdings den Nachteil, dass der gesamte Graph für die Berechnung im Speicher gehalten werden muss und die Berechnung somit auf kleinen Maschinen mit wenig Speicher nicht möglich ist. Im Gegensatz dazu kann die Berechnung mittels Iteration über die Knoten bzw. Kanten auch auf kleineren Maschinen durchgeführt werden. Bei diesem Verfahren ist die Graphendarstellung der auschlaggebende Faktor für den benötigten Speicher.

3.2 Approximative Berechnung

Wie beschrieben, ist die exakte Bestimmung des Clustering-Koeffizienten für große Graphen eine komplexe Aufgabe. Mit der Approximation wird deswegen versucht, den Clustering-Koeffizienten so genau wie möglich zu schätzen, um Speicher und Laufzeit zu sparen.
Bei den meisten der approximativen Verfahren wird für die Darstellung des Graphen ein Stream benutzt [BYKS02]. Dabei sind die beiden hauptsächlichen Stream-Arten der Kanten-Stream und der Inzidenz-Stream [TKM09]. Der Vorteil des Streaming-Ansatzes ist, dass hierbei nur ein einzelner Durchlauf über die Graphendaten nötig ist. Es ist somit nicht nötig, die gesamten Daten in den Speicher einzulesen [Tso08]. Der Streaming-Ansatz kann allerdings auch benutzt werden, wenn sich die kompletten Graphendaten im Speicher befinden.
Das Ziel dieser Arbeit ist es allerdings, den Wert exakt zu berechnen. Deswegen werden wir diese Methoden nicht genauer beschreiben. Einige Beispiele für die Verwendung des approximativen Ansatzes finden sich in [BYKS02] und [TKM09].

Algorithmus 1 Iteration über die Knoten eines Graphen

Eingabe: Graph $G = (V, E)$
Ausgabe: Anzahl an Dreiecken

```
1: Dreiecke = 0;
2: for v ∈ V do
3:     for alle Nachbarn u von v do
4:         if (v, u) ∈ E then
5:             Dreiecke++;
6:         end if
7:     end for
8: end for
9: return Dreiecke
```

4 Eigener Ansatz für die Berechnung des Clustering-Koeffizienten

In diesem Kapitel beschreiben wir unseren Ansatz zum exakten Berechnen des Clustering-Koeffizienten mittels verschiedener Berechnungsreihenfolgen und Caching-Strategien. Dazu geben wir in Abschnitt 4.1 einen Einblick in das Verfahren, mit welchem wir die Berechnung durchführen. In Abschnitt 4.2 stellen wir die verwendeten Caching-Strategien und in Abschnitt 4.3 die verwendeten Berechnungsreihenfolgen vor.

4.1 Verfahren zur exakten Berechnung der Clustering-Koeffizienten

Wie beschrieben, ist die Berechnung des exakten Clustering-Koeffizienten sehr speicher- und rechenintensiv. Vor allem, wie im vorherigen Kapitel beschrieben, benutzen die schnelleren Methoden, wie der AYZ-Algorithmus, eine Matrixdarstellung des Graphen. Die Darstellung besitzt jedoch einen Speicherverbrauch von $O(|V|^2)$, wodurch diese Darstellung nur auf Maschinen mit sehr viel Speicher möglich ist. Demzufolge ist diese Berechnung für kleinere Maschinen mit wenig Speicher nicht möglich.
Eine Möglichkeit den Clustering-Koeffizienten auch auf kleinen Maschinen zu berechnen, stellt der Streaming-Ansatz, wie im vorherigen Kapitel beschrieben, dar. Dieser Ansatz wird allerdings benutzt, um den Clustering-Koeffizient approximativ zu berechnen.
Die Idee hinter dieser Arbeit ist, die Berechnung des exakten Clustering-Koeffizienten auf den Originaldaten ähnlich dem Streaming-Ansatz durchzuführen. Die Originaldaten liegen in der Art vor, dass für jeden Knoten des Graphen, die Kanten seperat in Dateien gespeichert sind. Näheres zu den Daten beschreiben wir in Kapitel 6.
Um den Clustering-Koeffizienten zu bestimmen, kommt die Iteration über die Knoten des Graphen zum Einsatz. Wie beschrieben, wird bei diesem Verfahren für jeden Knoten überprüft, wieviele Kanten zwischen dessen Nachbarn existieren und somit der lokale Clustering-Koeffizient für einen Knoten bestimmt. Ein Problem, was folglich bei der Berechnung auf den Daten auftritt, ist, dass die Dateien für einen Knoten sehr oft von der Festplatte gelesen werden müssen, um auf die Existenz von Kanten zu prüfen. Auch ist es nicht möglich, die gesamten Dateien in den Speicher des Rechners zu laden, da dies die Speicherkapazität weit überschreiten würde.
Es kommt deswegen ein Cache zum Einsatz, mit welchem eine Anzahl an Knoten in den Speicher geladen und so die Zugriffszeit deutlich verringert werden kann. Um den Cache allerdings so effektiv wie möglich einzusetzen, ist es wichtig, die maximale Hit-Anzahl zu erreichen. Durch die Reihenfolge, in der die Knoten bearbeitet werden, soll die Hit-Anzahl erhöht und dementsprechend die Zugriffe auf den Cache erhöht werden. Dies würde zur Folge haben, dass die Zugriffe auf die langsamere Festplatte verringert werden, was die Bearbeitungzeit verbessern würde. Wie beschrieben, wird zur Berechnung die Iteration über die Knoten verwendet. Um die Reihenfolge der Knoten zu ändern, muss dazu in Algorithmus 1 beispielsweise die 2. Zeile geändert werden.
Eine Möglichkeit, wie die Reihenfolge der Knoten bestimmt wird, ist die Graphen-Traversierung. Bei der Graphen-Traversierung wird, beginnend von einem Startknoten, einer durch die Traversierung vorgegebenen Route durch den Graphen gefolgt. Eine weitere Möglichkeit, die Reihenfolge zu definieren, ist beispielsweise die Knoten anhand deren Grades zu sortieren und dementsprechend zu bearbeiten.
Das Phänomen, was hierbei benutzt wird, ist dass in einem Graphen aus einem sozialen Netzwerk die Verbundenheit zwischen Nachbarn eines Knotens höher ist als bei Graphen aus anderen Netzwerken [NP03]. Wird für einen Knoten der lokale Clustering-Koeffizient bestimmt, müssen hierzu ebenfalls die Nachbarn des Knotens betrachtet und somit geladen werden. Wird nun anstatt zufällig den nächsten Knoten zu wählen, der berechnet werden soll, einer der Nachbarn gewählt, ist die Wahrscheinlichkeit umso höher, dass viele dessen Nachbarn bereits bei der vorherigen Berechnung geladen wurden und sich bereits in dem Cache befinden. Dies würde verhindern, dass diese Knoten erneut von der Festplatte geladen werden müssten, was die Berechnungszeit deutlich verlangsamen würde.
Als Reihenfolge für die Abarbeitung der Knoten und für das Caching, können hierbei einige Strategien verwendet werden. In den folgenden Abschnitten beschreiben wir deswegen die Strategien, die wir für unseren Ansatz verwenden.

4.2 Caching-Strategien

Nachfolgend stellen wir die Caching-Strategien, die in unserem Verfahren zur exakten Berechnung verwendet werden, vor.

4.2.1 Least Recently Used

Bei der LRU (Least Recently Used) Strategie wird, wenn der Cache voll ist, immer jener Eintrag ersetzt, welcher am längsten nicht benutzt wurde. Dazu werden die Einträge des Caches in eine Liste sortiert. Je nach Implementierung ist entweder der Kopf oder das Ende der Liste das Element, welches am längsten nicht benutzt wurde. Wird ein Element benötigt, welches sich noch nicht in dem Cache befindet, wird es von dem Hintergrundmedium, in unserem Fall der Festplatte, geladen und an den Kopf bzw. an das Ende der Liste angehängt. Falls der Cache voll belegt ist, wird vorher

der Eintrag entfernt, der an dem Kopf oder an dem Ende der Liste steht, um den neuen Eintrag speichern zu können.
Wird ein Element abgefragt, was bereits in dem Cache existiert, wird dieses, um die Sortierung beizubehalten, in der Liste an die erste bzw. letzte Stelle verschoben. So kann gewährleistet werden, dass immer jenes Element entfernt wird, welches am längsten nicht benutzt wurde.
Wir vermuten, dass mit dieser Caching-Strategie die besten Ergebnisse für das Caching erzielt werden. Durch das Sortieren der Einträge nach deren letzten Zugriff, werden hierbei jene Knoten behalten, die vor kurzem benutzt wurden. Wie beschrieben, wollen wir mit unserem Ansatz die Eigenschaft der Graphen aus sozialen Netzwerke ausnutzen, dass die Verbundenheit der Nachbarn untereinander größer ist als bei anderen Graphen-Arten. Wird beispielsweise eine Nachbarschaft bearbeitet, werden bestimmte Knoten häufig gelesen. Dementsprechend befinden sich diese Knoten sehr weit vorne bzw. hinten in der Liste und werden somit in dem Cache gehalten. Dadurch sollte die Hit-Rate mit dieser Strategie sehr gut sein.

4.2.2 First In First Out

Wie der Name FIFO (First In First Out) bereits aussagt, wird bei dieser Strategie immer der Eintrag ersetzt, welcher sich am längsten in dem Cache befindet. Auch hier erfolgt die Sortierung anhand einer Liste. Der einzige Unterschied zu der LRU-Strategie ist, dass bei einer Abfrage eines existierenden Eintrages im Cache der Eintrag nicht an den Kopf bzw. das Ende der Liste verschoben wird. So wird, falls der Cache voll belegt ist, immer der Eintrag entfernt, der sich am längsten in dem Cache befindet.
Die FIFO-Strategie ist demnach im Grunde der LRU-Strategie ähnlich, nur das bei dieser die Einträge bei einem Hit nicht neu sortiert werden. Es ist demnach einer der einfachsten Lösungen einen Cache zu organisieren. Deswegen vermuten wir, dass aufgrund der fehlenden Sortierung, die Hit-Raten mit dieser Strategie gerringer sind, als mit der LRU-Strategie.

4.2.3 Least Frequently Used

LFU (Least Frequently Used) ist eine Strategie, bei der das Element erstzt wird, welches am wenigsten benutzt wurde. Dazu wird jedem Eintrag des Caches ein Hit-Count zugewiesen. Der Hit-Count ist dabei ein einfacher Zähler, der bei jedem Aufruf eines Cache-Eintrages um 1 erhöht wird. Muss ein Eintrag aufgrund des vollen Caches entfernt werden, wird der Eintrag entfernt, welcher den kleinsten Hit-Count besitzt.
Ein Problem, dass die Hit-Raten dieser Strategie negativ beeinflussen könnte, ist das Blockieren des Caches. Es kann zum Beispiel vorkommen, dass ein Eintrag zu einer bestimmten Zeit sehr oft verwendet wird und dementsprechend einen hohen Hit-Count besitzt. Wird dieser Eintrag danach nicht mehr benötigt, würde dieser Eintrag trotzdem in dem Cache, aufgrund des hohen Hit-Countes, verbleiben. Um diesem Problem entgegen zu wirken, kann zum Beispiel periodisch nach einer bestimmten Zeit, der Hit-Count der Einträge reduziert werden, die in dieser Zeitperiode nicht benutzt wurden. Somit kann das Problem zwar nicht komplett umgangen werden, da unbenutzte Einträge mit einem hohen Hit-Count trotzdem so lange in dem Cache verbleiben, bis deren Hit-Count so weit verringert wurde, bis sie aus dem Cache entfernt werden. Aber es stellt eine deutliche Verbesserung mit Blick auf die Hit-Rate im Vergleich zu der Variante ohne die Reduzierung der Hit-Counts.
Durch das Sortieren nach der Häufigkeit der Benutzung eines Eintrages, vermuten wir das sich damit relativ gute Hit-Raten erzielen lassen. Denn hierbei werden die Knoten in dem Cache behalten, die häufig benutzt werden und somit die Wahrscheinlichkeit groß ist, dass diese auch weiterhin oft gelesen werden. Allerdings könnte das Blockierungsproblem die Hit-Raten hierbei sehr stark beeinflussen.

4.2.4 Weighted

Weighted ist ein eigener Entwurf für eine Caching-Strategie im Hinblick auf die Verwendung bei der Berechnung des Clustering-Koeffizienten auf einem Graphen.
Durch die Weighted-Strategie soll der Fakt ausgenutzt werden, dass Knoten mit einem hohen Grad öfter gelesen werden als Knoten mit niedrigerem Grad. Deshalb werden die Einträge bzw. die Knoten, die in dem Cache gespeichert sind, anhand deren Grades gewichtet, sodass Knoten mit einem hohen Grad länger in dem Cache verbleiben.
Als Grundlage dient hierbei die LFU-Strategie. Anstatt nur den Hit-Count zu betrachten, wird jedem Eintrag des Caches ein Gewicht zugeordnet. Das Gewicht besteht dabei aus dem eigentlichen Hit-Count der LFU-Strategie multipliziert mit dem Grad des Knoten, sprich dem Eintrag des Caches. Dementsprechend besitzen Einträge bzw. Knoten mit hohem Grad ein höheres Gewicht und werden deshalb bevorzugt. Ist der Cache voll belegt und es muss ein Eintrag entfernt werden, wird entsprechend der Eintrag entfernt, welcher das kleinste Gewicht aufweist.
Auch bei dieser Strategie kann allerdings das Problem auftreten, dass ein Eintrag den Cache blockieren kann. Deshalb wird auch hier periodisch nach einer gewissen Zeit der Hit-Count jener Einträge verringert, welche in der Zeit nicht benutzt wurden.

4.2.5 Random

Bei der Zufalls-Strategie wird keine Sortierung der Einträge vorgenommen. Muss ein Eintrag entfernt werden, wird zufällig ein Eintrag des Caches ausgewählt und entfernt. Diese Strategie ist demnach die trivialste Lösung um einen Cache zu organisiern.

4.3 Reihenfolgen der Abarbeitung

Nachfolgend beschreiben wir die Abarbeitungsreihenfolgen, die in unserem Verfahren zur exakten Berechnung verwendet werden.

4.3.1 Breadth-first search

Bei der Breitensuche (BFS - Breadth-first search) werden für einen gegebenen Graphen $G = (V, E)$ und einen Startknoten $s \in V$ systematisch all die Knoten durchlaufen, welche von dem Startknoten aus erreichbar sind. Die Reihenfolge, in der die Knoten besucht werden, wird von einer Warteschlange geregelt. Der Zugriff auf die Warteschlange erfolgt durch das FIFO-Prinzip, was bedeutet, dass der erste Knoten aus der Warteschlange entnommen wird und die Nachbarn des Knotens an das Ende der Warteschlange angehängt werden. So werden zunächst alle Nachbarn des Startknoten s besucht und anschließend die nächsttiefere Ebene nach dem selben Prinzip abgearbeitet [CLRS07]. Abbildung 3 zeigt anhand eines Beispielgraphen eine BFS-Traversierung von Startknoten 1. Die Nummerierungen geben hierbei die Reihenfolge an, wann ein Knoten besuch wird. Die gestrichelten Kanten werden hierbei nicht benutzt, da die über sie zu erreichenden Knoten bereits besucht wurden.

