Karlsruhe Institute of Technology

Semi-externes Clustern von Graphen mit der Louvain-Methode

Bachelorarbeit
von

Oliver Plate

An der Fakultät für Informatik
Institut für Theoretische Informatik (ITI)

Erstgutachter:	Prof. Dr. Dorothea Wagner
Zweitgutachter:	Juniorprof. Dr. Henning Meyerhenke
Betreuender Mitarbeiter:	Michael Hamann, M.Sc.

Bearbeitungszeit: 16. Dezember 2015 – 13. April 2016

Selbständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Arbeit sowie der verwendete Quelltext meine eigene Arbeit sind. Arbeit anderer wurde durch Quellenverweise oder Nennung des Autors (Quelltext) eindeutig kenntlich gemacht.

Statement of Authorship

I hereby declare that this document and the used code have been composed by myself and describe my own work, unless otherwise acknowledged in the text.

Karlsruhe, 12. April 2016

Abstract

In context of this bachelor's thesis we regard the clustering of networks. The goal is to separate the nodes of a network into clusters, whereby the nodes in a cluster should have many connections with each other. Many different approaches and algorithms exist for finding these clusterings, Blondel et al.'s Louvain-Method [BGLL08] is one of them.

Most of the algorithms hold important data, for example the different nodes and edges of a graph, in the main memory during the computations. The fast growth of social networks or networks generated from the World Wide Web have resulted in increasing sizes of graphs which we want to examine. The graphs have multiple billions of nodes and edges, making it impossible for in-memory procedures to find clusterings due to the restricted amount of main memory.

In this bachelor's thesis, we analyse how far the Louvain-Method allows relocation of information from the main memory onto external memory (harddrives), so that it gets possible to cluster big graphs without extending the amount of main memory.

As part of the thesis, two algorithms have been developed, a semi-external variant and a full-external variant of the Louvain-Method. The former relocates only the information about edges, which normally make up the most memory space, onto the harddrive. Informations about the nodes are still kept in main memory. If needed, small blocks of data, containing information about the edges, are read from the harddrive into the main memory and processed afterwards. The second variant tries to exhaust the external memory concept by saving as much main memory space as reasonably possible.

After a theoretical examination of both variants, we also evaluate them with multiple experiments. We show that relocating information onto external memory is quite useful. The usage of main memory can be reduced significantly, and in case of the semi-external variant acceptable runtimes can be obtained.

Zusammenfassung

Im Kontext dieser Arbeit wird das Clustern von Netzwerken betrachtet. Ziel ist dabei, Netzwerke so in Cluster zu zerlegen, dass die Knoten innerhalb der Cluster möglichst viele Verbindungen untereinander aufweisen. Zur Findung solcher Clusterungen auf Graphen gibt es viele verschiedene Ansätze und Algorithmen, wovon einer durch die Louvain-Methode von Blondel et al. [BGLL08] beschrieben wird.

Die meisten Algorithmen behalten bei ihren Berechnungen alle relevanten Daten, wie zum Beispiel die verschiedenen Knoten und Kanten eines Graphen, im internen Speicher. Mit der rasant zunehmende Größe von sozialen Netzwerken oder Netzwerken, die aus dem World Wide Web generiert werden, müssen aber immer größere Graphen untersucht werden. Diese weisen oft mehrere Milliarden Knoten und Kanten auf, sodass den internen Verfahren durch den verfügbaren Hauptspeicher Grenzen gesetzt sind.

In dieser Arbeit wird untersucht, inwiefern die Louvain-Methode auch die Auslagerung von Daten auf externe Speicher (Festplatten) zulässt, sodass ohne Vergrößerung des Hauptspeicher auch die Clusterung großer Graphen durchgeführt werden kann.

Im Rahmen der Untersuchung wurden zwei Algorithmen entwickelt, eine semi-externe Variante und eine voll-externe Variante der Louvain-Methode. Erstere verlagert

lediglich die Speicherung der Kantendaten, also der Daten die in der Regel den meisten Speicherplatz benötigen, auf die Festplatte. Informationen zu den Knoten bleiben jedoch im Hauptspeicher. Werden einzelne Kantendaten benötigt so werden diese in Blöcken von der Festplatte gelesen und verarbeitet. Die zweite Variante versucht, das Konzept des externen Speichers maximal auszureizen und so viel wie möglich Hauptspeicher bei der Berechnung zu sparen.

Nach einer theoretischen Betrachtung beider Varianten werden diese in verschiedenen Experimenten evaluiert. Es wird gezeigt, dass die Auslagerung mancher Daten auf ein externes Medium durchaus sinnvoll ist. Der Hauptspeicherbedarf kann deutlich reduziert werden und zumindest im Fall der semi-externen Variante können mit einigen Optimierungen akzeptable Laufzeiten erreicht werden.

Inhaltsverzeichnis

1. Einleitung

Die vorliegende Arbeit befasst sich mit performanten Verfahren zur Clusterung von Graphen. Eine Clusterung dient der Findung von Gruppen innerhalb Daten mit ähnlichen Eigenschaften und/oder einer starken Verbindung untereinander. Es gibt dabei verschiedene Arten der Clusterung, solche die *überlappend* sind und einzelne Dateneinträge mehreren Clustern zuordnen können oder solche die *strikt partitioniert* sind, also bei denen jeder Dateneintrag genau einem Cluster zugeordnet wird. Auch kann zwischen *harten* oder *weichen* Clusterungen unterschieden werden, bei denen Dateneinträge auch nur teilweise zu einem Cluster gehören können (weich) oder nur als Ganzes (hart).