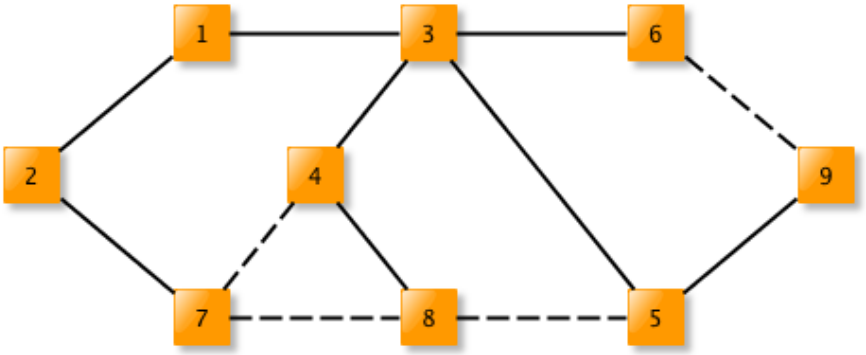

Abbildung 3: Beispielhafte BFS-Traversierung

Eine Erweiterung dieser Abarbeitungsreihenfolge ist die Order-Strategie. Bei der einfachen BFS-Traversierung wird die Reihenfolge, in der die Knoten einer bestimmten Ebene durchlaufen werden, zufällig bestimmt. Beispielsweise hätte in Abbildung 3 die Reihenfolge auch folgendermaßen aussehen können: 1,3,2,5,4,7,6,8,9. Die Order-Strategie sortiert hierbei nun die Knoten einer Ebene nach dem Grad der Knoten. So würde in Abbildung 3 immer der Knoten 3 als nächstes nach dem Startknoten bearbeitet werden, da dieser in der Wartschlange bestehend aus Knoten 2 und 3 den größten Grad besitzt. Die komplette Reihenfolge könnte demnach wie folgt aussehen: 1,3,2,4,5,7,6,8,9.
Durch BFS soll die Abarbeitungsreihenfolge so organisiert werden, dass immer die Nachbarn der zuvor bearbeiteten Knoten als nächstes bearbeitet werden. Wie bereits beschrieben, soll hierbei wieder die starke Verbundenheit der Nachbarn untereinander ausgenutzt werden. Wird also für einen Knoten der lokale Clustering-Koeffizient berechnet, müssen dazu entsprechend dessen Nachbarn geladen werden. Wird nun der nächste Nachbarknoten berechnet, ist die Wahrscheinlichkeit, dass bereits einige dessen Nachbarn bei der vorherigen Berechnung geladen wurden, sehr groß. Mit der Order-Strategie soll dies weiter verbessert werden. Die Idee der zusätzlichen Order-Strategie ist, dass pro Ebene zunächst die Knoten mit großem Grad berechnet werden. So müssen bereits zu Beginn sehr viele Knoten in den Cache geladen werden und somit steigt die Wahrscheinlichkeit, bei späteren Berechnungen viele Hits zu erreichen.
Durch diese Reihenfolge sollten sich gute Hit-Raten mit dem Einsatz eines Caches erzielen lassen. Ein Problem hierbei könnte allerdings sein, dass die verschiedenen Ebenen sehr groß werden. Als Beispiel in Abbildung 3 besitzt die erste Ebene lediglich einen Knoten. Die zweite Ebene demnach 2 Knoten und die dritte Ebene bereits 4 Knoten. Je mehr Knoten deshalb in einer Ebene sind, desto weniger Verbundenheit werden die Knoten dieser Ebene miteinander aufweisen. Somit müssten wieder mehr Knoten von der Festplatte geladen werden, was die Hit-Raten beeinflussen.

4.3.2 Depth-first search

Bei der Tiefensuche (DFS - Depth-first search) wird für einen gegebenen Graphen $G = (V, E)$ und einen Startknoten $s \in V$ zunächst einem Pfad in die Tiefe des Graphen gefolgt. Ist die tiefste Ebene des Graphen erreicht, sprich der aktuelle Knoten hat keine Nachbarn, die noch nicht besucht wurden, wird ein Schritt des Pfades zurück gegangen und der nächste, noch unbekannte Knoten besucht. Anders als bei der Breitensuche wird bei der Tiefensuche die Reihenfolge, in der die Knoten besucht werden, durch einen Stack (Kellerspeicher) bestimmt. Der Stack arbeitet nach dem Last-In-First-Out-Prinzip (LIFO), bei welchem der Knoten entnommen wird, der als letztes in den Stack gelegt wurde [CLRS07]. In Abbildung 4 ist anhand eines Beispielgraphen die DFS-Traversierung von Startknoten 1 zu sehen. Auch hier geben die Nummerierungen die Reihenfolge an und eine gestrichelte Kante signalisiert eine Kante die nicht benutzt wird, da die über sie zu erreichenden Knoten bereits besucht wurden.

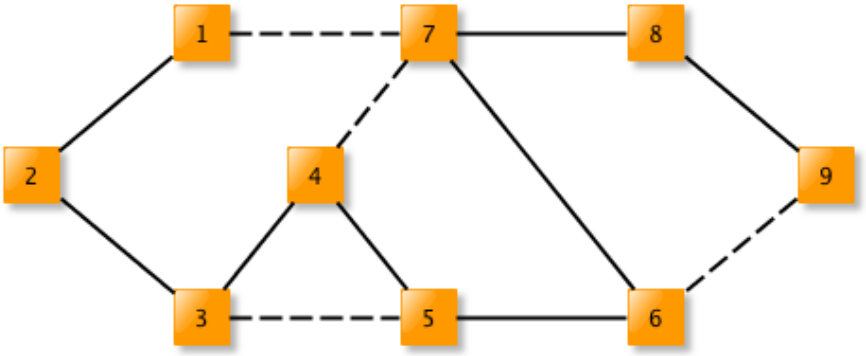

Abbildung 4: Beispielhafte DFS-Traversierung

Auch bei DFS wird der nächste zu berechnende Knoten zufällig bestimmt. Beispielsweise hätte anstatt Knoten 2 auch Knoten 7 als nächstes nach dem Startknoten bearbeitet werden können. Deshalb kann ebenfalls wie bei BFS die Order-Strategie hinzugefügt werden. Mit der Order-Strategie wäre Knoten 7 demnach immer der Nachfolgeknoten des Startknotens, da Knoten 7 den größten Grad aufweist. Eine mögliche Abarbeitungsreihenfolge mit Order-Strategie könnte somit wie folgt aussehen: 1,7,6,5,3,4,2,9,8.
Mit der DFS-Abarbeitungsreihenfolge erwarten wir geringere Hit-Raten im Vergleich zur BFS-Abarbeitungsreihenfolge. Wir vermuten, dass bei dieser Strategie die Eigenschaft der starken Verbundenheit der Nachbarn nicht so gut ausgenutzt werden kann wie mit BFS, denn bei DFS wird nur jeweils ein Nachbarknoten bearbeitet.

4.3.3 BFS-Bubble

Die BFS-Bubble-Abarbeitungsreihenfolge ist wie der Weighted-Cache ein eigener Entwurf. Die Idee dieser Abarbeitungsreihenfolge ist, das Problem der großen Ebenen des BFS zu umgehen.
Diese Abarbeitungsreihenfolge ist eine Art Mix aus BFS und DFS. Für einen gegebenen Graphen $G = (V, E)$ wird wie bei BFS und DFS ein Startknoten $s \in V$ zufällig bestimmt. Nachdem der Startknoten bearbeitet wurde, werden als nächstes alle dessen Nachbarn durchlaufen. Bis zu diesem Schritt verhält sich die BFS-Bubble-Abarbeitungsreihenfolge analog zu der BFS-Abarbeitungsreihenfolge. Der Unterschied zu BFS besteht nun darin, dass in dem nächsten Schritt nicht alle Nachbarn der Nachbarknoten besucht werden, sondern nur die Nachbarn eines Knotens. Es werden sozusagen Blasen über bestimmte Knoten und deren Nachbarn aufgespannt, die als nächstes besucht werden. So wird wie bei der DFS-Abarbeitungsreihenfolge zunächst einem Pfad in die Tiefe des Graphen gefolgt, bis ein Knoten erreicht ist, der keine weiteren ungesehenen Nachbarn besitzt. Ist dies der Fall, wird analog zu DFS ein Schritt des Pfades zurück gegangen und der nächste noch unbesuchte Knoten besucht.
Um die Reihenfolge, in der die Knoten besucht werden, zu realisieren, wird bei dieser Abarbeitungsreihenfolge jeweils eine Warteschlange nach FIFO-Prinzip und ein Stack benutzt. Abbildung 5 zeigt eine Traversierung mit BFS-Bubble anhand eines Beispielgraphen mit Startknoten 1. Wieder geben die Nummerierungen die Reihenfolge an und die gestrichelten sind Kanten, die nicht benutzt werden, da die über sie zu erreichenden Knoten bereits besucht wurden. Auch bei dieser Abarbeitungsreihenfolge wird der als nächstes zu berechnende Knoten per Zufall bestimmt. Deswegen kann hierbei ebenfalls die Order-Strategie hinzugefügt werden und somit die interne Reihenfolge der Knoten sortiert werden.
Durch diese Abarbeitungsreihenfolge glauben wir, sehr gute Ergebnisse in Bezug auf die Hit-Raten erzielen zu können. Dadurch, dass jeweils nur die Nachbarn eines Knotens berechnet werden, werden zu große Ebenen vermieden und die

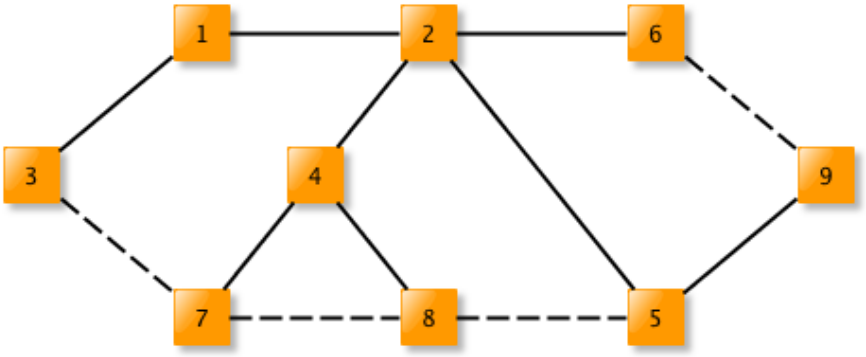

Abbildung 5: Beispielhafte BFS-Bubble-Traversierung

Verbundenheit der Nachbarn untereinander gut ausgenutzt.

4.3.4 Sorted-by-Degree

Bei der Abarbeitungsreihenfolge anhand des Grades der Knoten wird die Reihenfolge, in der die Knoten bearbeitet werden, durch den Grad derer bestimmt. Dazu werden alle Knoten anhand deren Grades, beginnend mit dem größten, in eine Warteschlange mit FIFO-Prinzip einsortiert. Es wird nun immer der erste Knoten aus der Wartschlange entfernt und dieser bearbeitet. Besitzt die Warteschlange keine weiteren Einträge, ist die Berechnung abgeschlossen. In Abbildung 6 ist die Reihenfolge für Sorted-by-Degree anhand eines Beispielgraphen zu sehen.

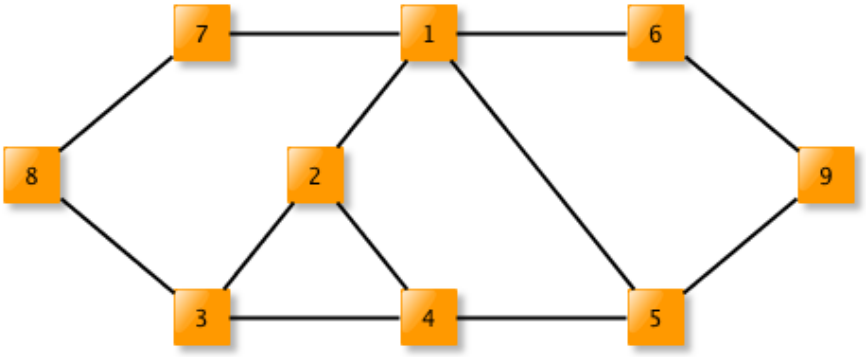

Abbildung 6: Beispielhafte Sorted-by-Degree-Abarbeitungsreihenfolge

Die Idee hinter dieser Abarbeitungsreihenfolge ist, dass durch die bevorzugte Berechnung der Knoten mit großem Grad dementsprechend viele Knoten geladen werden müssen. Diese müssten bei späteren Berechnungen somit nicht noch einmal geladen werden. Ein Problem bei dieser Reihenfolge könnte sein, dass es viele große Knoten gibt, die nicht miteinander verbunden sind. So müssten trotz der vielen Knoten in dem Cache wieder viele von der Festplatte geladen werden. Das würde sich natürlich auf die Hit-Rate auswirken. Wir gehen deshalb von mittelmäßigen Ergebnissen für diese Reihenfolge aus.

4.3.5 Random

Bei der zufälligen Abarbeitungsreihenfolge besteht keine Ordnung in der Reihenfolge, wie die Knoten bearbeitet werden. Wann ein Knoten besucht wird, hängt alleine vom Zufall ab.
Dies ist die trivialste Reihenfolge, alle Knoten zu bearbeiten. Da hierbei keine Ordnung vorliegt, gehen wir hier ganz klar von den schlechtesten Ergebnissen in Bezug auf die Hit-Rate aus.

5 Implementierung

Die Grundidee für die Implementierung ist, dass das Programm, mit dem der Clustering-Koeffizient berechnet werden soll, modular aufgebaut wird. Das bedeutet, dass per Parametereingaben bestimmt werden kann, welche Abarbeitungsreihenfolge und welche Caching-Strategie benutzt wird. Ebenfalls lässt sich per Parameter für BFS, DFS und BFS-Bubble bestimmen, ob die Order-Srategie verwendet wird und wie viel Speicherplatz dem Cache zugewießen wird. Zusätzlich wird die Start-ID übergeben, von welchem Knoten aus die Berechnung gestartet wird.
Die Struktur des Programmes lässt sich grob in die 3 folgenden Teile aufteilen:

1. der Hauptteil mit der Berechnung des Clustering-Koeffizienten,
2. der Teil, der die Reihenfolge der Knoten bestimmt und
3. der Cache mit den jeweiligen Caching-Strategien.

Diese Unterteilung ist möglich, da sich die eigentliche Berechnung des Clustering-Koeffizienten bei den verschiedenen Kombinationen aus Abarbeitungsreihenfolge und Caching-Strategie nicht unterscheiden. Es ändert sich lediglich die Reihenfolge in der die Knoten bearbeitet werden und die Caching-Strategie. Die verschiedenen Abarbeitungsreihenfolgen und Caching-Strategien sind in dem vorherigen Kapitel beschrieben.
Der Programmaufruf erfolgt dabei aus einem Bash-Script. In diesem Bash-Script wird pro Testlauf aller Kombinationen, eine zufällige Start-ID generiert und das Programm für alle Kombinationen aus Abarbeitungsreihenfolge und Caching-Strategie mit dieser Strart-ID aufgerufen. Dieses Bash-Script werden wir hier nicht näher erläutern. Es besteht im Grunde nur aus den verschiedenen Aufrufen des Programmes.

In diesem Kapitel werden wir die Implementierung der verschiedenen Teile näher beschreiben. Als Implementierungssprache wurde die objektorientierte Sprache Java gewählt. Ein Vorteil dieser Sprache ist die Plattformunabhängigkeit. Somit ist es möglich, Programme auf verschiedenen Plattformen wie Windows, Mac OS oder Linux laufen zu lassen, ohne Änderungen an dem Code vornehmen zu müssen [Ull11]. Der gesammte Quellcode inklusive Bash-Script befindet sich zusätzlich auf der beiliegenden DVD.

5.1 Caching-Strategien

Der Cache ist so implementiert, dass ein Interface *AllCache.java* die benötigten Funktionen bereitstellt und je nach Wahl der Caching-Strategie das Interface implementiert wird. Das Interface *AllCache.java* ist in Listing 1 definiert.

```
public interface AllCache {
        ...
        public Node getNode(int nodeID);
        ...
}
```

Listing 1: Interface AllCache.java

Zusätzlich zu der Funktion *getNode()* sind noch weitere Funktionen definiert, die allerdings nur für statistische Zwecke benutzt werden. Beispielsweise die Funktion *getHits()*, die die aktuelle Anzahl an Hits ausgibt. Mit der Funktion *getNode()* wird ein eindeutiger Knoten bei dem Cache abgefragt. Hierzu muss erwähnt werden, dass die Knoten des Graphen alle einen eindeutigen Identifier besitzen, über welchen die Knoten adressiert werden können. Sollte sich der gesuchte Knoten nicht in dem Cache befinden, greift eine Funktion des Caches, welche den Knoten von der Festplatte lädt und anschließend zurück gibt. Abbildung 7 zeigt den schematischen Ablauf einer Cache-Anfrage als UML-Diagramm.