Die Arbeit fokussiert sich auf strikt partitionierte und harte Clusterungen, bei der für Graphen eine vollständige Unterteilung der Gesamtmenge der Knoten in disjunkte Teilmengen (Cluster) erfolgt. Interessant für die vorliegende Arbeit sind diejenigen Clusterungen, bei denen die Knoten innerhalb eines Clusters viele gemeinsame Kanten aufweisen während zwischen den Knoten unterschiedlicher Cluster nur wenige Kanten existieren.

Einige Verfahren optimieren gemäß eines Qualitätsmaßes, das bedeutet sie suchen die Clusterung bei denen das Maß den besten Wert erreicht. Zu diesen Verfahren gehört auch die *Louvain-Methode* von Blondel et al. [BGLL08], welche den Ausgangspunkt dieser Arbeit darstellt. Im Falle dieser Methode ist das verwendete Qualitätsmaß die sogenannte *Modularity*, dessen Idee in Abschnitt 1.1.2 erläutert wird.

Die Louvain-Methode selbst wurde von Blondel et al. so implementiert, dass sie sämtliche Daten im Hauptspeicher hält. Bereits der ursprüngliche Algorithmus arbeitet sehr schnell, dennoch wurde in verschiedenen Arbeiten versucht, die vorhandene Methode weiter zu optimieren um sie noch performanter zu machen.

Staudt und Meyerhenke [SM16] haben eine parallele Variante der Methode entwickelt. Durch die Nutzung mehrere Prozessorkerne konnten hier deutlich schnellere Laufzeiten für die Methode erzielt werden.

Eine anderer Ansatz wurde durch Sortera [Sot14] veröffentlicht, bei dem die ursprüngliche Methode zu einer verteilten Methode erweitert wurde, die die Berechnungen in einer Cloud ermöglicht. Diese Variante weicht allerdings teilweise von der ursprünglichen Louvain-Methode ab. Sie arbeitet nicht immer auf aktuellen Ergebnissen einzelner Phasen und nimmt dadurch einen kleinen Verlust der Ergebnisqualität in Kauf.

Alle genannten Implementierungen ermöglichen performante Clusterungen von Graphen, bei denen alle benötigten Daten wie Knoten, Kanten und Cluster im Hauptspeicher gehalten

werden. Für sie gilt: Je größter die betrachteten Graphen sind, desto mehr Hauptspeicher wird benötigt. Zur Berechnung von *Big-Data-Graphen*, die als *Webgraphen* oder Graphen zu *sozialen Netzwerken* oftmals mehrere Milliarden Knoten und Kanten umfassen, werden deshalb Rechner benötigt, die einen sehr großen Hauptspeicher besitzen (die verteilte Implementierung ausgenommen).

Ziel dieser Arbeit ist es, ein performantes Verfahren zu beschreiben und zu implementieren, das es ermöglicht, große Graphen auch mit vergleichsweise wenig Hauptspeicher zu clustern. Der für die Tests der Implementierungen verwendete Rechner war dabei mit 32 GB Hauptspeicher ausgestattet. Die geringere Anforderung an den Hauptspeicher lässt sich natürlich nur realisieren, wenn ein Großteil der Daten auf externem Speicher verwaltet wird.

Neben der Begrenzung des Hauptspeichers ist eine weitere Rahmenbedingung für das zu implementierende Verfahren, dass die ermittelten Clusterungen keine schlechteren Qualitäten aufweisen als die Ergebnisse der ursprünglichen Louvain-Methode. Unter Einhaltung dieser Rahmenbedingungen wird für mehrere Varianten untersucht, wie die Berechnungen von Clusterungen hinsichtlich der Laufzeit optimiert werden können.

Die Arbeit beginnt mit der Erläuterung grundlegender Begriffe und befasst sich dabei auch mit der Idee der Modularity als Qualitätsmaß für Clusterungen. Nach weiterer Einführung in die Grundlagen des internen und externen Speichers wird die Louvain-Methode inhaltlich vorgestellt. Damit ist die Basis geschaffen, um die beiden für die Arbeit entworfenen Algorithmen vorzustellen, die als semi-extern und voll-extern bezeichnet werden.

Nachfolgend werden die mit den Algorithmen durchgeführten Experimente an unterschiedlichen Graphen sowie deren Ergebnisse hinsichtlich Qualität der Clusterung, Laufzeit und Hauptspeicherbedarf beschrieben und analysiert. Es wird gezeigt, dass die entwickelten Varianten bei der Berechnung von großen Graphen mit deutlich weniger Hauptspeicher auskommen als die ursprünglich interne Implementierung und dass zumindest die semi-externen Variante akzeptabler Laufzeiten aufweist.

1.1 Vorbemerkungen

Im folgenden Abschnitt wird ein kurzer Einstieg in den theoretischen Teil der Arbeit gegeben. Zudem finden sich hier die wichtigsten Definitionen die im weiteren Verlauf benötigt werden.

1.1.1 Definitionen

Sofern nicht anders angegeben sei $G = (V, E, \omega)$ ein einfacher, gewichteter und ungerichteter Graph mit einer nicht-leeren Knotenmenge V, Kantenmenge E und Gewichtsfunktion $\omega : E \to \mathbb{R}^+$. Ist kein konkretes ω angegeben so sei $\omega \equiv 1$. Das Gesamtgewicht des Graphen ist definiert als $w = \frac{1}{2}\sum_{i,j}\omega(\{i,j\})$. Dabei bezeichnet $\{i,j\}$ eine ungerichtete Kante zwischen den Knoten i und j. Des Weiteren seien $n := |V|$ und $m := |E|$ sofern nicht anderweitig definiert.

Ein *Cluster* C_i ist eine nichtleere Teilmenge der Knotenmenge V. Ein Cluster mit nur einem Knoten wird als *Singleton* bezeichnet.