5.1.1 Node

Node.java ist ein Objekt, in welchem alle Informationen bezüglich eines Knotens gespeichert werden. Die Datenstruktur des *Node* Objektes ist in Listing 2 zu sehen.

```
public class Node {
        private int id;
        private Set<Integer> in;
        private Set<Integer> out;
        ...
}
```

Listing 2: Objekt Node.java

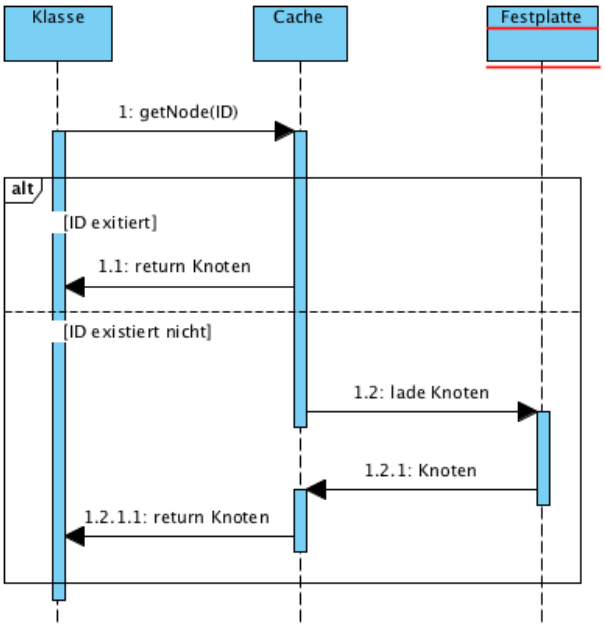

Abbildung 7: Schematischer Ablauf einer Cache-Anfrage

Die *id* ist hier der eindeutige Identifier eines Knotens und besteht aus einer Ganzzahl. *in* und *out* beschreiben die Kanten, die ein Knoten besitzt. Dabei finden sich in *in* bzw. *out* jeweils die Identifier der Knoten, welche eine Verbindung zu diesem Knoten besitzen. Der Graph, welcher für die Berechnungen benutzt wird, ist ein gerichteter Graph, weshalb die Kanten in eingehende und ausgehende Kanten unterteilt sind. Die Kantenlisten werden hierbei duch ein *HashSet (java.util.HashSet)* realisiert, um die Überprüfungen, ob ein Knoten eine Verbindung zu einem bestimmten Knoten besitzt, zu beschleunigen. Ein *HashSet* ist eine Datenstruktur, bei der die Einträge in einer Hash-Tabelle gespeichert werden und somit eine konstante Zeit $O(1)$ für eine Abfrage eines Eintrages möglich ist.
Die Datenzugriffe auf ein solches Objekt werden über *getter* und *setter* Methoden realisiert. Ebenfalls besitzt ein Node Objekt die Methoden zum Lesen und Schreiben eines Knotens von bzw. auf die Festplatte.

5.1.2 CacheEntry

Das Objekt *CacheEntry.java* ist das eigentliche Objekt, welches in dem Cache gespeichert wird. Die Struktur dieses Objektes ist in Listing 3 beschrieben.

```
public class CacheEntry {
        private CacheEntry prev; // pointer to previous
        private CacheEntry next; // pointer to next
        private Node value;
        private int hitCount;
        ...
}
```

Listing 3: Objekt CacheEntry.java

value ist hierbei der eigentliche Knoten als *Node* Objekt. Der *hitCount* ist eine einfache Zählervariable, die für die Implementierung des LFU- und Weighted-Cache wichtig ist. Wie üblich erfolgt hier der Zugriff auf die Daten ebenfalls per *getter* und *setter* Methoden.
Wie im vorherigen Kapitel beschrieben, müssen die Einträge eines Caches sortiert werden, um je nach Strategie das richtige Element zu entfernen, falls der Cache voll belegt ist. Diese Liste wird hier durch eine doppelt verkettete List realisiert. *next* und *prev* sind hierzu Pointer auf das vorherige bzw. nachfolgende Element der Liste. Eine doppelt verkettete Liste hat den Vorteil, dass in der Liste vor und zurück gegangen werden kann, was bei der eigentlichen Implementierung des Caches von Vorteil ist.
Mit der Funktion *getSizeInByte()* kann die Größe des Objektes in Byte abgefragt werden.

5.1.3 LRU-Cache

Die Klasse *LRUCache.java* implementiert das Interface *AllCache.java*. Um die Daten, sprich die Objekte der Klasse *CacheEntry* zu speichern, wird wie bei allen Cache-Implementierungen eine *HashMap (java.util.HashMap)* benutzt. Dazu wird der Identifier eines Knotens als Schlüssel und das *CacheEntry* Objekt als Wert benutzt. Wie auch das *HashSet* hat die *HashMap* den Vorteil, dass Abfragen eines bestimmten Eintrages in konstanter Zeit $O(1)$ möglich sind.
Um einen Knoten von dem Cache abzufragen, wird die Funktion *getNode()* aufgerufen. Die Funktion ist in Listing 4 zu sehen.

```
public Node getNode(int id) {
        // get entry from map
        CacheEntry entry = map.get(id);
        if(entry == null) {
                // if entry == null --> read node from hdd
                entry = new CacheEntry(Node.read(path + nodePathes.get(id)));
                // is cache full --> delete eldest entry
                while(isCacheFull(entry.getSizeInByte())) {
                        // get eldest entry
                        CacheEntry deleteEntry = tail.getPrev();
                        // delete entry from map
                        map.remove(deleteEntry.getValue().getID());
                        // decrease current cache size
                        CURRENT_SIZE -= deleteEntry.getSizeInByte();
                        // remove eldest entry
                        remove(deleteEntry);
                }
                // add new entry
                addFront(entry);
                map.put(id, entry);
                // increase current cache size
                CURRENT_SIZE += entry.getSizeInByte();
        }
        else {
                // entry was in map
                accessed(entry);
        }
        return entry.getValue();
}
```

Listing 4: Funktion getNode() der LRU Implementierung

Zunächst wird hierbei überprüft, ob sich der gesuchte Knoten bereits in der *HashMap* befindet. Ist dies nicht der Fall, wird der Knoten von der Festplatte eingelesen und in ein neues *CacheEntry* Objekt gespeichert. Da der Cache auf eine bestimmte Größe beschränkt ist, muss nun überprüft werden, ob für den neuen Eintrag noch genügend Speicherplatz vorhanden ist. Die aktuelle Größe des Caches wird in *CURRENT_SIZE* gespeichert. Ist dies nicht der Fall, wird wie für die LRU-Strategie beschrieben, das Element entfernt, welches am längsten nicht benutzt wurde. Bei unserer Implementierung befindet sich dieses Element am Ende der Liste. Mit *tail.getPrev()* kann eben dieses Element aus der Liste gelesen werden. *head* und *tail* sind hierbei Ankerpunkte, die den Kopf bzw. das Ende der Liste markieren. Diese Prozedur wird nun solange wiederholt, bis genügend freier Speicher für das neue *CacheEntry* Objekt vorhanden ist. Anschließend wird der neue Eintrag an den Kopf der Liste einsortiert, in die *HashMap* gespeichert und die aktuelle Größe des Caches um die Größe des neuen Elementes erhöht.
Befindet sich der gesuchte Knoten bereits in der *HashMap*, wird der vorherige Teil übersprungen. Um die Sortierung der Liste allerdings aufrecht zu erhalten, wird lediglich das *CacheEntry* Objekt an den Kopf der Liste verschoben. Dazu müssen lediglich die Pointer der *CacheEntry* Objekte verändert werden. Die hierzu benötigten Funktionen sind in Listing 5 aufgeführt.
Abschließend wird der gesuchte Knoten als *Node* Objekt zurück gegeben.

```
private void remove(CacheEntry entry) {
        if (entry == head || entry == tail) {
                return; // error
        }
```

```
        // set new pointers between previous and next
        entry.getPrev().setNext(entry.getNext());
        entry.getNext().setPrev(entry.getPrev());
}

private void addFront(CacheEntry entry) {
        // add the new entry at head of the list
        head.getNext().setPrev(entry);
        entry.setPrev(head);
        entry.setNext(head.getNext());
        head.setNext(entry);
}

private void accessed(CacheEntry entry) {
        if (entry.getPrev() != head) {
                // remove from its current location in the list
                remove(entry);
                // add to the list
                addFront(entry);
        }
}
```

Listing 5: Funktionen zum Sortieren der Liste für LRU Implementierung

5.1.4 FIFO-Cache

Die Implementierung des FIFO-Cache ist analog zu der des LRU-Caches. Der einzige Unterschied besteht darin, dass bei einem Hit, sprich der Knoten befindet sich bereits in der *HashMap*, das *CacheEntry* Objekt nicht neu in die Liste einsortiert wird.

5.1.5 LFU-Cache

Auch die Implementierung des LFU-Caches beruht auf dem gleichen Prinzip wie der LRU-Cache. Der Unterschied bei dieser Implementierung besteht darin, dass zur Sortierung der Liste nun der *hitCount* des *CacheEntry* Objektes benutzt wird.
Anders als bei dem LRU-Cache wird ein neuer Eintrag nicht an den Kopf eingefügt, sondern nach dessen *hitCount* in die Liste einsortiert. Für einen neuen Eintrag beträgt dieser den Wert 1. Wird ein existierender Knoten angefragt wird der *hitCount* um den Wert 1 erhöht und in der Liste an die richtige Position einsortiert.
Um wie in Kapitel 4 beschrieben das Blockieren von Einträgen zu verhindern, werden pro 10.000 Cache-Anfragen der *hitCount* all derjenigen *CacheEntry* Objekte verringert, die in dieser Zeit nicht benutzt wurden. Dazu müssen allerdings die IDs derer Knoten gespeichert werden, welche in dieser Zeit benutzt wurden. Die Implementierung der Sortierung der Liste für den LFU-Cache ist in Listing 6 zu sehen.

```
private void sort(CacheEntry entry) {
        if ((entry.getPrev() == head)
                && (entry.getHitCount() >= entry.getNext().getHitCount()))
                return; // first element and biggest hitCount
        if ((entry.getNext() == tail)
                && (entry.getHitCount() <= entry.getPrev().getHitCount()))
                return; // last element and smallest hitCount

        remove(entry);

        if (entry.getHitCount() > entry.getPrev().getHitCount())
                sortBackward(entry);
        else
                sortForward(entry);
}
```

Listing 6: Funktion sort() der LFU Implementierung

Mit *sortBackward()* und *sortForward()* werden die Einträge an die richtige Position der Liste sortiert. Hierbei kommt nun auch der Vorteil einer doppelt verketteten Liste zum tragen. Dadurch ist es möglich, in der Liste vor und zurück zu gehen. In Listing 7 ist als Beispiel die Funktion *sortBackward()* zu sehen.

```
private void sortBackward(CacheEntry entry){
        CacheEntry current = entry.getPrev();
        // find new position
        while( (current != head) && (current.getHitCount() < entry.getHitCount()) ){
                current = current.getPrev();
        }

        // get the pointers from the elements
        entry.setPrev(current);
        entry.setNext(current.getNext());
        // insert the new
        current.getNext().setPrev(entry);
        current.setNext(entry);
}
```

Listing 7: Funktion sortBackward() der LFU Implementierung

Allerdings ist diese Sortierung im Vergleich zu der des LRU-Caches mit mehr Aufwand verbunden, da hier die Liste oft durchlaufen werden muss. Ebenfalls das Verringern der unbenutzten Einträge verursacht einen Mehraufwand, der mit Berechnungszeit verbunden ist.

5.1.6 Weighted-Cache

Die Implementierung des Weighted-Cache ist analog zu der des LFU-Caches. Der Unterschied besteht darin, dass zur Positionsfindung nicht alleine der *hitCount* benutzt wird, sondern ein Wert aus $hitCount \cdot d(i)$. Also dem *hitCount* multipliziert mit dem Grad des Knotens, der in dem *CacheEntry* Objekt gespeichert ist. Demnach sieht beispielsweise die *while*-Schleife der *sortBackward* Funktion folgendermaßen aus:

```
// find new position
while( (current != head) && ( (current.getHitCount() * current.getValue().getDegree())
        < (entry.getHitCount() * entry.getValue().getDegree()) )){
        current = current.getPrev();
}
```

Listing 8: while-Schleife der Funktion sortBackward() der Weighted Implementierung

5.1.7 Random-Cache

Die Implementierung des Random-Cache ist im Grunde analog zu der des FIFO-Caches. Der Unterschied besteht darin, dass ein neuer Eintrag anstatt am Kopf der Liste an einer zufälligen Position der Liste einsortiert wird. Dazu wird für jeden neuen Eintrag ein zufälliger Index generiert, an dem der neue Eintrag einsortiert wird.

5.2 Abarbeitungsreihenfolgen

Die Implementierung der Abarbeitungsreihenfolgen ist ähnlich der des Caches. Auch hier werden alle benötigten Funktionen durch ein Interface definiert und später durch die verschiedenen Abarbeitungsreihenfolgen implementiert. Die wichtigsten Funktionen des Interfaces *Traversal.java* sind in Listing 9 beschrieben.

```
public interface Traversal {
        public int getNext();
        public void enqueue(Node node);
        public void setStartNode(int id);
        ...
}
```

Listing 9: Interface Traversal.java

Die wichtigste Funktion ist hierbei *getNext()*, die immer die ID des Knotens zurück gibt, welcher als nächstes berechnet werden soll. Weitere Funktionen sind *enqueue()* zum Einfügen der Nachbarn eines Knotens in die Warteschlange und *setStartNode()*. Mit dieser wird der Startknoten zu Beginn in die Warteschlange eingefügt. Die Erstellung der Abarbeitungsreihenfolge erfolgt dabei sozusagen on-the-fly. Das heisst, die Reihenfolge in der die Knoten berechnet werden, wird erst während der eigentlichen Berechnung erstellt. Wird aktuell ein Knoten berechnet, werden dessen Nachbarn mit der Funktion *enqueue()* in die Warteschlange eingefügt.

5.2.1 BFS

Die Klasse *BFSTraversal.java* implementiert das Interface *Traversal.java*. Die Struktur dieser Klasse besteht hauptsächlich aus der eigentlichen Warteschlange, die als *queue (java.util.Interface.Queue)* definiert ist und einer weiteren Liste *neighbors (java.util.LinkedList)*. Das Interface *queue (java.util.Interface.Queue)* ist hierbei als *java.util.LinkedList* implementiert. Warum bei der Implementierung 2 Listen verwendet werden, wird sehr gut an der Funktion *getNext()* in Listing 10 deutlich.

```
public int getNext() {
        if(queue.isEmpty()) {
                if(!neighbors.isEmpty()) {
                        if(order)
                                Collections.sort(neighbors, new SortByDegree(degrees));
                        else
                                Collections.shuffle(neighbors, rndm);

                        queue.addAll(neighbors);
                        neighbors.clear();
                }
                else
                        return -1;
        }
        return queue.poll();
}
```

Listing 10: Funktion getNext() der BFS Implementierung

Zunächst wird mit der Funktion *setStartNode()* der zufällig generierte Startknoten in die *queue* eingefügt. *queue* ist die Warteschlange, die bestimmt, welcher Knoten als nächstes berechnet wird. Bei der Berechnung eines Knotens werden dessen unbearbeiteten Nachbarn zunächst mit der Funktion *enqueue()* (Listing 11) in *neighbors* eingefügt. Dazu wird jeder Nachbar vorher überprüft, ob dieser Knoten noch nicht bearbeitet wurde und nur dann einsortiert. Diese Funktion ist bei der Implementierung aller Abarbeitungsreihenfolgen identisch.

```
public void enqueue(Node node) {
        for(int idOut : node.getOut())
                if(!nodesDone.contains(idOut)) {
                        neighbors.add(idOut);
                        nodesDone.add(idOut);
                }

        for(int id : node.getIn())
                if(!nodesDone.contains(id)) {
                        neighbors.add(id);
                        nodesDone.add(id);
                }
}
```

Listing 11: Funktion enqueue() der BFS Implementierung

Wurde eine Ebene des Graphen bearbeitet, sprich *queue* ist leer, wird überprüft, ob *neighbors* Knoten enthält. Ist dies der Fall, wird überprüft, ob die Order-Strategie verwendet wird. Wird diese verwendet, werden die Knoten in der Liste *neighbors* anhand deren Grades sortiert. Wird die Order-Strategie nicht verwendet, werden die Knoten der Liste *neighbors* zufällig anhand des Random-Seed angeordnet. Als Random-Seed wird hierbei einfach die zufällige Start-ID benutzt. Der Random-Seed wird hierbei benutzt, um für einen Testlauf für alle Kombinationen aus Abarbeitungsreihenfolge und

Caching-Strategie die selbe zufällige Sortierung zu bekommen, um so einen besserern Vergleich der Kombinationen zu erhalten. Anschließend werden alle Knoten der *neighbors* Liste in die *queue* geschrieben und die Liste *neighbors* geleert.
Ist *queue* nicht leer, wird entsprechend der erste Knoten der Lsite entnommen und zurück gegeben.
Sind beide Listen leer, wurden alle Knoten bearbeitet und es wird mit *-1* ein Kill-Signal zurück gegeben.
Zwei getrennte Listen sind hierbei nötig, da jeweils für einen Ebene des Graphen alle Knoten sortiert oder zufällig angeordnet werden müssen.

5.2.2 DFS

Die Klasse *DFSTraversal.java* ist in der Struktur der BFS-Implementierung sehr ähnlich. Der einzige Unterschied in der Struktur ist, dass das Interface *queue* hier als *Stack (java.util.Stack)* implementiert ist. Der Stack arbeitet wie beschrieben nach dem Last-In-First-Out-Prinzip (LIFO), bei welchem der Knoten entnommen wird, der als letztes in den Stack gelegt wurde. Der Unterschied zu BFS wird in Listing 12 für die *getNext()* Funktion deutlich.

```
public int getNext() {
        if(!neighbors.isEmpty()) {
                if(order)
                        Collections.sort(neighbors, new SortByDegree(degrees));
                else
                        Collections.shuffle(neighbors, rndm);

                while(neighbors.size() > 0)
                        queue.push(neighbors.removeLast());
        }
        else if(queue.empty())
                return -1;

        return queue.pop();
}
```

Listing 12: Funktion getNext() der DFS Implementierung

Auch hier werden mit der Funktion *enqueue()* während der Berechnung eines Knotens alle dessen unbearbeiteten Nachbarn in *neighbors* eingefügt. Anders als bei BFS wird hier zunächst *neighbors* überprüft, ob diese Liste nicht leer ist. Ist dies der Fall, wurde demnach gerade ein Knoten bearbeitet, der noch unbearbeitete Nachbarn besitzt. Nach Definition wird nun als nächstes einer dieser Nachbarn bearbeitet. Deshalb die Überprüfung von *neighbors*. Als nächstes erfolgt die Überprüfung, ob die Order-Strategie verwendet wird und *neighbors* wird dementsprechend sortiert.
Nach dem Sortieren wird *neighbors* von hinten nach vorne in die *queue* geschrieben. Dadurch wird gewährleistet, falls die Order-Strategie benutzt wird, der Knoten mit dem größten Grad als letztes in die *queue* geschrieben wird und somit als erstes bearbeitet wird. Ist die *queue* leer, wurden alle Knoten des Graphen bearbeitet.

5.2.3 BFS-Bubble

Auch die Implementierung von *BFSBubbleTraversal.java* ist im Grunde ähnlich der BFS-Implementierung. Ein Unterschied ist eine zusätzlicher Stack, indem jeweils die Knoten gespeichert werden, deren Nachbarn als nächstes bearbeitet werden. Der Hauptunterschied wird hier wieder durch die *getNext()* Funktion in Listing 13 deutlich. Wie beschrieben, ist BFS-Bubble ein Mix aus BFS und DFS. So auch in der Implementierung. Wie beschrieben, werden in *stack* die Knoten gespeichert, deren Nachbarn berechnet werden sollen. In *queue* werden wieder entsprechend die Reihenfolge der Knoten gelistet, die berechnet werden. Die genaue Funktionsweise lässt sich hier sehr gut an einem Beispiel erklären.
Beim Start der Berechnung befindet sich jeweils in *queue* und *stack* die ID des Startknotens. Dieser Knoten wird nun bearbeitet und dessen Nachbarn in *neighbors* eingetragen. Die Besonderheit hierbei ist aber, dass nur die Nachbarn derer Knoten in *neighbors* eingetragen werden, wo der Knoten an der obersten Stelle von *stack* steht. Nach dieser Berechnung ist *queue* leer und es wird *stack* auf Einträge überprüft. Ist *stack* nicht leer, wird *neighbors* überprüft. In unserem Beispiel ist *neighbors* nicht leer und deswegen wird der Startknoten aus dem *stack* entfernt, *neighbors* je nach Order-Strategie sortiert und in die *queue* eingefügt. Der Startknoten kann aus dem *stack* entfernt werden, da dessen Nachbarn nun in der *queue* gelistet sind und somit sicher berechnet werden. Zusätzlich werden die Knoten aus *neighbors* wie bei DFS von hinten nach vorne in den *stack* geschrieben. Somit wird der Knoten an die oberste Stelle des *stack* gestellt, dessen Nachbarn als nächstes berechnet werden sollen. Dies wird nun so lange fort geführt, bis *stack* und *queue* leer sind und somit alle Knoten des Graphen bearbeitet sind.

```
public int getNext() {
        while(queue.isEmpty() ) {
                if(!stack.empty()) {
                        if(neighbors.isEmpty())
                                enqueue(cache.getNodeWithoutCounting(stack.pop()));
                        else
                                stack.pop();

                        if(order)
                                Collections.sort(neighbors, new SortByDegree(degrees));
                        else
                                Collections.shuffle(neighbors, rndm);

                        queue.addAll(neighbors);

                        while(neighbors.size() > 0)
                                stack.push(neighbors.removeLast());
                }
                else
                        return -1;
        }
        return queue.poll();
}
```

Listing 13: Funktion getNext() der BFS-Bubble Implementierung

Ist *neighbors* bei der Überprüfung leer, weil beispielsweise der aktuell berechnete Knoten keine unbearbeitete Nachbarn bestitz, wird der Knoten, der an oberster Stelle in *stack* steht, mit der Funktion *getNodeWithoutCounting()* aus dem Cache geladen und dessen Nacbarn in *neighbors* eingetragen. Um die Messergebnisse nicht zu verfälschen, werden bei der Funktion *getNodeWithoutCounting()* keine Statistiken wie die Hit-Rate notiert.

5.2.4 Sorted-by-Degree

Die Klasse *Sorted.java* implementiert ebenfalls das Interface. Die Struktur dieser Klasse besteht allerdings nur aus einer einfachen *queue*, die als *java.util.LinkedList* implementiert ist. In diese werden die Knoten absteigend nach deren Grad einsortiert. Mit der Funktion *getNext()* wird entsprechend jeweils das erste Element zurück gegeben. Hierbei spielt die Funktion *setStartNode()* keine Rolle, da die Reihenfolge der Knoten immer gleich ist und sich der Startknoten aus der sortierten Reihenfolge ergibt.

5.2.5 Random

Die Implementierung von *Random.java* ist hierbei analog zu der *Sorted.java*. Einziger Unterschied ist hierbei, dass die Knoten nicht sortiert, sondern in zufälliger Reihenfolge anhand des Random-Seed in die *queue* geschrieben werden.

5.3 Berechnung des Clustering-Koeffizienten

Die Klasse *ClusteringCoefficient.java* ist bei der Implementierung der Hauptteil, in dem sich ebenfalls die *main* Methode befindet.
Ein Großteil dieser Klasse beinhaltet hierbei den Parser, der die übergebenen Parameter für die Abarbeitungsreihenfolgen und Caching-Strategien bearbeitet. Auf diesen Teil werden wir hier nicht näher eingehen. Der Quellcode hierzu befindet sich wie beschrieben auf der beiligenden DVD.
Der wichtigste Teil dieser Klasse ist sicherlich die Berechnung des Clustering-Koeffizienten und der Transitivität. In Abbildung 8 wird die schematische Berechnung anhand eines UML-Diagrammes verdeutlicht. Der Algorithmus zu der Berechnung ist in Listing 14 zu sehen. Die Funktion bekommt als Übergabe die ID des Knotens, der berechnet werden soll. Zu Beginn wurde natürlich mit dem Aufruf der Funktion *setStartNode()* der übergebene Startknoten einsortiert. Dieser Knoten wird nun aus dem Cache geladen. Es wird anschließend abgefragt, ob die Nachbarn dieses Knotens in die

```
private static void countingTriangleForNode(int nodeID) {
        double edgeCount = 0;
        Node node = cache.getNode(nodeID);

        if(traversal.enqueueNeighbors(nodeID))
                traversal.enqueue(node);

        LinkedList<Integer> neighbors = new LinkedList<Integer>();

        if(node.getOutDegree() < node.getInDegree()) {
                for(int id : node.getOut()) {
                        if(node.getIn().contains(id))
                                neighbors.add(id);
                }

        }
        else {
                for(int id : node.getIn())
                        if(node.getOut().contains(id))
                                neighbors.add(id);
        }

        // if neighbors < 2 the Local Cluster Coefficient must be 0
        if(neighbors.size() < 2) {
                return;
        }

        double size = neighbors.size() * (neighbors.size() - 1);
        possibleTriangles += size;

        Node neighbor1;

        while(neighbors.size() > 0) {
                neighbor1 = cache.getNode(neighbors.removeFirst());
                ListIterator<Integer> outer_iter = neighbors.listIterator();

                while(outer_iter.hasNext()) {
                        int neighbor2 = (Integer) outer_iter.next();

                        if(neighbor1.getOut().contains(neighbor2))
                                edgeCount++;
                        if(neighbor1.getIn().contains(neighbor2))
                                edgeCount++;
                }
        }

        neighbors = null;
        triangles += edgeCount;
        sumLocalCC += edgeCount / size;
}
```

Listing 14: Funktion zur Berechnung des Clustering-Koeffizienten

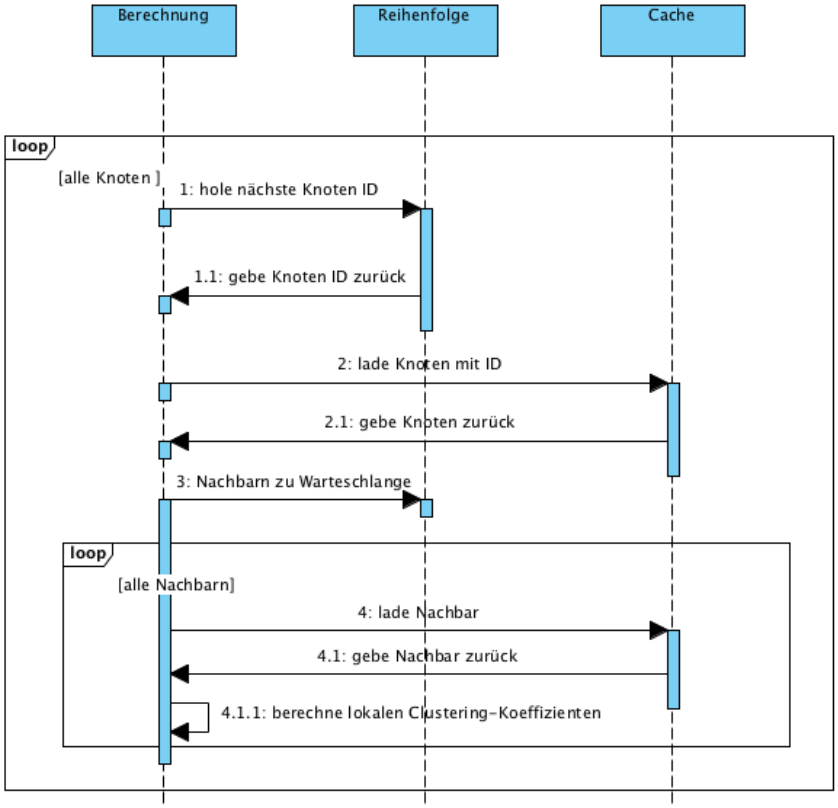

Abbildung 8: Schematischer Ablauf der Clustering-Koeffizient-Berechnung

Warteschlange eingereiht werden sollen. Diese Abfrage ist deswegen, da wie beschrieben bei der Implementierung von BFS-Bubble jeweils nur die Nachbarn des Knotens eingereiht werden, der an oberster Stelle des Stack steht. Liefert diese Abfrage entsprechend *true*, werden die Nachbarn einsortiert.
Als nächster Schritt werden in *neighbors* all die Knoten aufgenommen, die wie per Definition festgelegt, eine bidirektionale Verbindung zu dem Knoten besitzen. Dazu werden die Nachbarn überprüft, ob diese jeweils in der *in* und in der *out* Liste des Knotens vorkommen. Um Laufzeit zu sparen, wird hier die kleinere der beiden Listen durchlaufen.
Sollte die neu erstellte Liste *neighbors* eine Länge kleiner 2 Einträge besitzen, kann die Berechnung abgebrochen werden, da dieser Knoten einen lokalen Clustering-Koeffizient von 0 besitzt.
In *size* wird, falls der Knoten mehr als 2 Nachbarn besitzt, nach Definition die maximale Anzahl an Dreicken bzw. Kanten zwischen dessen Nachbarn ermittelt. In *possibleTriangles* wird für die Berechnung der Transitivität diese Größe addiert.
Um die Anzahl der Kanten bzw. Dreiecke zu bestimmen, werden nun wie durch die Iteration über die Knoten des Graphen beschrieben, alle Nachbarn des Knotens durchlaufen und überprüft, ob diese eine Verbindung untereinander besitzen. Wie beschrieben, ist durch die Implementierung der *in* und *out* Listen als *HashSet* hierbei eine konstante Abfrage von $O(1)$ möglich. Da es sich bei dem Graphen um einen gerichteten handelt, muss hierbei für die Nachbarn jeweils auf die Existenz von 2 Kanten überprüft werden.
Abschließend werden die ermittelten Werte für die Anzahl der Dreiecke und der lokale Clustering-Koeffizient in *triangles* und *sumLocalCC* addiert. Der Aufruf dieser Funktion ist in Listing 15 zu sehen.

```
while(nextID != -1) {
        numberOfNodes++;
        countingTriangleForNode(nextID);
        nextID = traversal.getNext();
}

transitivity = triangles / possibleTriangles;
avgCC = sumLocalCC / numberOfNodes;
```

Listing 15: Funktionsaufruf der Funktion countingTriangleForNode()

Es wird hierbei eine Schleife entsprechend solange durchlaufen, bis der Teil für die Abarbeitungsreihenfolge -1 zurück gibt, was signalisiert, dass alle Knoten berechnet wurden. Anschließend werden die finalen Werte für den durchschnittlichen Clustering-Koeffizienten und die Transitivität bestimmt und in Dateien geschrieben, die später ausgewertet werden.

6 Evaluation

In diesem Kapitel beschreiben wir die Ergebnisse, die bei den Testläufen der 40 möglichen Kombinationen aus Abarbeitungsreihenfolge und Caching-Strategie ermittelt werden konnten. Dazu beschreiben wir zunächst, unter welchen Vorraussetzungen die Test durchgeführt wurden. Anschließend bewerten wir die gesammelten Ergebnisse.

6.1 Evaluationssetup

Nachfolgend beschreiben wir unter welchen Vorraussetzungen die Testläufe durchgeführt wurden. Dazu geben wir zunächst einen Überblick über die Testdaten und die verwendete Testumgebung. Anschließend folgt die Erklärung der verschiedenen Testläufe.

6.1.1 Testdaten

Die Daten bzw. die Graphen, auf denen die Tests durchgeführt wurden, entstammen dem sozialen Netzwerk Google+. Durch crawlen des Netzwerkes wurden über einen langen Zeitraum verschiedene Datensätze erstellt. Dazu wurden jeweils für einen Tag von einem zufälligen Benutzer eine BFS-Traversierung gestartet, die besuchten Knoten notiert und somit pro Tag ein Datensatz erstellt. Diese Datensätze wurden bereits vor dem Beginn dieser Arbeit gesammelt und werden hier ausschließlich für die Tests unseres Ansatzes verwendet.
Zu diesem Zweck wurden 5 Datensätze aus den gesammelten Datensätzen ausgewählt. Die Datensätze wurden dabei so gewählt, dass diese verschiedene Größen im Hinblick auf Knoten- und Kantenanzahl aufweisen.
Die Datenstruktur besteht demnach aus einer Datei pro Knoten, in der die Informationen für den Knoten gespeichert sind. Eine Beispieldatei ist in Abbildung 9 dargestellt. Die erste Nummer dieser Datei ist die Google-ID, also der eindeutige Identifier, den Google pro Benutzer vergibt. Der Timestamp ist die Zeitangabe, wann dieser Knoten gecrawlt wurde. Der Dateiname sieht für diese Datei wie folgt aus: 11347957-103476608540533144781. Dabei ist die erste Zahl des Dateinamens eine Task-ID, die durch den Crawler vergeben wird und die zweite entsprechend die Google-ID.

```
# User:
103476608540533144781;;;""
# Timestamp:
1320940146644
# Out:
49
# In:
33
# Out list:
108672553383490362497;;;"Max Mustermann"
106546620669151845945;;;"Erika Musterfrau"
…
# In list:
108640203338803576083;;;"Franz-Xaver Gabler"
116359522951147071421;;;"Max MacMusterman"
…
```

Abbildung 9: Beispielhafte Datei eines Knotens

Die Kanten werden hierbei in eingehende und ausgehende Kanten unterteilt. Demzufolge ist es ein gerichteter Graph.
Um den Speicherverbrauch für die Datensätze zu verringern und um die Daten besser verarbeiten zu können, wurden die Originaldaten in ein neues Datenformat transformiert. Dazu wurden die Google-IDs durch neue kürzere IDs ersetzt und die Namen, die für die späteren Tests nicht von Bedeutung sind, entfernt. Ebenfalls wurden die Originaldatei in 2 Dateien aufgeteilt. Eine Datei für die eingehenden und eine Datei für die ausgehenden Kanten. Da die Anzahl der Kanten somit direkt aus der Länge der Dateien abgelesen werden kann, konnten die Einträge für die Anzahl der Kanten ebenfalls entfernt werden. In Abbildung 10 und 11 sind beispielhafte Dateien für das neue Datenformat zu sehen. Die Dateien beinhalten dementsprechend nur noch die neuen IDs derer Knoten, die eine Verbindung zu diesem Knoten besitzen.

```
12
19
64
1337
52
…
```

Abbildung 10: Beispielhafte In-Datei eines Knotens

```
8
12
19
128
52
…
```

Abbildung 11: Beispielhafte Out-Datei eines Knotens

Die Dateiname dieser Dateien sehen wie folgt aus: 1_103476608540533144781_1320940146644_IN bzw.