Eine Clusterung eines Graphen $G = (V, E, \omega)$ beschreibt eine Menge $\mathcal{C} = \{C_1 \dots C_k\}$, von Clustern, die paarweise disjunkt sind und deren Vereinigung V ergibt, also $C_i \cap C_j = \emptyset$ für $i, j \in [1 \dots k]$ und $i \neq j$ und $\bigcup \mathcal{C} = C_1 \cup C_2 \cup \dots C_k = V$. Ein Knoten ist somit stets genau einem Cluster zugeordnet. Jedem Knoten i wird durch $c(i)$ ein Index zugewiesen, der die Nummer des Clusters ist, zu dem der Knoten gehört.

Das Volumen $vol(v)$ eines Knotens v ist definiert als die Summe aller Gewichte der zu v inzidenten Kanten, $vol(v) = \sum_{\{e \in E | v \in e\}} \omega(e)$. Das Volumen eines Clusters ist definiert durch $vol(C_i) = \sum_{v \in C_i} vol(v)$.

1.1.2 Modularity

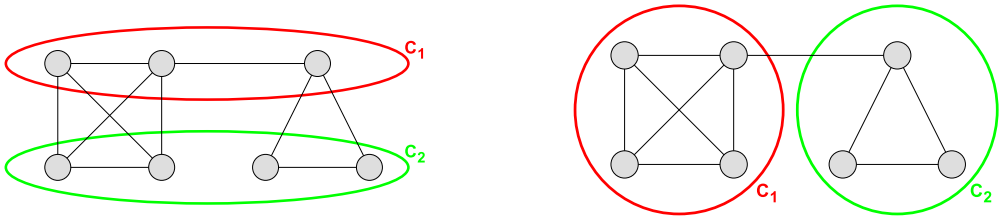
Als ein geeignetes Maß für die Qualität einer gegebenen Unterteilung eines Graphen dient die Modularity, welche 2004 von Newman und Girvan vorgestellt wurde [NG04]. Durch sie lassen sich verschiedene Clusterungen eines Graphens untereinander hinsichtlich ihrer Qualität vergleichen.

Für eine gegeben Clusterung $\mathcal{C}$ sei $A \subset E$ die Teilmenge der Kanten, deren Endknoten gemeinsam in einem Cluster liegen. Das Gesamtgewicht $vol(A)$ aller Kanten aus A sei als Abdeckung der Clusterung $\mathcal{C}$ im Graphen G bezeichnet. Gesucht wird eine Clusterung bei dem die Abdeckung $vol(A)$ einen möglichst hohen Wert in Relation zu einem Erwartungswert EA annimmt. Zur Definition von EA betrachtet man die Menge aller volumenäquivalenten Graphen zu G. Zwei Graphen G_1 und G_2 heißen dabei volumenäquivalent, wenn die Knotenmenge identisch ist und G_1 und G_2 in jedem Knoten das gleiche Volumen haben.

Für zwei Knoten i, j des Graphen G ist der Erwartungswert für das Kantengewicht $\omega_E(\{i, j\})$ gegeben durch $\frac{vol(i) \cdot vol(j)}{2w}$. Die (normierte) Summe der Differenz aus tatsächlichen Werten und erwarteten Werten ist dann:

$$Q = \frac{1}{2w} \sum_{i,j \in V} \left[\omega(\{i, j\}) - \frac{vol(i) \cdot vol(j)}{2w} \right] \delta(C_{c(i)}, C_{c(j)}) \tag{1.1}$$

Wobei $\frac{1}{2w}$ der Normierungsfaktor ist und die Dirac-Funktion $\delta()$ dafür sorgt, dass nur über Kanten in A summiert wird. Der berechnete Wert liegt in $[-\frac{1}{2}, 1)$ (siehe [Gö10]), wobei ein größerer Wert für eine bessere Clusterung auf einem Graphen steht. In Abbildung 1.1 sind zwei Beispielclusterungen eines einfachen Graphen aufgezeigt mit zugehörigen Modularity-Werten.



(A) Schlechte Clusterung (B) Gute Clusterung

Abbildung 1.1: Beispiel zwei verschiedener Clusterungen eines Graphen. (A) besitzt eine Modularity von 0,215. (B) eine Modularity von 0,505.

Die Modularity findet ihren Einsatz in der Optimierung von Clusterungen. Das exakte Optimieren der Modularität eines Graphen ist zwar $\mathcal{NP}$-schwer [BDG+08], aber Cluster-Algorithmen machen es sich zu nutze, dass sie durch die Modularity dennoch effizient Schritte berechnen können, die die Qualtität der Lösung verbessern. Von dieser Eigenschaft macht auch die Louvain-Methode [BGLL08] Gebrauch, wie später genauer erläutert wird.

1.2 Externes Speicher-Modell

In einem vereinfachten Modell kann Speicher in zwei Arten aufgeteilt werden, intern und extern. Interner Speicher, oder auch Hauptspeicher (RAM), ist schnell, flüchtig und auf zufällige Zugriffe ausgelegt. Externer Speicher hingegen ist zwar langsamer, stellt dafür allerdings weitaus mehr Speicherplatz zur Verfügung, der nicht flüchtig ist. Als externes Speichermedium werden meist *Festplatten* (HDD) oder auch sogenannte *Solid-State-Drives* (SSD) verwendet, wobei im weiteren Verlauf übersichtshalber nur der Begriff *HDD* verwendet wird. Nimmt man internen und externen Speicher zusammen, so bilden sie eine Speicherhierarchie (vgl. Abbildung 1.2).

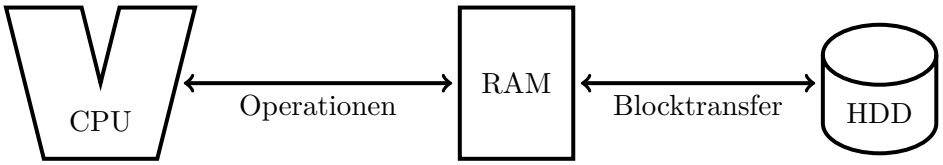


Abbildung 1.2: Einfaches Speichermodell nach Aggarwal und Vitter.

Die CPU kann direkt auf die Daten im internen Speicher zugreifen und diese verarbeiten. Daten auf dem externen Speicher hingegen müssen erst in den internen Speicher geladen werden bevor die CPU diese nutzen kann. Das Verschieben von Daten zwischen internem und externem Speicher ist im Vergleich zu restlichen Operationen langsam und dominiert somit die Rechenzeit vieler Anwendungen.

Um die Komplexitäten verschiedener Algorithmen auf externem Speicher entsprechend betrachten und bewerten zu können haben Aggarwal und Vitter [AV87] 1987 das Modell des externen Speichers beschrieben.

Es wird ein Modell verwendet, welches aus einem Prozessor sowie zwei Speicherstufen besteht. Die erste Speicherstufe, der Hauptspeicher, besitzt die Größe M, welche allerdings nicht groß genug für die Programmausführung ist. Die zweite Speicherstufe hingegen liefert den benötigten Speicherplatz, in diesem Fall ist dies die Festplatte. Das Schreiben der Daten auf die Festplatte geschieht durch die Übertragung von Blöcken der Größe B (i.d.R. 128KB bis 2MB) zwischen dem internen und externem Speicher, dem sogenannten Blocktransfer. Des Weiteren sei die Größe der Eingabe als N bezeichnet, diese liegt mit $\frac{N}{B}$ Blöcken auf der zweiten Speicherstufe. Hierbei ist anzumerken, dass in der Regel $M \ll N$ gilt. Sollen Daten auf die Festplatte geschrieben werden, deren Umfang kleiner als ein einzelner Block ist, so wird dennoch ein ganzer Block transferiert und auf der Festplatte gespeichert.

Jeder Blocktransfer zwischen internem und externem Speicher wird als *I/O-Operation* bezeichnet. Die Anzahl aller I/O-Operationen dient in diesem Modell als Metrik für die Komplexität und es gilt somit als Ziel dieses Modells die Anzahl der Zugriffe auf den externen Speicher, welche eben durch diese Metrik repräsentiert wird, zu minimieren.

1.2.1 Komplexität verschiedener Operationen

Als Grundlage für die späteren Untersuchungen in der Arbeit werden hier kurz die Komplexitäten verschiedener interessanter Operationen auf dem externen Speichermodell aufgeführt und erläutert. Operationen, die benötigt werden, sind unter Anderem der **sequentielle Scan** über die Daten auf der Festplatte, das **Sortieren** der Daten und die Operationen auf **Queues**, welche auf dem externen Medium gespeichert sind.

Sequentieller Scan

Bei einer $Scan(N)$ Operation werden die Daten N alle in sequentieller Reihenfolge abgearbeitet. Da jeder Lesezugriff auf der Festplatte jeweils einen Block der Größe B in den

Hauptspeicher transferiert und jeder Block insgesamt nur einmal geladen werden muss, ergibt sich daraus, dass $\left\lceil \frac{N}{B} \right\rceil$ Lesezugriffe auf der Festplatte ausreichen, um alle Daten einmal zu betrachten. Demnach gilt für die I/O-Komplexität: $Scan(N) = \mathcal{O}\left(\frac{N}{B}\right)$. Bei der Verwendung mehrerer Festplatten können diese zudem parallel arbeiten, wodurch sich die Komplexität bei Verwendung von D Festplatten auf $\mathcal{O}\left(\frac{N}{DB}\right)$ reduziert.

Sortieren

Das Sortieren von Daten auf externem Speicher ist, zum Beispiel durch einen *Multiway Merge Sort*, ebenfalls effizient möglich. Von der Eingabe N werden Stücke der Größe M in den internen Speicher gelesen und dort sortiert, sodass insgesamt $\ell = \left\lceil \frac{N}{M} \right\rceil$ sortierte Teilsequenzen entstehen, welche auf die externe Festplatte ausgelagert werden. Die einzelnen sortierten Teile werden anschließend mit einem *k-way-merging* Verfahren zu einer komplett sortierten Sequenz zusammengeführt (vgl. Abbildung 1.3). Die I/O-Komplexität dieses Verfahrens liegt amortisiert in $\mathcal{O}\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right) = Sort(N)$ [Vit08]. Die Zeitkomplexität liegt amortisiert in $\mathcal{O}(n \log n)$ und unterscheidet sich somit nicht von einem Sortierungsverfahren auf internem Speicher.

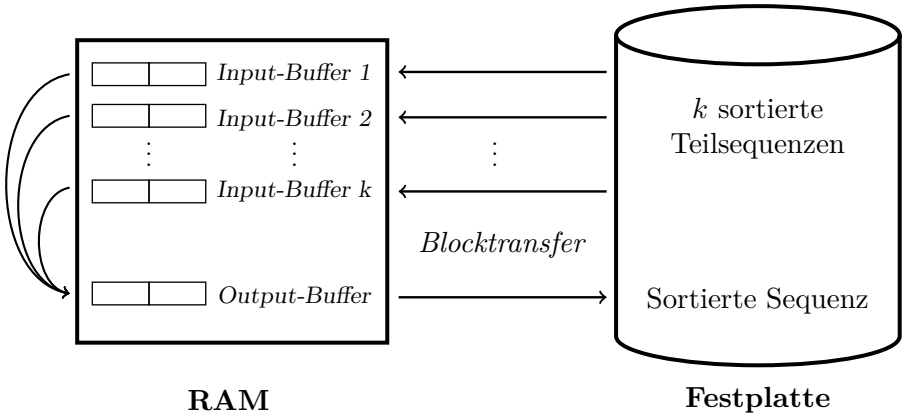


Abbildung 1.3: Abstrakte Darstellung des *k-way-merging*. Jede sortierte Teilsequenz besitzt einen Buffer im internen Speicher, in welchen jeweils die Blöcke der Teilsequenzen geschrieben werden. Im internen Speicher werden die Input-Buffer in einen Output-Buffer zusammengeführt, welcher wieder auf die Festplatte geschrieben wird, sobald er entsprechend gefüllt ist. Jegliche I/O-Operationen zwischen Hauptspeicher und Festplatte können dabei synchron zur eigentlichen Berechnung ausgeführt werden, wodurch die CPU nicht unnötig lange auf die Daten der Festplatte warten muss.