1_103476608540533144781_1320940146644_OUT. Dabei ist die erste Zahl die neue interne ID und die zweite die eigentliche Google-ID. Die dritte Zahl ist der Timestamp und abschließend der Zusatz *IN* bzw. *OUT* für die Bestimmung der eingehenden bzw. ausgehenden Kanten.
Die Datensätze umfassen allerdings nicht den kompletten Graphen von Google+, sondern nur einen Teilgraphen. Deswegen kann es vorkommen, dass Kanten zu einem Knoten zeigen, der in diesem Datensatz nicht enthalten ist. Ebenfalls kann es durch Inkonsistenzen, zum Beispiel bei den Sichtbarkeitseinstellungen der Nutzer, dazu kommen, dass beispielsweise Knoten *A* eine ausgehende Verbindung zu Knoten *B* besitzt, Knoten *B* allerdings keine eingehende Verbindung zu Knoten *A*. Um diese Probleme zu umgehen, wurden die Datensätze bereinigt, indem alle Verbindungen auf ebendiese Probleme untersucht und gegebenenfalls korrigiert wurden. Der Quellcode für diese Implementierung findet sich ebenfalls auf der beiliegenden DVD.
Nach dieser Bereinigung besitzen die 5 Datensätze folgende Größen in Bezug auf Knoten- und Kantenanzahl:

Datensatz	Knoten	Kanten	Erstellungsdatum
1	165.801	19.363.678	30. Juli 2011
2	402.702	33.259.850	1. November 2011
3	403.474	62.140.128	17. Dezember 2011
4	321.797	78.598.862	30. Januar 2012
5	285.941	94.461.312	20. Februar 2012

Tabelle 1: Auflistung der 5 Datensätze

Anhand der Tabelle 1 fällt auf, dass beispielsweise der 5. Datensatz mit 285.941 Knoten nur die zweitmeisten Knoten besitzt, dafür allerdings mit 94.461.312 die höchste Anzahl an Kanten. Die Erklärung hierfür ist, dass das Netzwerk während den verschiedenen Erstellungsdaten gewachsen ist und so die Benutzer untereinander besser verbunden sind.

6.1.2 Testumgebung

Die Testläufe wurden auf 5 identischen Maschinen durchgeführt. Bei den Maschinen handelt es sich um HP ProLiant MicroServer mit jeweils 5 GB DDR3 SDRAM PC3-10600 Arbeitsspeicher. Prozessor ist hierbei ein Athlon2 1,3GHZ DualCore und eine Festplatte mit 250 GB Serial ATA-300 mit 7200 rpm.
Als Betriebssystem kam Debian 6 Squeeze zum Einsatz. Zusätzlich OpenJDK Runtime Environment mit der Java Version 1.6.0_18.

6.1.3 Testläufe

Ein Testlauf besteht daraus, alle 40 möglichen Kombinationen aus der Abarbeitungsreihenfolge und den Caching-Strategien zu testen. Die 40 Kombinationen bestehen dabei aus den beschriebenen Abarbeitungsreihenfolgen: BFS, DFS, BFS-Bubble, Sorted-by-Degree und Random sowie aus den Caching-Strategien: LRU, FIFO, LFU, Weighted und Random. Dabei kann bei BFS, DFS und BFS-Bubble zusätzlich die Order-Strategie hinzugefügt werden. Eine Kombination wäre beispielsweise als Abarbeitungsreihenfolge BFS mit einem LRU-Cache. Es ergeben sich somit insgesamt 40 verschiedene Kombinationsmöglichkeiten.
Bei den ersten Testläufen der verschiedenen Kombinationen zeigte sich bei allen 5 Datensätzen das gleiche Verhalten. Wir haben uns deshalb entschieden, die weiteren Tests nur auf dem kleinsten Datensatz Nummer 1 fort zu führen. Dadurch war es uns möglich, für alle der 40 verschiedenen Kombinationen 25 Wiederholungen durchzuführen und somit Schwankungen während der Berechnung so gut wie möglich auszuschließen.
Die nachfolgenden Ergebnisse sind deshalb exemplarisch für die 5 Datensätze aus dem arithmetischen Mittel der 25 Wiederholungen je Kombination auf dem 1. Datensatz. Als Cachegrößen wurden bei den Tests Werte von: 128MB, 256MB, 512MB, 768MB und 1024MB gewählt.

6.2 Ergebnisse

Wie beschrieben, wurden die Testläufe auf Daten des sozialen Netwerkes Google+ durchgeführt. Die nachfolgenden Ergebnisse sind deswegen nur repräsentativ für diese Art von Graphen aus sozialen Netzwerken.
Auch bei den verschiedenen Cachegrößen zeigte sich wie bei den verschiedenen Datensätzen das gleiche Verhalten. Wir erläutern deswegen exemplarisch anhand einer Cachegröße von 512MB die Ergbnisse unserer Tests.

6.2.1 Hit-Raten

In diesem Abschnitt werden wir die gemessenen Hit-Raten der verschiedenen Kombinationen aus der Reihenfolge, wie die Knoten berechnet werden, und der Caching-Strategie erläutern. In Tabelle 2 sind die Hit-Raten für die 40 Kombinationen aufgelistet. Der Zusatz Order bei BFS, DFS, und BFS-Bubble beschreibt hier, ob die Order-Strategie verwendet wurde.

	BFS Order	BFS	DFS Order	DFS	BFS-Bubble Order	BFS-Bubble	Sorted-by-Degree	Random
LRU-Cache	60,01 %	43,33 %	68,51 %	73,87 %	69,63 %	75,28 %	53,37 %	30,88 %
FIFO-Cache	56,86 %	42,09 %	67,02 %	73,12 %	68,18 %	74,53 %	50,48 %	30,91 %
LFU-Cache	22,08 %	26,37 %	31,17 %	36,10 %	26,65 %	31,01 %	20,20 %	30,98 %
Weighted-Cache	19,17 %	21,54 %	23,33 %	24,73 %	21,21 %	22,80 %	18,83 %	22,93 %
Random-Cache	56,13 %	41,77 %	66,53 %	72,38 %	67,65 %	73,81 %	50,26 %	30,94 %

Tabelle 2: Hit-Raten mit Cachegröße 512MB

Hit-Rate wird stark von der Abarbeitungsreihenfolge beeinflusst

Aus den Ergebnissen unserer Evaluation wird ersichtich, dass die Wahl der Caching-Strategie und der Abarbeitungsreihenfolge einen starken Einfluss auf die Hit-Raten hat.
Anhand des trivialsten Ansatzes, der zufälligen Abarbeitungsreihenfolge, lässt sich hierbei erkennen, dass die Hit-Raten für alle Caching-Strategien, außer der Weighted-Strategie, durchweg fast die gleichen Werte aufweisen. Benutzt man stattdessen eine andere Abarbeitungsreihenfolge, verändern sich die Hit-Raten deutlich. Daraus zeigt sich, dass die Abarbeitungsreihenfolge einen starken Einfluss auf die Hit-Raten hat.

LRU und FIFO erzielen die besten Hit-Raten

In Bezug auf die Hit-Raten ist LRU die überwiegend geignetste Caching-Strategie. Mit dieser Caching-Strategie werden bei allen Kombinationen, außer der zufälligen Abarbeitungsreihenfolge, die besten Hit-Raten erzielt. Beispielsweise konnte mit der BFS-Bubble-Abarbeitungsreihenfolge und der LRU-Strategie mit 75,28 % der insgesamt höchste Wert von allen Kombinationen erreicht werden. Dieses Ergebnis deckt sich hierbei auch mit unserer Annahme, dass LRU die wohl beste Strategie ist. Bei der LRU-Strategie werden wie beschrieben die Einträge nach der Zeit sortiert, wann sie das letzte mal benutzt wurden. Es zeigt sich, dass diese Art der Cache-Internen Sortierung für unsere Berechnungen am besten geeignet ist.
Mit der FIFO-Strategie werden ebenfalls durchweg gute Ergebnisse in Bezug auf die Hit-Raten erzielt. Die Prozentwerte liegen bei dieser Strategie nur wenige Punkte hinter denen der LRU-Strategie. So konnte beispielsweise ebenfalls für die BFS-Bubble-Abarbeitungsreihenfolge mit 74,53 % der beste Wert für die FIFO-Strategie erreicht werden. Die Differenz zur LRU-Strategie beträgt dabei lediglich 0,75 %. Diese Ergebnisse der FIFO-Strategie decken sich allerdings nicht komplett mit unserer Annahme. Zwar sind wie erwartet die Hit-Raten im Vergleich zu denen der LRU-Strategie geringer, jedoch nicht so stark wie wir erwartet haben. Eine mögliche Erklärung könnten hierbei die Abarbeitungsreihenfolgen sein. Da die Knoten in einer bestimmten Reihenfolge bearbeitet werden, ist es möglich, dass Knoten die zu Beginn benutzt werden, bei der Berechnung der Knoten, die weiter hinten in der Reihenfolge sind, nicht mehr betrachtet werden müssen. Dies würde erklären, warum die Hit-Raten bei der FIFO-Strategie ebenfalls sehr hoch sind.

LFU und Weighted leiden an dem Blockierungsproblem

Für die LFU- und Weighted-Strategie ergeben sich im Vergleich zu LRU und FIFO deutlich geringere Hit-Raten. So konnten für die sich abzeichnende beste Abarbeitungsreihenfolge BFS-Bubble lediglich 31,01 % bzw. 22,80 % erreicht werden. Die niedrigen Hit-Raten für die LFU-Strategie lassen sich durch das Blockierungsproblem erklären.
Das Blockierungsproblem tritt bei der LFU-Strategie und dementsprechend bei der Weighted-Strategie, die im Grunde auf der LFU-Strategie aufbaut, auf. Wie in Kapitel 4 beschrieben, kann es dazu kommen, dass Einträge, die nicht mehr verwendet werden, trotzdem in dem Cache aufgrund eines hohen Hit-Countes verbleiben. Somit blockieren diese Einträge Speicherplatz und neue Einträge können nicht in den Cache geschrieben werden.
Bei dem Blockierungsproblem hat die Reihenfolge, in der die Knoten bearbeitet werden, wie bei der FIFO-Strategie, einen großen Einfluss. Das Blockierungsproblem wird durch die guten Hit-Raten der FIFO-Strategie und der dementsprechenden Vermutung, dass Knoten größtenteils nur zu einer bestimmten Zeit verwendet werden, unterstützt. Bestimmte Knoten werden demzufolge nur in einer gewissen Zeit häufig benutzt und besitzen deswegen einen hohen Hit-Count. Werden diese Knoten nicht mehr gebraucht, verbleiben diese allerdings solange in dem Cache, bis deren Hit-Count durch die periodische Reduzierung so weit verringert wurde, bis diese entfernt werden können. Während dieser Zeit blockieren diese Knoten somit andere Knoten, die zur jetzigen Zeit benötigt werden. Dieses Problem zieht sich durch die gesamte Berechnung, was die niedrigeren Hit-Raten erklärt.
Bei der Weighted-Strategie kommt hinzu, dass hier die Knoten mit großem Grad bevorzugt werden. Durch den erhöhten

Speicherverbrauch dieser Knoten können insgesamt weniger Knoten in dem Cache zwischengespeichert werden. Dies erklärt die niedrigeren Hit-Raten im Vergleich zur LFU-Strategie.

Nicht erklärbare Ergebnisse der Random-Caching-Strategie

Eine Besonderheit bei den Caching-Strategien stellt die Random-Strategie dar. Bei dieser Strategie werden ebenfalls gute Hit-Raten erzielt. Mit dieser Strategie konnte beispielsweise für BFS-Bubble ein Wert von 73,81 % erreicht werden und liegt dementsprechend lediglich 1,47 % hinter der LRU-Strategie. Bis dato konnten wir allerdings nicht genau klären, weshalb für diese Strategie so gute Werte erziehlt werden. Eine mögliche Erklärung könnte hierbei der Aufbau des Graphen selbst sein.

BFS-Bubble und DFS sind bezüglich der Hit-Raten die besten Abarbeitungsreihenfolgen

Für die Abarbeitungsreihenfolgen ergibt sich aus Tabelle 2, dass mit unserem eigenen Ansatz, der BFS-Bubble-Abarbeitungsreihenfolge, die besten Werte erziehlt werden konnten. So konnte für die Caching-Strategien: LRU, FIFO und Random jeweils mit 75,28 %, 74,53 % und 73,81 % die besten Werte erzielt werden. Lediglich für die LFU- und Weighted-Strategie konnten mit dieser Reihenfolge nicht die höchsten Werte erreicht werden. Es zeigt also, dass unsere Idee für diese Art von Graphen gut geeignet ist und sich somit wie erwartet gute Hit-Raten erzielen lassen. Der BFS-Bubble-Ansatz besteht dabei aus einem Mix aus BFS und DFS.
Mit der DFS-Abarbeitungsreihenfolge werden ebenfalls gute Hit-Raten erziehlt. Die Ergebnisse für DFS liegen nur sehr gering unter denen der BFS-Bubble-Abarbeitungsreihenfolge. So konnte beispielsweise für DFS und LRU eine Hit-Rate von 73,87 % erreicht werden. Dieser Wert liegt dabei nur 1,41 % hinter dem Wert der besten Kombination BFS-Bubble und LRU.

Nicht erklärbare Ergebnisse mit der Order-Strategie bei BFS-Bubble und DFS

Eine Besonderheit bei den beiden Abarbeitungsreihenfolgen: BFS-Bubble und DFS sind die unterschiedlichen Werte mit und ohne Order-Strategie.
Die Order-Strategie ist eine Erweiterung, die die interne Bearbeitungsreihenfolge von BFS, DFS und BFS-Bubble so ändert, dass die Knoten nach deren Grad absteigend bearbeitet werden. Die Idee hinter dieser Strategie ist, dass durch das bevorzugte Bearbeiten der großen Knoten dementsprechend viele Nachbarn geladen werden müssen. Somit müssten diese Knoten später nicht erneut geladen werden.
Entgegen unserer Annahme verringern sich die Hit-Raten bei dem Einsatz der Order-Strategie. So konnte mit der Order-Strategie für BFS-Bubble und LRU 69,63 % und für DFS mit LRU 68,51 % erreicht werden. Im Schnitt konnte mit der Order-Strategie für BFS-Bubble und DFS 4,8 % weniger Hits erreicht werden.
Warum sich dieses Verhalten bei diesen Abarbeitungsreihenfolgen im Gegensatz zu der BFS-Abarbeitungsreihenfolge, wo durch den Einsatz der Order-Strategie die Ergbnisse verbessert werden, zeigt, können wir nicht komplett erklären. Dieses Phänomen könnte, wie bei der Random-Caching-Strategie, mit dem Aufbau des Graphen zu tun haben.

Erwartete Ergebnisse bei BFS und Sorted-by-Degree

Die Abarbeitungsreihenfolgen: BFS und Sorted-by-Degree liefern hingegen die erwarteten Ergebnisse. Bei der BFS-Abarbeitungsreihenfolge zeigt sich wie erwartet eine deutliche Verbesserung mit dem Einsatz der Order-Strategie. Ohne Order-Strategie konnte für LRU ein Wert von 43,33 % und mit Order-Strategie 60,01 % erreicht werden. Im Durchschnitt konnten mit der Order-Strategie 7,8 % mehr Hits erzielt werden. Im Vergleich zu BFS-Bubble sind die Hit-Raten allerdings deutlich niedriger. Das lässt die Vermutung nahe, dass wie beschrieben, das Problem der großen Ebenen einen Einfluss auf die Hit-Raten hat.
Die Order-Strategie zeigt bei dieser Abarbeitungsreihenfolge allerdings wie erwartet eine deutliche Steigerung der Hit-Raten.

BFS-Bubble und DFS bereits bei kleiner Cachegröße mit sehr guten Hit-Raten

Die Entwicklungen der Hit-Raten anhand der Größe des Caches sind in den Abbildungen 12 und 13 exemplarisch für die LRU- und Weighted-Strategie zu sehen. Mit der FIFO- und Random-Strategie ergibt sich hierbei ein ähnliches Bild wie in Abbildung 12 und für die LFU-Strategie ein ähnliches Bild wie in Abbildung 13. In Abbildung 12 zeigt sich, dass BFS-Bubble und DFS bereits bei kleiner Cachegröße sehr gute Hit-Raten aufweisen, wohingegen für die anderen Abarbeitungsreihenfolgen die Hit-Raten stärker durch die Erhöhung der Cachegröße steigen. In Abbildung 13 dagegen steigen die Hit-Raten für alle Abarbeitungsreihenfolgen etwa gleich stark an.

Mit BFS-Bubble und LRU-Cache werden die besten Hit-Raten erzielt

Zusammenfassend lässt sich also sagen, dass die Kombination aus LRU-Strategie mit BFS-Bubble ohne Order-Strategie in Bezug auf die Hit-Raten, die effektivste Variante ist. Mit dieser Kombination wurde der Maximalwert von 75,28 % erreicht.

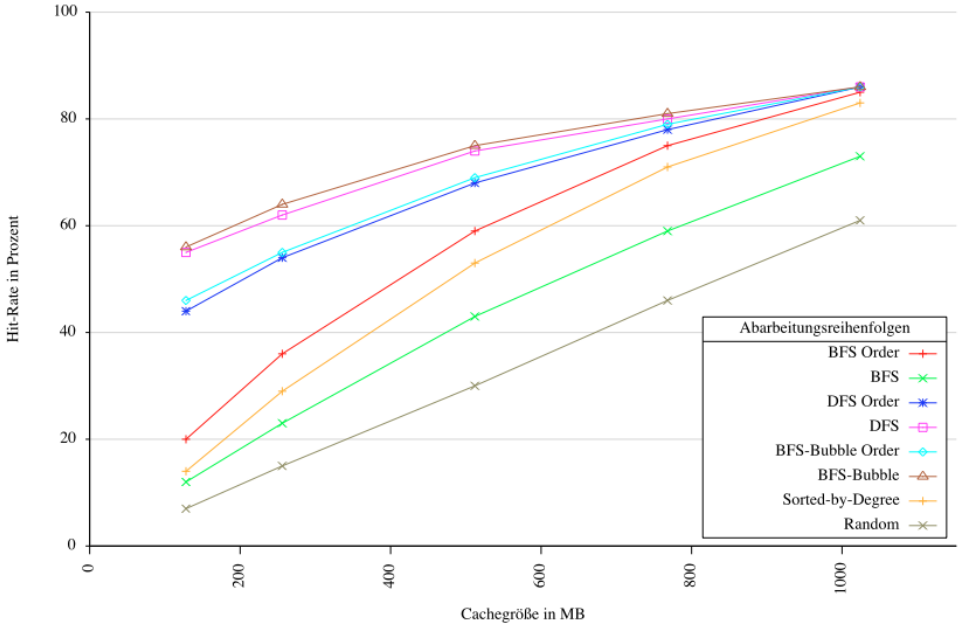

Abbildung 12: Verhältnis Cachegröße zu Hit-Rate anhand der LRU-Strategie

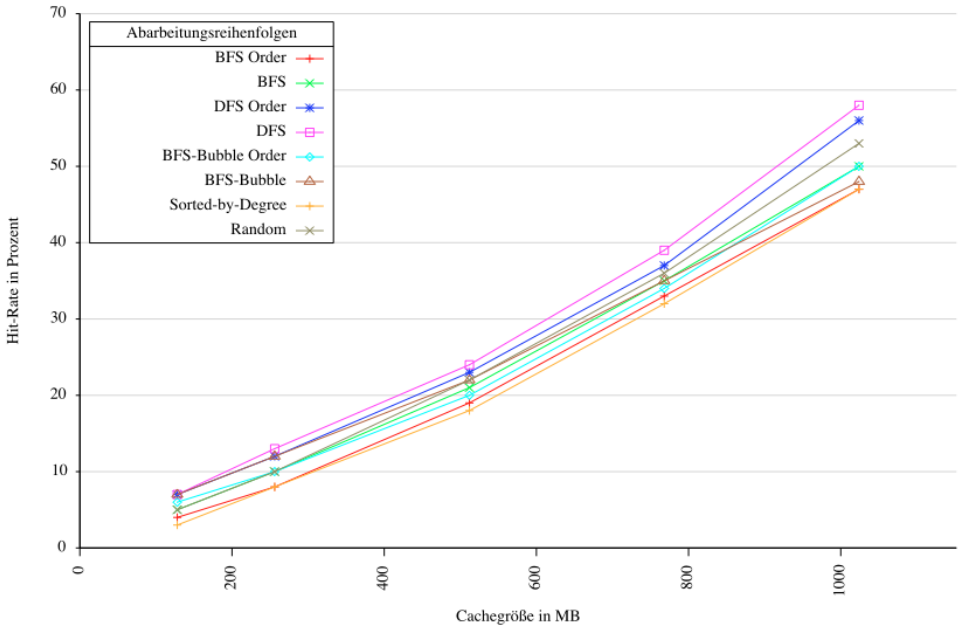

Abbildung 13: Verhältnis Cachegröße zu Hit-Rate anhand der Weighted-Strategie

6.2.2 Laufzeiten

In diesem Abschnitt werden wir die ermittelten Laufzeiten zu den Kombinationen beschreiben. In Tabelle 3 sind die jeweiligen Laufzeiten der verschiedenen Kombinationen aufgelistet. Auch hier beschreibt der Zusatz Order bei BFS, DFS, und BFS-Bubble, ob die Order-Strategie verwendet wurde.

	BFS Order	BFS	DFS Order	DFS	BFS-Bubble Order	BFS-Bubble	Sorted-by-Degree	Random
LRU-Cache	812,64 s	1060,28 s	634,00 s	504,72 s	626,36 s	493,12 s	868,84 s	1262,92 s
FIFO-Cache	915,52 s	1192,68 s	780,60 s	597,12 s	755,16 s	571,76 s	1025,08 s	1369,72 s
LFU-Cache	704,08 s	672,68 s	752,00 s	734,08 s	688,56 s	671,80 s	702,16 s	621,04 s
Weighted-Cache	657,08 s	642,08 s	607,44 s	614,24 s	648,40 s	656,84 s	649,32 s	593,80 s
Random-Cache	3920,52 s	4200,32 s	2791,08 s	2327,44 s	2729,84 s	2336,00 s	4434,68 s	4169,76 s

Tabelle 3: Laufzeiten mit Cachegröße 512MB

Sehr hohe Laufzeiten der Random-Caching-Strategie durch Implementierung bedingt

Was bei dieser Auflistung sofort auffällt, sind die sehr hohen Laufzeiten der Random-Caching-Strategie. Obwohl die Hit-Raten bei der Berechnung mit der Random-Strategie gut sind, verhalten sich die Laufzeiten hierbei nicht wie erwartet. So beträgt die Laufzeit für BFS-Bubble mit Random 2336,00 Sekunden bei einer Hit-Rate von 73,81 %. Als Vergleich hierzu die Laufzeit, ebenfalls mit BFS-Bubble und FIFO-Caching-Strategie, wo nur eine leicht höhere Hit-Rate von 74,53 % erreicht wurde und eine Laufzeit von 571,76 Sekunden gemessen werden konnte.
Die hohen Laufzeiten lassen sich durch die Implementierung der Random-Strategie erklären. Da kein zufälliger Zugriff auf einzelne Elemente der Liste möglich ist, wird deswegen für jeden neuen Eintrag ein Index zufällig bestimmt, an dem das neue Element einsortiert wird. Um dieses Element dort einzufügen, muss dazu allerdings immer die Liste bis zu dem erstellten Index durchlaufen werden, was die hohe Laufzeit dieser Strategie erklärt.

Niedrige Laufzeiten für LFU und Weighted trotz niedriger Hit-Raten

Weitere Verhalten, die bei den Laufzeiten auffallen, sind die doch recht niedrigen Laufzeiten für die LFU- und Weighted-Strategie. Obwohl die Hit-Raten teilweise deutlich hinter denen mit LRU- oder FIFO-Strategie liegen, sind die Laufzeiten teilweise geringer. Zum Beispiel konnten bei der BFS-Abarbeitungsreihenfolge mit LRU-Strategie 60,01 % Hits erreicht und eine Laufzeit von 812,64 Sekunden gemessen werden. Als Vergleich hierzu mit der LFU- und Weighted-Strategie. Mit diesen Strategien konnten 22,08 % und 19,17 % Hit-Rate erreicht und eine Laufzeit von 704,08 und 657,08 Sekunden gemessen werden. Hier stehen demnach die Hit-Raten nicht in Abhängigkeit zu den Laufzeiten.

Großteil der benötigten Daten werden aus dem Cache gelesen

Um das Phänomen der niedrigen Laufzeiten trotz niedrigen Hit-Raten zu erklären, ist in Abbildung 14 die Entwicklung der gelesenen Datenmengen beispielsweise während der Berechnung mittels BFS mit Order-Strategie für die LRU- und Weighted-Strategie zu sehen. Hit-Größe bezeichnet hierbei die gelesene Datenmenge aus dem Cache und Miss-Größe die Datenmenge, die von der Festplatte gelesen wurde. Aus Abbildung 14 zeigt sich, dass bei der LRU-Strategie deutlich mehr Daten aus dem Cache gelesen werden anstatt von der Festplatte. Auch bei der Weighted-Strategie wird ein Großteil der Daten aus dem Cache gelesen. Allerdings ist hier das Verhältnis zwischen Hit- und Miss-Größe weniger stark als bei der LRU-Strategie. Als Vergleich ist in Abbildung 15 die gleiche Entwicklung für die schnellste Variante, BFS-Bubble ohne Order-Strategie mit LRU-Cache, zu sehen.
Es wird also deutlich, dass die Hit-Raten und daraus resultierenden Datenmengen, die aus dem Cache gelesen werden, bei den Caching-Strategien LRU und FIFO einen wie erwarteten Einfluss auf die Laufzeiten haben. Denn je weniger Daten von der Festplatte gelesen werden, desto schneller sollte es sein. Aus Abbildung 14 wird allerdings nicht ersichtlich, warum beispielsweise die Weighted-Strategie trotz der deutlich niedrigeren Hit-Raten im Vergleich zur LRU-Strategie eine relativ große Datenmenge aus dem Cache lädt. Ebenfalls wird aus dieser Abbildung nicht klar, weshalb beispielsweise die Berechnung mittels BFS mit Order-Strategie und LRU-Cache langsamer ist als mit einem Weighted-Cache, obwohl hier mehr Daten von der Festplatte gelesen werden, und somit langsamer sein müsste.

Weighted und LFU lesen überwiegend kleine Knoten von der Festplatte

In Abbildung 16 ist für die gleiche Konfiguration wie in Abbildung 14 die Verteilung der gelesenen Knoten von der Festplatte zu sehen. Die X-Achse bezeichnet hierbei die Größe der Knoten in Byte und die Y-Achse die Anzahl der Knoten mit einer bestimmten Größe. Aus der Größe der Knoten lässt sich demnach der Grad der Knoten folgern.
Aus dieser Abbildung wird sehr gut unsere Annahme zu der Weighted-Strategie deutlich. Es zeigt sich, dass bei der Weighted-Strategie (grün) der Großteil der von der Festplatte gelesenen Knoten relativ kleine Knoten sind und dementsprechend die Knoten mit großem Grad im Cache gehalten werden. Im Gegensatz zur LRU-Strategie, wo die Verteilung

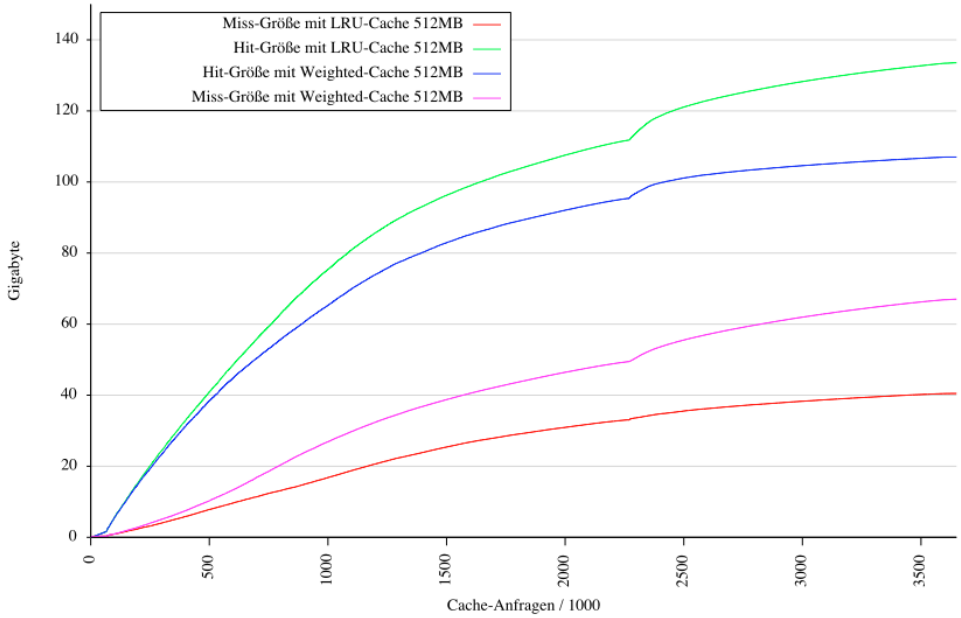

Abbildung 14: Entwicklung der gelesenen Datenmengen anhand BFS mit Order-Strategie

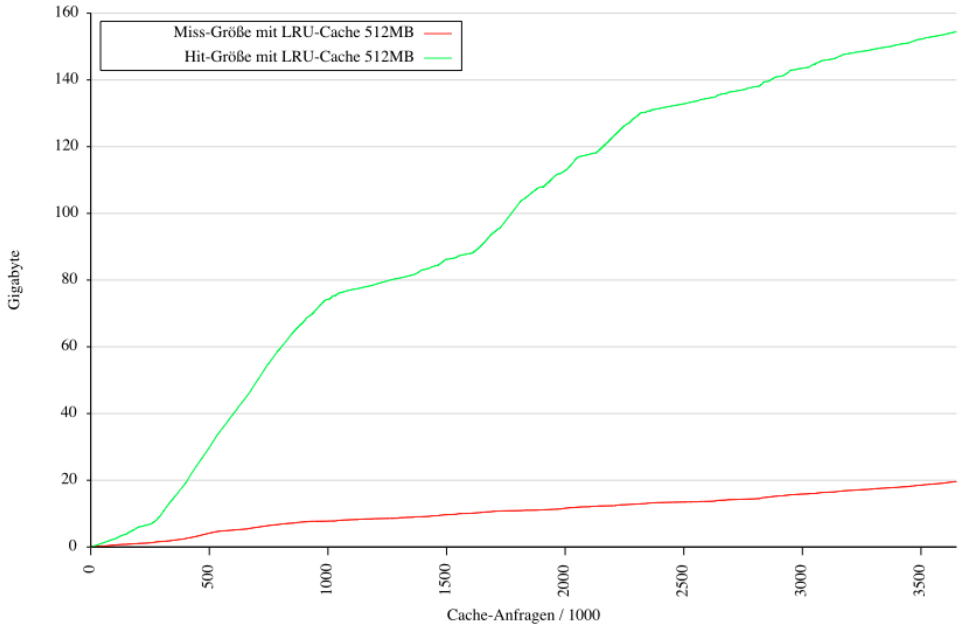

Abbildung 15: Entwicklung der gelesenen Datenmengen anhand BFS-Bubble ohne Order-Strategie

der Knoten wesentlich ausgeglichener ist. Ein ähnliches Bild ergibt sich hierbei ebenfalls für die anderen Caching-Strategien. Aus Abbildung 14 und Abbildung 16 wird deutlich, dass bei der Weighted- und LFU-Strategie, die ein ähnliches Verhalten zeigt, mehr Daten von der Festplatte gelesen werden, aber diese Datenmenge hauptsächlich durch das Laden kleinerer Knoten verursacht wird.