Prioritätswarteschlangen

Die verwendeten Prioritätswarteschlangen besitzen vier Operationen: $insert(x)$, $deleteFirst()$, $top()$ und $empty()$. Auf die genaue Herleitung der Komplexität für die Operationen wird in dieser Arbeit nicht genauer eingegangen. Hier benötigte I/O-Komplexität für das Einfügen ($insert(x)$) sowie das Löschen des obersten Elements ($deleteFirst()$) befinden sich jeweils amortisiert in $\mathcal{O}\left(\frac{1}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$ [BK98]. Die Zeitkomplexität hingegen dieser beiden Operationen befinden sich amortisiert in $\mathcal{O}(\log n)$. Zudem sind die Operationen $top()$, welche das oberste Element zurück gibt, und $empty()$, was angibt, ob die Prioritätswarteschlange leer ist, jeweils in $\mathcal{O}(1)$ ausführbar.

2. Die Louvain-Methode

Die Louvain-Methode ist ein Algorithmus zum Clustern von Graphen, der 2008 von Blondel et al. [BGLL08] vorgestellt wurde. Er zeichnet sich sowohl durch seine Performanz auf großen Graphen als auch seine qualitativ hochwertigen Ergebnisse aus. Da das Optimieren der Modularity eines Graphen $\mathcal{NP}$-schwer ist [BDG+08], verwendet der Algorithmus ein heuristisches Verfahren, welches nach dem Greedy-Prinzip immer die Schritte ausführt, die den größten Gewinn versprechen. Da das Vorgehen in beiden entworfenen (semi-)externen Methoden der Bachelorarbeit implementiert ist, wird es im folgenden genauer beschrieben.

Der Algorithmus beginnt damit, jedem Knoten sein eigenes Cluster zuzuweisen und testet anschließend für alle Knoten, in zufälliger Reihenfolge, ob es möglich ist, die Modularity zu verbessern, indem der Knoten einem anliegenden Cluster zugeordnet wird, mit dem der Knoten über eine Kante verbunden ist. Gibt es eine solche Verbesserungsmöglichkeit, wird der Knoten in das Cluster verschoben, welches den größtmöglichen Gewinn an Modularity liefert. Ist es nicht mehr möglich die Modularity mit solchen Verschiebungen einzelner Knoten zu verbessern, kann eine weitere Verbesserung nur erreicht werden, wenn ganze Gruppen von Knoten verschoben werden. Dazu wird der Algorithmus rekursiv auf einem vergröberten Graphen ausgeführt, der anhand der zuvor gefundenen Cluster erstellt wird. Ist es auch in der Rekursion nicht mehr möglich die Modularity zu steigern, so kehrt der Algorithmus aus dem rekursiven Aufruf zurück und die zuletzt gefundene Clusterzuordnung wird auf die ursprünglichen Knoten abgebildet.

2.1 Die ursprüngliche Variante

Zu Beginn wird einmalig jedem Knoten i des Graphen sein eigenes Cluster C_i zugeordnet. Anschließend unterteilt sich das Vorgehen in zwei Phasen, in denen die Modularity der Clusterung schrittweise vergrößert wird.

Phase 1: Local-Moving

In dieser Phase werden einzelne Knoten des Graphen zwischen benachbarten Clustern verschoben. Es wird in zufälliger Reihenfolge über die Knoten i iteriert. Dabei betrachtet man alle zu i, durch Kanten $\{i, j\} \in E$, benachbarten Cluster $C_{c(j)}$ und berechnet den Gewinn an Modularity, der erzielt wird, wenn Knoten i in das Cluster $C_{c(j)}$ von Knoten j verschoben wird. Dabei ist stets das Cluster C_k zu speichern, bei dessen Verschiebung von i der größte Gewinn erzielt wird. Hat man alle zu i benachbarten Cluster betrachtet, wird

i dem Cluster C_k zugewiesen, sofern der potentielle Gewinn an Modularity einen positiven Wert hat. Ist dies nicht der Fall, bleibt Knoten i weiterhin in seinem bisherigen Cluster. Dieser Vorgang wird so lange über alle Knoten des Graphen wiederholt, bis sich keine Verbesserung der Modularity mehr finden lässt.

Damit die Berechnungen der Verschiebungen performant bleiben, wird die Veränderung der Modularity nicht jedes mal mit der ausführlichen Formel 1.1 neu berechnet. Stattdessen berechnet man für die Verschiebung eines Knotens i vom Cluster $C_{c(i)}$ in ein Cluster $C_{c(j)}$ nur die Veränderung der Modularity durch ΔQ.

$$\Delta Q = \left[\frac{k_{i\_in}(j) - k_{i\_in}(i)}{w}\right] + \left[\frac{((vol(C_{c(i)}) - voli) - vol(C_{c(j)})) \cdot vol(i)}{2w^2}\right] \tag{2.1}$$

Hierbei sei $k_{i\_in}(j)$ definiert als die Summe aller Kantengewichte die von Knoten i in das Cluster $C_{c(j)}$ des Knoten j führen. Des weiteren ist $w = \frac{1}{2}\sum_{ij}\omega(\{i,j\})$ die Summe aller Kantengewichte im Graphen.

Phase 2: Kontraktion

Ein neuer Graph wird gebildet, indem die bis dahin ermittelten Cluster zu neuen Knoten kontrahiert werden. Zwischen jedem Paar dieser neuen Knoten wird eine Kante gebildet, sofern es zwischen den zugrundeliegenden Clustern mindestens eine Verbindungskante gab, Als Gewicht dieser neuen Kanten wird die Summe aller Kantengewichte der ehemaligen Verbindungskanten gewählt:

$$\sum_e \omega(e) \text{ mit } e \in \{(u,v) \in E | u \in C_i \wedge v \in C_j\}$$

Damit das Gesamtgewicht des Graphen unverändert bleibt, werden zudem alle internen Kanten eines Clusters als Schleife mit entsprechendem Gewicht dargestellt. Das Gewicht der Schleife von Cluster C_i ist hierbei $2\sum_e \omega(e)$ mit $e \subseteq C_i$. Durch diese Art der Konstruktion wird sichergestellt, dass die Modularity des neuen Graphen in der Ausgangssituation der neuen Clusterung (jeder Knoten hat genau ein Cluster zugewiesen) , **identisch** ist mit der Modularity der zuletzt bestimmten Clusterung des ursprünglichen Graphen. Die Gleichheit der Modularity bleibt auch nach Optimierungsschritten im neuen Graphen erhalten, d.h. eine auf dem neuen Graphen berechnete Modularity entspricht stets der Modularity für den ursprünglichen Graphen, die sich ergibt, wenn die Inhalte der gefundenen Cluster wieder auf die ursprünglichen Knoten zurückgeführt werden. Damit kann die Optimierung mittels Berechnung der Modularity auf dem neuen Graphen nahtlos weitergeführt werden.

Werden nun innerhalb des rekursiven Aufrufs Verschiebungen der neuen Knoten in andere Cluster durchgeführt, so können diese nach der Rekursion wieder auf die ursprünglichen Cluster abgebildet werden. Der Gewinn an Modularity, welcher in der Rekursion erzielt wurde, überträgt sich dank der zuvor erwähnten Eigenschaft somit auch auf die Modularity des eigentlichen Graphen.

Diese zweite Phase inklusive Rekursion wird so lange wiederholt, bis sich die Anzahl der Cluster durch den rekursiven Aufruf nicht mehr verkleinern lässt. Ein Nebeneffekt dieser Vorgehensweise ist, dass nicht nur Cluster eines Graphen gefunden werden, sondern ebenso deren jeweilige Sub-Cluster.

Komplexitätsbetrachtung

Die exakte Komplexität der Methode kann nicht genau bestimmt werden, da sowohl die Anzahl der Runden in der Local-Moving-Phase als auch die Rekursionstiefe nicht eindeutig ermittelbar sind. Allerdings vermuten die Autoren der ursprünglichen Methode dass sie insgesamt in $\mathcal{O}(n \log n)$ liegt [Blo14], mit Hauptaufwand in der ersten Local-Moving-Heuristik. Eine einzelne Runde dieser Heuristik lässt sich jedoch bezüglich ihrer Komplexität genauer betrachten:

Das Festlegen der zufälligen Knotenreihenfolge benötigt lediglich das Füllen eines Vektors der Größe n mit einer Permutation der Ausgangsknoten und hat somit Aufwand $\mathcal{O}(n)$. Anschließend wird jeder Knoten potentiell einmal verschoben, wobei alle seine ausgehenden Kanten (mehrfach) betrachtet werden. Für jeden Knoten i muss $k_{i\_in}(j)$ für alle benachbarten Cluster j berechnet werden, was mit einmaligem Betrachten der zu i inzidenten Kanten erledigt wird. Bei diesem Vorgang kann gleichzeitig auch das Volumen des Knotens berechnet werden. Die Volumen der verschiedenen Cluster werden stets abgespeichert und aktuell gehalten und sind in $\mathcal{O}(1)$ abrufbar. Somit ist auch die Berechnung nach Formel 2.1 direkt für jede von i ausgehende Kante in $\mathcal{O}(1)$ durchführbar. Insgesamt ergibt sich für den Worst-Case, dass von jedem Knoten i alle inzidenten Kanten zweimal betrachtet werden müssen, einmal um die Volumen der benachbarten Cluster zu berechnen und einmal um den Modularitygewinn darauf zu berechnen. Für den ganzen Graphen heißt das wiederum, dass jede Kante insgesamt vier mal betrachtet wird, nämlich jeweils zweimal ausgehend von den beiden anliegenden Knoten. Somit ergibt sich für eine einzelne Runde der Heursitik eine Worst-Case Laufzeit von $\mathcal{O}(m)$.

2.2 Multilevel-Refinement

Um mit der Louvain-Methode ein qualitativ besseres Ergebnis zu erzielen, wurde der ursprüngliche Algorithmus 2011 von Rotta und Noackum [RN11] minimal erweitert. Sie verfeinerten den Algorithmus, indem sie nach der zweiten Phase erneut die Verschiebung von einzelnen Knoten ermöglichten. Um dies zu erreichen führen sie die Local-Moving-Heuristik jeweils erneut aus, nachdem die Cluster entsprechend dem Ergebnis der Rekursion in der zweiten Phase gesetzt wurden. Diese erneuten Verschiebungen einzelner Knoten nach den Verschiebungen ganzer Cluster können weitere – eher kleine – Verbesserungen der Modularity bringen. Um den Ablauf der Louvain Methode inklusive des Multilevel-Refinements zu verdeutlichen dient der Algorithmus 2.1.