Disk-Cache könnte das Caching beeinflussen

Trotz dieser Erkenntnis ist es für uns nicht komplett nachvollziehbar, weshalb das Laden vieler kleiner Dateien und dazu die größeren Datenmengen schneller ist, als das Laden geringerer Datenmengen dafür aber mit größeren Dateien. Aus unserem Verständnis, müsste das Laden beispielsweise einer großen Datei schneller sein als das Laden von mehreren kleinen Dateien. Eine plausible Lösung hierfür könnte der in Kapitel 2 beschriebene Disk-Cache sein. Wie beschrieben, wird dieser vom Betriebssystem und der Festplatte selbst benutzt, um Daten zwischenzuspeichern. Es könnte demnach sein,

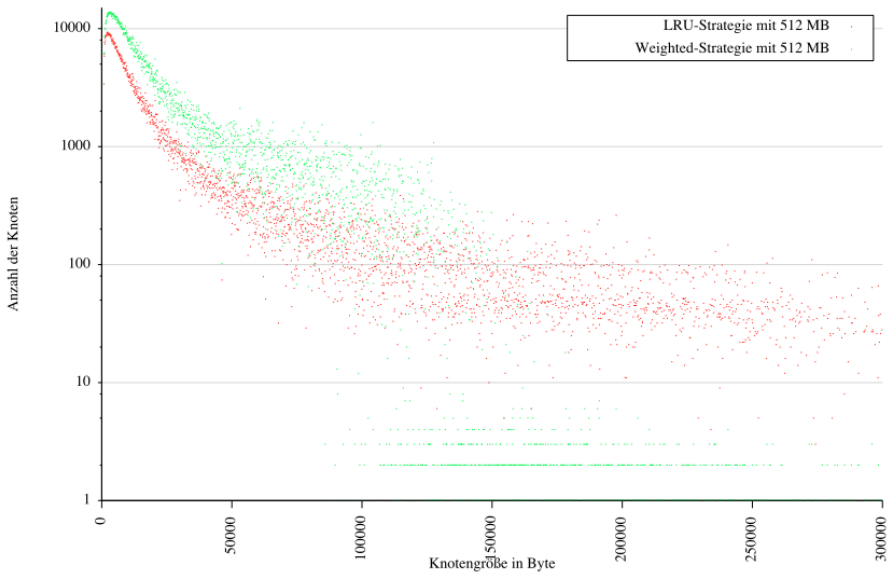

Abbildung 16: Verteilung der gelesenen Knoten von der Festplatte mit BFS Order

dass in diesem Disk-Cache während der Berechnung Dateien zwischengespeichert werden, die häufig von der Festplatte geladen werden. Demnach würden bei der Berechnung mit der Weighted- oder LFU-Strategie hauptsächlich kleinere Dateien in dem Disk-Cache zwischengespeichert. Das hätte natürlich zur Folge, dass so mehr Dateien dort gespeichert werden als beispielsweise bei der Berechnung mit der LRU-Strategie, da hier die Verteilung der gelesenen Dateien sehr viel ausgeglichener ist. Somit müssten bei der Berechnung mit Weighted- oder LFU-Strategie zwar theoretisch mehr Daten von der Festplatte gelesen werden, allerdings kann hier deswegen vermehrt auf den Disk-Cache zurückgegriffen werden. Dieses Verhalten hätte zwar ebenfalls Einfluss auf die Berechnungen mit den restlichen Caching-Strategien, aber nicht so einen starken wie bei der Weighted- oder LFU-Strategie.

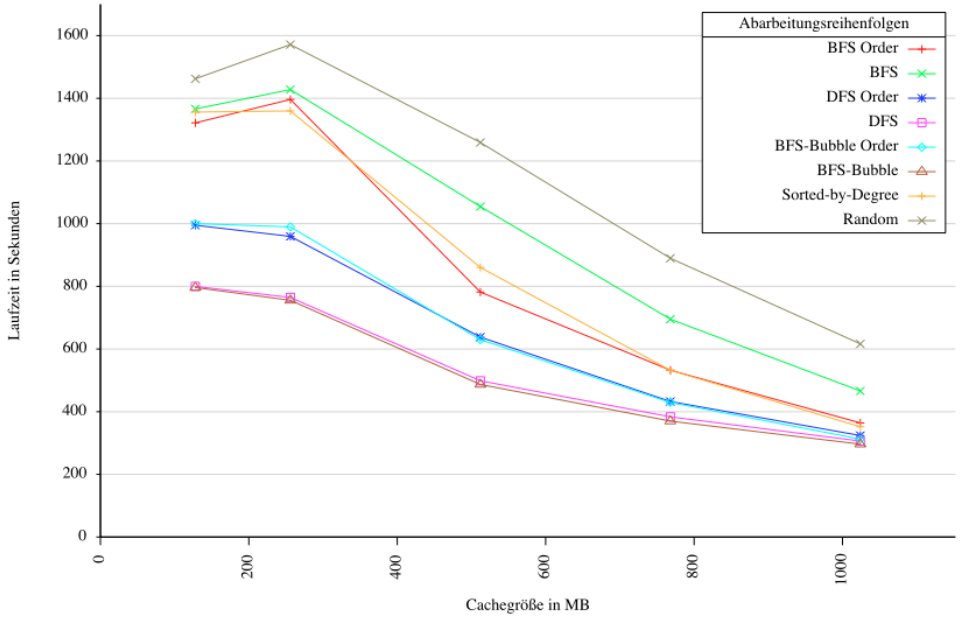

Abbildung 17: Verhältnis Laufzeit zu Cachegröße anhand der LRU-Strategie

Erhöhte Laufzeit bei kleinen Cachegrößen für LRU und FIFO

Zur Übersicht sind in Abbildungen 17 und 18 für die LRU- und Weighted-Strategie die Entwicklung der Laufzeiten für die verschiedenen Abarbeitungsreihenfolgen für die verschiedenen Cachegrößen dargestellt.

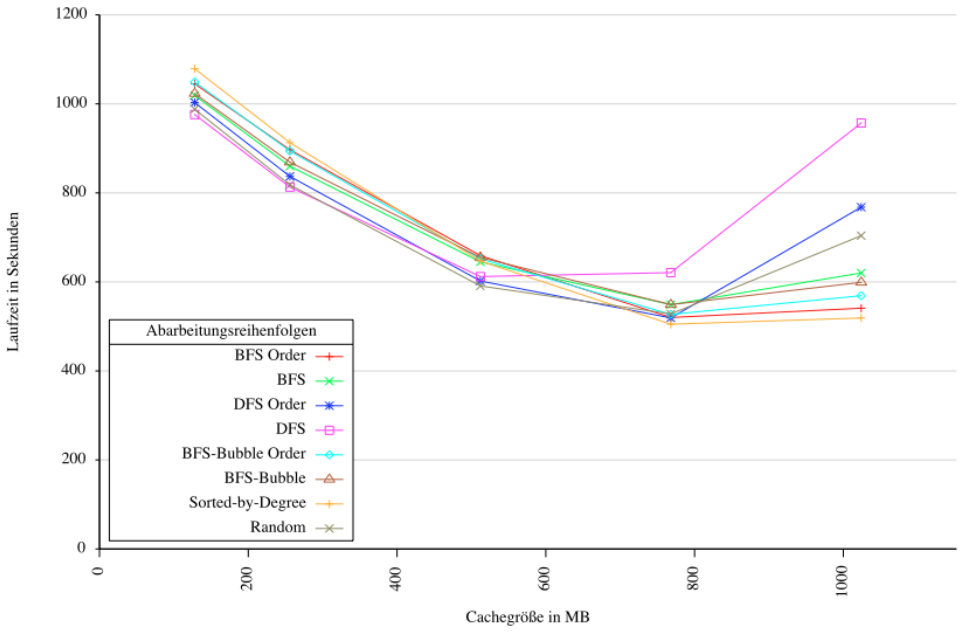

Abbildung 18: Verhältnis Laufzeit zu Cachegröße anhand der Weighted-Strategie

Die FIFO-Strategie zeigt hierbei das gleiche Verhalten wie die LRU-Strategie. Es fällt auf, dass von 128MB Cachegröße zu 256MB keine bzw. bei einigen Verfahren sogar eine Steigerung der Laufzeit beobachtet werden kann. Um mögliche Messfehler und Schwankungen bei der Berechnung auszuschließen, haben wir zusätzlich 20 Testläufe mit einer Cachegröße von 64MB und 192MB durchgeführt. Die Ergebnisse dieser Berechnungen sind wieder exemplarisch anhand der LRU-Strategie in Abbildung 19 dargestellt. Auch die zusätzlichen Messungen zu der FIFO-Strategie zeigen das gleiche Verhalten. Die zusätzlichen Messungen bestätigen demnach die Laufzeiten aus Abbildung 17. Die Theorie, die wir zu die-

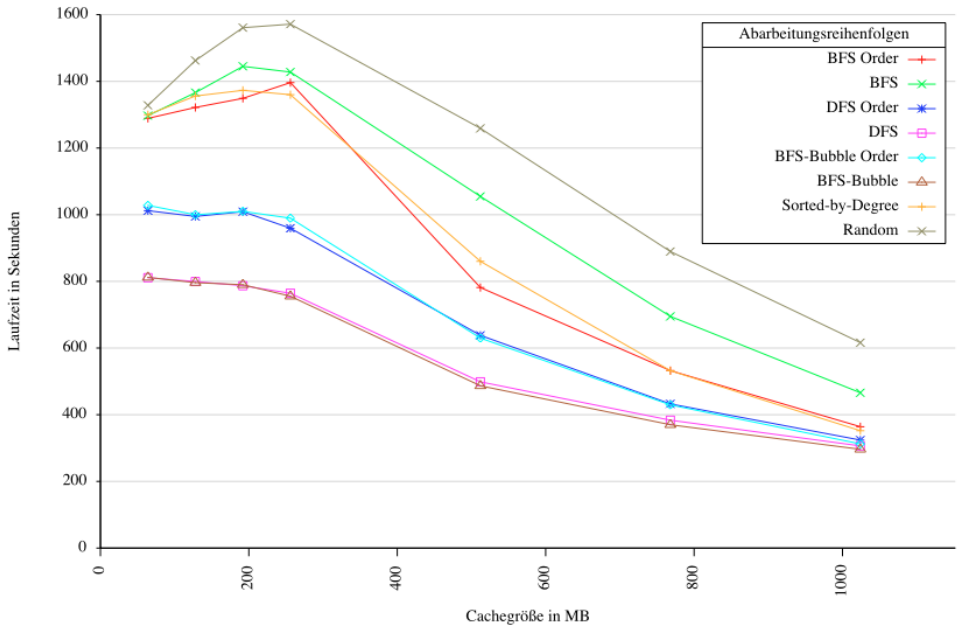

Abbildung 19: Verhältnis Laufzeit zu Cachegröße anhand der LRU-Strategie

sen ungewöhnlichen Lauzeiten haben, umfasst wieder den Disk-Cache. Wir vermuten, dass bei einer kleinen Cachegröße der Disk-Cache effektiver arbeitet als unser eigener Cache und somit die Laufzeiten beeinflusst werden.

Weighted und LFU für große Cachegrößen nicht geeignet

Ein ähnliches Verhalten wie bei der LRU- und FIFO-Strategie zeigt sich in Abbildung 18 für die Weighted- und LFU-Strategie. Hier steigt die Laufzeit für fast alle Abarbeitungsreihenfolgen ab einer Cachegröße von ca. 780MB an. Um dieses Verhalten ebenfalls durch Messfehler oder Schwankungen bei der Berechnung ausschließen zu können, haben wir hierzu 20 Testläufe mit einer Cachegröße von 1280MB durchgeführt. Die Ergebnisse zu diesen Testläufen sind in Abbildung 20 zu sehen. Auch hier werden die Ergebnisse bestätigt.
DFS zeigt hierbei ein stärkeres Verhalten als die anderen Verfahren. So steigt die Kurve bei DFS bereits früher als bei den anderen Abarbeitungsreihenfolgen. Auch hier könnte der Aufbau des Graphen, wie beispielsweise bei der Order-Strategie, eine wichtige Rolle spielen.
Die Erklärung, warum die Laufzeiten bei den Kombinationen mit Weighted- und LFU-Strategie überhaupt steigen, ist der Mehraufwand, der bei der internen Sortierung des Caches auftritt. Je größer die Cachegröße wird, desto mehr Knoten können in dem Cache zwischengespeichert werden. Gerade bei der Weighted- und LFU-Strategie bedeutet bei mehr Einträgen mehr Verwaltungsaufwand für das Sortieren der Liste und das periodische Reduzieren der nicht verwendeten Einträge. Dadurch ist ab einer bestimmten Cachegröße der Verwaltungs-Overhead größer als der eigentliche Nutzen des Caches.

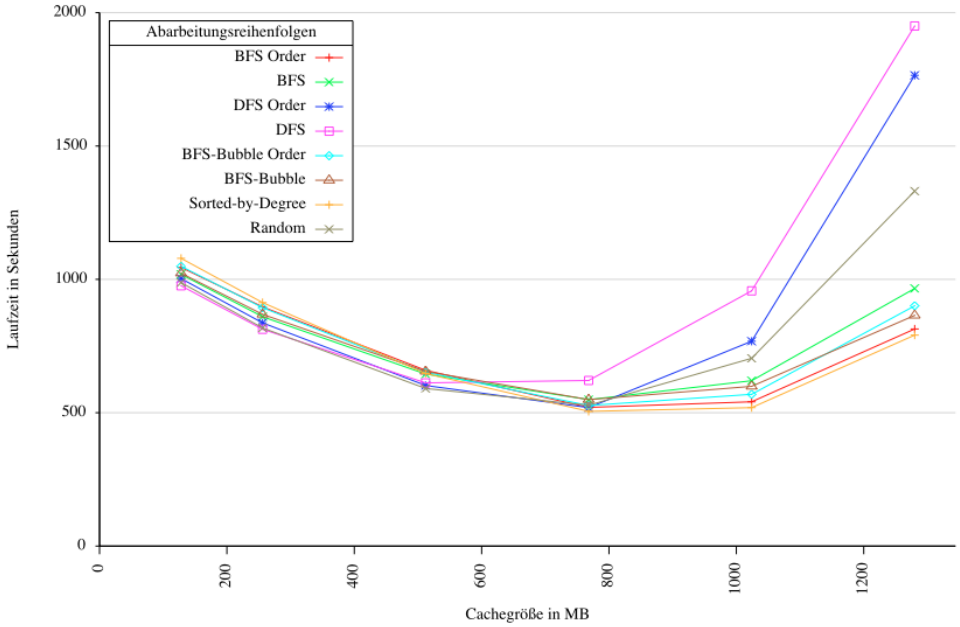

Abbildung 20: Verhältnis Laufzeit zu Cachegröße anhand der Weighted-Strategie

BFS-Bubble mit LRU-Cache ist die schnellste Kombination

Auch bei den Laufzeiten zeigt sich, dass BFS-Bubble ohne Order-Strategie und LRU die effektivste Variante ist. Mit 493,12 Sekunden konnte für diese Kombination die insgesamt schnellste Laufzeit gemessen werden. Auch für die FIFO-Strategie konnte mit BFS-Bubble ohne Order-Strategie die beste Laufzeit von 571,76 Sekunden erreicht werden.

6.3 Ergebnisse des Clustering-Koeffizienten

Das primäre Ziel dieser Arbeit ist es, unseren Ansatz zum Berechnen des Clustering-Koeffizienten zu evaluieren. In diesem Abschnitt geben wir deshalb nur einen kurzen Überblick über die ermittelten Werte für den Clustering-Koeffizienten und einen Vergleich mit anderen Ergebnissen.

Für den kleinsten Datensatz konnte bei unseren Tests ein durchschnittlicher Clustering-Koeffizient von ca. 0.309 ermittelt werden. Das bedeutet, dass in diesem Teilgraphen des Google+ Netzwerkes, im Durchschnitt ca. 30.9 % der Nachbarn eines Knotens miteinander verbunden sind.
Für die Transitivität konnte ein Wert von ca. 0.286 ermittelt werden. Die Wahrscheinlichkeit beträgt hier demnach 28.6 % in unserem Fall eines sozialen Netzwerkes, dass für einen Benutzer der Freund eines Freundes auch sein Freund ist.

In [JH98] zeigen Watts und Strogatz für das Schauspieler-Netzwerk (http://www.imdb.com) einen Clustering-Koeffizienten von 0,79. Bei diesem Graphen besteht eine Verbindung zwischen 2 Knoten / Schauspielern, wenn diese zusammen in einem Film mitgespielt haben. Das gleiche Netzwerk besitzt nach [New03] eine Transitivität von 0.2. Im gerichteten Graphen eines E-Mail-Netzwerkes konnte ein Clustering-Koeffizient von 0.168 ermittelt werden [NFB02].
In einer anderen Studie konnte für das derzeit größte soziale Netzwerk Facebook für den durchschnittlichen Clustering-Koeffizient ein Wert von 0.16 ermittelt werden [GKBM09].

6.4 Mögliche approximative Berechnung

Bei der Berechnung des Clustering-Koeffizienten ist es möglich, zu jedem Zeitpunkt den aktuellen Wert des Clustering-Koeffizienten und der Transitivität auszugeben. Die Idee ist, dass die unterschiedlichen Abarbeitungsreihenfolgen womöglich einen Einfluss auf die Werteentwicklung des Clustering-Koeffzienten und der Transitivität haben. Zu diesem Zweck haben wir während der Berechnung immer die aktuellen Werte für den durchschnittlichen Clustering-Koeffizienten und die Transitivität ermittelt.
In Abbildung 21 ist die Werteentwicklung bei der Berechnung mittels BFS ohne Order-Strategie und LRU-Cache zu sehen. Hierbei hat die verwendete Caching-Strategie keinen Einfluss auf die Werte. Diese sind zum Beispiel für BFS ohne Order-Strategie für alle verschiedenen Caching-Strategien identisch.