Algorithmus 2.1 : LOUVAIN-MR

Input : Graph $G = (V, E, \omega)$, Clusterzuordnung $c()$
Output : Clusterzuordnung $c()$

```
1  c() ← localMoving(G, c())
2
3  if numberOfClusters(c()) < |V| then
4      // Kontraktion
5      G_reduced ← reduceNetwork(G, c())
6      c_reduced() ← [1 ... numberOfNodes(G_reduced)]
7      clusterMapping: c(i) ↦ c_reduced(j) wobei c(i) = c_reduced(j) gilt
8
       // Rekursion
9      c_reduced() ←louvainMR(G_reduced, c_reduced())
10
       // Abbilden des Rekursionsergebnisses auf Ausgangsgraphen
11     for i ← 0 to |V| − 1 do
12         c(i) ← c_reduced(clusterMapping(c(i)))
13
       // Multilevel-Refinement
14     c() ← localMoving(G, Cluster[ ])
15 return c()
```

3. Einbeziehung von externem Speicher

Das Problem der Louvain-Methode ist nicht die Performanz auf großen Graphen, sondern die Datenmenge die dabei im internen Speicher gehalten wird. In seiner ursprünglichen Form behält der Algorithmus stets alle Knoten, Kanten sowie Cluster im Speicher, wodurch der benötigte Hauptspeicher mit der Größe der Graphen skaliert. Viele für die Clusterung zu betrachtende Graphen werden wiederum immer größer, da sie aus verschiedenen sozialen Netzwerken oder aus dem Internet als sogenannte Webgraphen generiert werden. Die verfügbare Größe des Hauptspeichers stellt damit eine Limitation dar, die eine Anwendung der Louvain-Methode bei sehr großen Graphen ohne Einbeziehung externer Speicher unmöglich macht.

Um diese Problematik zu lösen werden im Folgenden zwei Abwandlungen des Algorithmus betrachtet, die im Rahmen der Bachelorarbeit entwickelt wurden. Die erste Variante stellt eine semi-externe Lösung dar, welche trotz Einbeziehung externen Speichers möglichst performant bleiben sollte. Bei ihr werden nur die zu speichernden Kanten in externen Speicher verlagert, alles andere bleibt intern gespeichert. Die zweite Version zielt darauf ab, so viele Daten wie möglich auf externen Speicher auszulagern. Beide Varianten liefern dabei Ergebnisse der gleichen Qualität, unterscheiden sich allerdings in Performanz und Speicherbedarf, wie sich bei den in Kapitel 4 beschriebenen Experimenten gezeigt hat.

3.1 Semi-externe Variante

Die semi-externe Variante ähnelt der internen Implementierung der Louvain-Methode sehr. Den hauptsächlichen Unterschied stellt die Speicherung der Kantendaten des Graphen dar, welche auf dem externen Speicher gehalten werden. Zusätzlich wurde, im Gegensatz zur ursprünglichen Version, das Multilevel-Refinement aus Kapitel 2.2 implementiert um qualitativ bessere Ergebnisse zu erzielen.

Allgemein befinden sich die meisten benötigten Daten weiterhin im internen Speicher, dazu gehören die Knoten, Cluster und Cluster-Volumen. Auch solche Werte, die im Laufe des Algorithmus immer wieder berechnet werden, verbleiben im internen Speicher. Dazu gehören zum Beispiel das Volumen $vol(i)$ eines Knotens i oder die verschiedenen Werte für $k_{i\_in}()$. Der externe Speicher wird nur für die Kantendaten genutzt, die bei großen Graphen den meisten Speicherplatz benötigen.

Wie die ursprüngliche Methode auch, beinhaltet diese Variante die Möglichkeit bei jedem Durchgang der Local-Moving-Heuristik die Knoten in zufälliger Reihenfolge zu betrachten.

Hieraus ergibt sich eine neue Anforderung an die externe Speicherung der Kantendaten. Um zu vermeiden, dass bei der Überprüfung jeden Knotens ein Random-Zugriff auf den externen Speicher mit resultierender hoher Laufzeit für die I/Os erfolgt, wird für jede neu gewählte Reihenfolge der Knoten die Sortierung der Kantendaten im externen Speicher angepasst. Im Gegensatz zur semi-externen Variante stellt die Randomisierung bei der internen Implementierung der Louvain-Methode keinen großen Mehraufwand dar, da Random-Zugriffe im Hauptspeicher unkritisch sind.

Für die Verwendung einer Randomisierung sprechen folgende Gründe: Die Louvain-Methode beschreibt einen Greedy-Algorithmus, der oft nur lokale Optima als Ergebnis findet. Durch Permutieren der Bearbeitungsreihenfolge der einzelnen Knoten wird dem entgegengewirkt. Zum einen kann der komplette Algorithmus mit alternierenden Randomisierungen mehrfach ausgeführt werden, zum anderen könnte eine stets gleichbleibende Abarbeitungsreihenfolge der Knoten dazu führen, dass gewisse Verschiebungen nie in Betracht gezogen werden können. Außerdem lassen sich die verschiedenen Implementierungen (ursprünglich, semi-extern und voll-extern) besser vergleichen, wenn sie auf gleiche Weise vorgehen.

Zur Umsetzung der Randomisierung wurde ein externer Sorter genutzt, welcher einen *Multiway Merge Sort* entsprechend Abschnitt 1.2.1 implementiert. Wenn eine neue Reihenfolge festgelegt werden soll, wird jedem Knoten des Graphen eine zufällige Nummer zugewiesen, nach der anschließend für die Betrachtungsreihenfolge der Knoten sortiert wird.