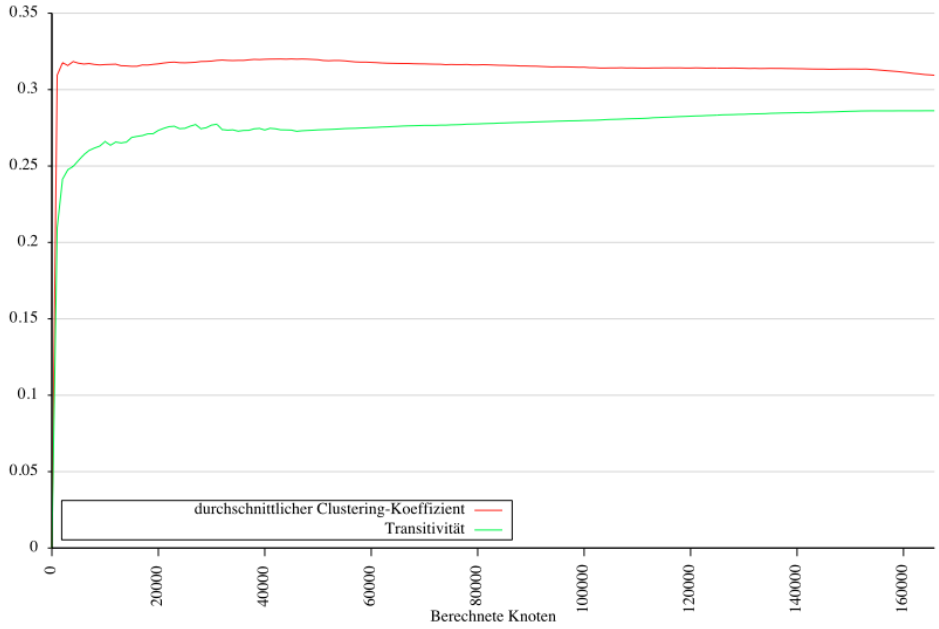

Abbildung 21: Werteentwicklung bei der Berechnung mittels BFS und LRU-Cache

Es wird deutlich, dass vorallem der durchschnittliche Clustering-Koeffizient sich sehr schnell dem endgültigen Wert annähert. Während der Berechnung sind nur noch leichte Schwankungen zu erkennen. Bei 20.000 berechneten Knoten liegt die Differenz zu dem finalen Wert beispielsweise bei lediglich 2,4 % und bei 80.000 berechneten Knoten bei 2,2 %. Es wäre also durchaus möglich, mit dieser Abarbeitungsreihenfolge die Berechnung in diesem Fall beispielsweise nach 20.000 berechneten Knoten abzubrechen und so eine gute Schätzung des durchschnittlichen Clustering-Koeffizienten zu erhalten.
Um eine Schätzung für die Transitivität zu erhalten, könnte beispielsweise Sorted-by-Degree verwendet werden, wie in Abbildung 22 zu sehen. Bei 20.000 berechneten Knoten liegt die Differenz bei 3,7 % und bei 80.000 Knoten nur noch bei 0,3 %. Hier könnte also bei ca. der Hälfte der Knoten gestoppt werden und man bekäme eine sehr gute Schätzung. Für die Schätzung des durchschnittlichen Clustering-Koeffizienten wäre diese Reihenfolge allerdings schlechter geeignet als BFS. Bei 80.000 berechneten Knoten ist die Differenz, wie auch der Abbildung entnommen werden kann, mit 10,1 % recht hoch.
Als Negativbeispiel kann hierfür die DFS-Abarbeitungsreihenfolge angenommen werden. Wie in Abbildung 23 zu sehen, sind für DFS Order die Schwankungen für die Transitivität sehr groß. Ein ähnliches Bild zeichnet hierbei ebenfalls DFS ohne die Order-Strategie.
Unser Ziel ist es zwar den exakten Wert für den Clustering-Koeffizienten zu berechnen, aber wie wir mit den verschiedenen Abarbeitungsreihenfolgen zeigen konnten, wäre es demzufolge ebenfalls möglich, durch unseren Ansatz eine Schätzung dieser Werte zu bekommen.

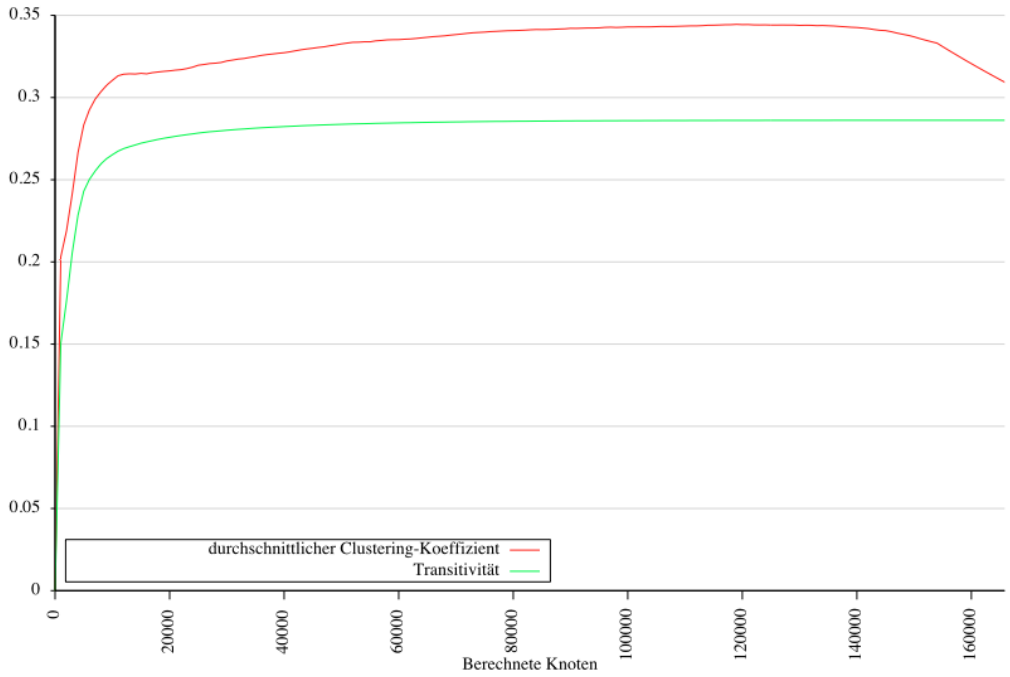

Abbildung 22: Werteentwicklung bei der Berechnung mittels Sorted-by-Degree und LRU-Cache

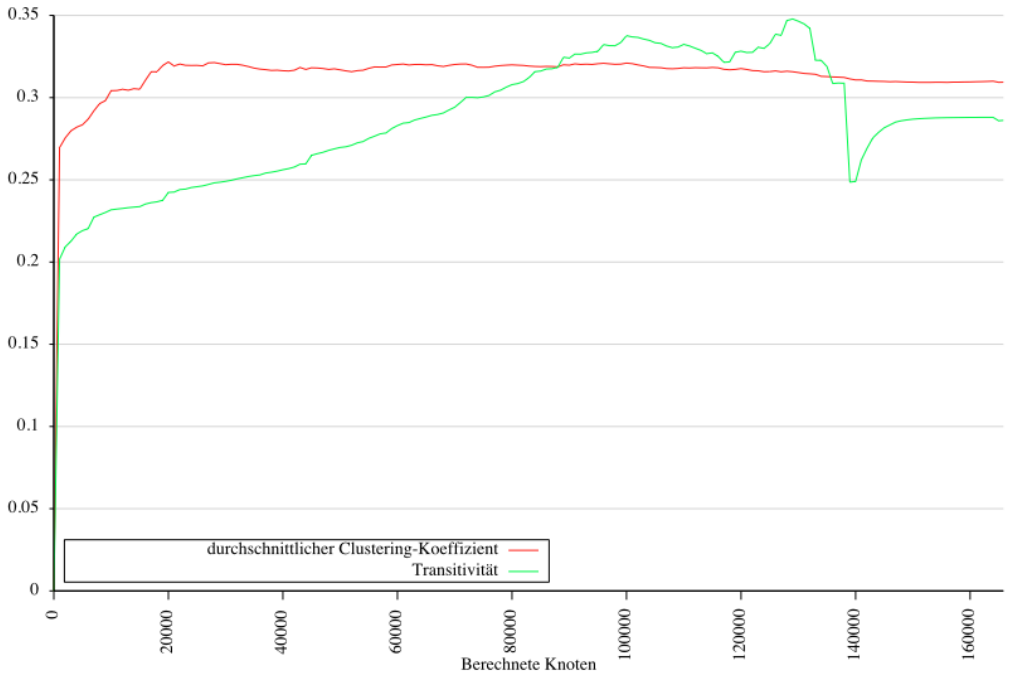

Abbildung 23: Werteentwicklung bei der Berechnung mittels DFS und LRU-Cache

7 Zusammenfassung und Ausblick

Wie wir beschrieben haben, ist die exakte Berechnung des Clustering-Koeffizienten für große Graphen sehr speicher- und rechenintensiv. Das Hauptproblem bei den beschriebenen schnellen und exakten Verfahren, wie der AYZ-Algorithmus, ist dabei der sehr große Speicherverbrauch wegen der Graphen-Darstellung als Matrix. Ebenso ist es bei diesen Verfahren für die Berechnung nötig, den gesamten Graphen im Speicher zu halten. Deswegen ist es auf Maschinen mit wenig Speicher nicht möglich, mit diesen Verfahren den exakten Wert zu berechnen.
Die Problemstellung vor dieser Arbeit war demnach, ein Verfahren zu finden, mit welchem die exakte Berechnung des Clustering-Koeffizienten für große Graphen auch auf Maschinen mit wenig Speicher möglich ist. Mit dieser Arbeit haben wir gezeigt, dass durch unseren Ansatz, bestehend aus den verschiedenen Abarbeitungsreihenfolgen und den verschiedenen Caching-Strategien, die exakte Berechnung auch auf Maschinen mit wenig Speicher durchgeführt werden kann.
Für die Berechnung des exakten Wertes haben wir die Iteration über die Knoten des Graphen verwendet. Mit diesem Verfahren ist es allerdings nötig, dass Knoten und deren Kanten sehr oft gelesen und überprüft werden. Würden diese Daten stets von der Festplatte geladen, würde die Laufzeit für die Berechnung sehr stark steigen. Durch den Einsatz eines Caches konnten wir erreichen, dass ein Großteil der benötigten Daten aus dem Cache geladen wird und nicht von der Festplatte. Dadurch reduziert sich die Laufzeit, da ein Zugriff auf den Arbeitsspeicher, in welchem der Cache die Daten zwischenspeichert, sehr viel weniger Zeit benötigt als ein Zugriff auf die Festplatte.
Als geeignetste Caching-Strategie hat sich hierbei die LRU-Strategie herausgestellt. Bei dieser Strategie werden die Einträge nach der Zeit sortiert, wann sie das letzte Mal benutzt wurden. So konnten mit dieser Caching-Strategie für alle Abarbeitungsreihenfolgen, außer der zufällige Abarbeitungsreihenfolge die besten Hit-Raten erzielt werden. Selbst bei der zufälligen Reihenfolge ist der Unterschied zu den anderen Caching-Strategien so gering, dass sich diese Differenz womöglich bei weiteren Wiederholungen aufheben würde.
Für die verschiedenen Abarbeitungsreihenfolgen hat sich unsere Idee BFS-Bubble als sehr gut herausgestellt. Die Idee dieser Reihenfolge ist ein Mix aus BFS und DFS. Dabei werden für den aktuell bearbeiteten Knoten jeweils die Nachbarknoten dessen als nächstes bearbeitet. Dadurch wird das Phänomen, dass die Nachbarn in einem sozialen Graphen enger miteinander verbunden sind als bei anderen Graphen-Arten, sehr gut ausgenutzt. So konnten beispielsweise mit dieser Abarbeitungsreihenfolge und der LRU-Caching-Strategie die insgesamt beste Hit-Rate und Laufzeit von allen Kombinationen erreicht werden.
Eine zusätzliche Beobachtung die wir während unsereren Testläufen machen konnten, dass die Abarbeitungsreihenfolge einen Einfluss auf die Werteentwicklung der Clustering-Koeffizienten hat. So konnte beispielsweise anhand der BFS-Abarbeitungsreihenfolge gezeigt werden, dass sich der Wert für den durchschnittlichen Clustering-Koeffizienten bereits früh während der Berechnung dem finalen Wert annähert. Die Idee hierbei ist, dass somit die Berechnung nach einer bestimmten Zeit gestoppt werden kann und man so eine gute Schätzung des Clustering-Koeffizienten bekommt.

Unsere Berechnungen und Tests wurden allerdings nur auf Graphen aus sozialen Netzwerken durchgeführt. Deswegen sind die ermittelten Werte auch nur für diese Art von Graphen aussagekräftig. Weitere Tests könnten beispielsweise darin bestehen, unseren Ansatz auf anderen Graphen wie dem Zufallsgraph zu testen. Hierbei könnten die Ergebnisse deutlich von den unseren abweichen.
Anhand der Tests auf einem bekannten Graphen-Modell könnten sich auch einige Besonderheiten erklären, die wir bei unseren Tests feststellen konnten. So wurden durch den Einsatz der Order-Strategie bei DFS und BFS-Bubble entgegen unserer Annahme schlechtere Hit-Raten erzielt wie ohne die Order-Strategie. Dieses Verhalten können wir anhand unserer getesteten Graphen nicht klären, da uns der Aufbau dieser Graphen nicht komplett bekannt ist. Es ist durchaus denkbar, dass sich diese Besonderheiten bei Tests auf Graphen-Modellen, bei denen die Struktur bekannt ist, erklären lassen.
Bei den Abarbeitungsreihenfolgen sind durchaus Erweiterungen und Verbesserungen möglich. Eine Idee könnte hierbei beispielsweise eine mögliche Erweiterung der BFS-Bubble-Abarbeitungsreihenfolge sein. Es wäre möglich, nicht nur die direkten Nachbarn des zuvor berechneten Knotens als Nachfolger zu wählen, sondern Nachbarn, die über eine Weglänge von 2 oder mehr erreichbar sind. Generell könnten sich aber auch weitere Abarbeitungsreihenfolgen, die wir nicht in unserer Arbeit getestet haben, als geeignet erweisen.
Verbesserungen der Caching-Strategien könnten ebenfalls die Ergebnisse positiv beeinflussen. Eine Möglichkeit der Verbesserung bieten allen voran die LFU- und Weighted-Strategie. Wie wir bei den Tests zeigen konnten, sind diese Strategien für große Cachegrößen ungeeignet, da der Verwaltungsoverhead durch das Sortieren der Einträge sehr groß wird. Hier könnte besipielsweise die Liste, in der die Einträge sortiert sind, in einzelne Blöcke unterteilt werden, um so das Sortieren einzelner Einträge gegebenenfalls auf diese Blöcke zu begrenzen. Dadurch müssten Einträge nicht durch die komplette Liste sortiert werden, was weniger Rechenzeit in Anspruch nehmen sollte.
Allerdings ist es ebenfalls möglich, dass es weitere Caching-Strategien gibt, mit denen gute Ergebnisse erreicht werden können.

Literatur

[AYZ97] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[BYKS02] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[CLRS07] Th. H. Cormen, Ch. E. Leiserson, R. Rivest, and C. Stein. *Algorithmen - Eine Einführung*. Oldenbourg Wissenschaftsverlag GmbH, 2007.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990. <ce:title>Computational algebraic complexity editorial</ce:title>.

[GKBM09] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. A walk in facebook: Uniform sampling of users in online social networks. *CoRR*, abs/0906.0060, 2009.

[JH98] WATTS D. J. and STROGATZ S. H. Collective dynamics of 'small-world' networks. *Nature (London)*, 393(6684):440–442, 1998. eng.

[KNT10] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In Philip S. Yu, Jiawei Han, and Christos Faloutsos, editors, *Link Mining: Models, Algorithms, and Applications*, pages 337–357. Springer New York, 2010.

[New03] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):pp. 167–256, 2003.

[NFB02] M. E. J. Newman, Stephanie Forrest, and Justin Balthrop. Email networks and the spread of computer viruses. *Phys. Rev. E*, 66:035101, Sep 2002.

[NP03] M. E. J. Newman and Juyong Park. Why social networks are different from other types of networks. *Phys. Rev. E*, 68:036122, Sep 2003.

[NWS02] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(Suppl 1):2566–2572, 2002.

[Smi85] Alan J. Smith. Disk cache — miss ratio analysis and design considerations. *ACM Trans. Comput. Syst.*, 3(3):161–203, 1985.

[SW04] T. Schank and D. Wagner. *Approximating clustering-coefficient and transitivity*. Universität Karlsruhe, Fakultät für Informatik, 2004.

[SW05] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In SotirisE. Nikoletseas, editor, *Experimental and Efficient Algorithms*, volume 3503 of *Lecture Notes in Computer Science*, pages 606–609. Springer Berlin Heidelberg, 2005.

[TKM09] Charalampos E. Tsourakakis, Mihail N. Kolountzakis, and Gary L. Miller. Approximate triangle counting. *CoRR*, abs/0904.3761, 2009.

[Tso08] C.E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on*, pages 608 –617, dec. 2008.

[Ull11] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Computing, 2011.