Im Vorgriff auf die Komplexitätsbetrachtungen in Abschnitt 3.1.1 sei erwähnt, dass die Sortierung im externen Speicher viel Laufzeit beansprucht. Dieser Nachteil kann die Vorteile der Randomisierung (potentiell eine höhere Modularity) zunichtemachen wenn in der gesamten verfügbaren Zeit nicht mehr alle Optimierungsrunden durchführbar sind. Es ist daher sinnvoll, optional die Anzahl der Randomisierungen verringern zu können. Deshalb wurde in der hier vorgestellten Variante die Möglichkeit implementiert, festzulegen, nach wie vielen Runden des Local-Moving jeweils eine neue Betrachtungsreihenfolge der Knoten bestimmt werden soll. Auf dieser Basis wurden entsprechende Tests ausgeführt, deren Ergebnisse in Kapitel 4 festgehalten sind.

Ein weiterer Ansatz zur Reduzierung der Laufzeit, der allerdings auch schon in der Version von Blondel et al. [BGLL08] implementiert ist, ist der frühzeitige Abbruch der Local-Moving-Heuristik, sofern der Gewinn an Modularity zu gering wird oder zu viele Runden durchlaufen werden. Besonders bei Graphen mit vielen Knoten erweisen sich solche Bedingungen als sinnvoll, da sich hier in den späteren Runden der Heuristik oft nur noch kleine Veränderungen finden lassen, wodurch die Relation von Modularitygewinn zu benötigter Laufzeit einer einzelnen Runde des Local-Moving immer ungünstiger wird. Auch hierzu wurden verschiedene Testläufe durchgeführt, deren Ergebnisse in Kapitel 4 dargestellt sind.

3.1.1 Komplexitätsbetrachtungen

Die Gesamtlaufzeit dieser Variante ist, wie auch bei der eigentlichen Louvain-Methode, nicht exakt festzustellen, da auch hier nicht vorhersehbar ist, wie oft die Local-Moving-Phase und die Rekursion durchlaufen wird. Die Komplexität einzelner Teilphasen kann allerdings bestimmt werden. Im Folgenden werden daher folgende Bestandteile der gesamten Clusterung betrachtet:

- Eine **einzelne Runde** der Local-Moving-Heuristik
- Kontraktion zu einem neuen Graphen
- Rückführen des Ergebnisses auf den Ursprungsgraphen

Für Rekursion und Refinement ist lediglich eine Messung der tatsächlichen Laufzeit möglich. Entsprechende Messergebnisse sind in Kapitel 4 dargestellt.

Local-Moving-Heuristik

Um eine Runde der Heuristik durchzuführen, wird zunächst die Reihenfolge festgelegt, in der die Knoten betrachtet werden. Dazu wird ein Vektor im internen Speicher mit zufälligen Zahlen gefüllt, die bestimmen, welcher Knoten an welcher Stelle betrachtet wird. Das Füllen des Vektors benötigt $\mathcal{O}(n)$ Zeit. Anschließend werden die Kanten entsprechend dieses Vektors sortiert und auf die externe Festplatte geschrieben. Das Sortieren auf externen Medien liegt, wie auch das Sortieren im internem Speicher, in $\mathcal{O}(n \log n)$. Die I/O-Komplexität befindet sich gemäß Abschnitt 1.2.1 in $\mathcal{O}\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$.

Anschließend wird jeder einzelne Knoten i wie folgt abgearbeitet: Es werden jeweils alle Kanten e, die von diesem Knoten ausgehen, in einen internen Vektor geschrieben. Wenn es w solcher Kanten e gibt, kann nun in $\mathcal{O}(w)$ das Volumen $vol(i)$ des Knoten sowie alle Werte $k_{i\_in}(k)$ zu benachbarten Clustern $C_{c(k)}$ berechnet und abgespeichert werden. Die Werte für $k_{i\_in}()$ werden dabei in einen internen Vektor geschrieben. Dadurch sind alle notwendigen Daten vorhanden, um für jede von i ausgehende Kante die Veränderung der Modularity gemäß Formel 2.1 zu berechnen, wenn der Knoten i in das Cluster verschoben wird, in dem der andere Endknoten der Kante liegt. Pro Kante kann die Berechnung damit in $\mathcal{O}(1)$ ausgeführt werden. Wird ein Knoten anschließend verschoben, so müssen nur die beiden Volumen der betroffenen Cluster und der Clusterwert für den Knoten i entsprechend aktualisiert werden. Alle diese Daten befinden sich im internen Speicher und können in $\mathcal{O}(1)$ aktualisiert werden. Da diese Prozedur für alle Kanten e aller Knoten i abläuft, ergibt sich insgesamt, dass über die gesamten Kantendaten im externen Speicher genau einmal sequentiell gelesen wird, was eine I/O-Komplexität von $Scan(N)$ (siehe Abschnitt 1.2.1) aufweist. Als dominierender Faktor für die Größenordnung der Laufzeit ergibt sich also die Anzahl der Kanten und damit $\mathcal{O}(m)$, wobei von Graphen ausgegangen wird, für die $m > n$ gilt.

Es ist zu sehen, dass die Randomisierung somit die Laufzeit der Local-Moving-Heuristik dominiert. Auch ist festzuhalten, dass Schreibvorgänge auf das externe Medium ebenfalls nur bei Permutation der Betrachtungsreihenfolge erforderlich werden. Wird eine Runde des Local-Movings allerdings ohne die Randomisierung durchgeführt, so fällt die Sortierung der Daten auf dem externen Speicher weg und es müssen keine Daten auf die Festplatte geschrieben werden. Die einzelne Runde kann deutlich schneller berechnet werden, was ein weiteres Argument dafür liefert, nicht nach jeder Local-Moving-Runde zu randomisieren.

Kontraktion

Für die Reduzierung des Graphen werden den Knoten i zunächst neue IDs gemäß ihren Werten $c(i)$ zugewiesen. Anschließend wird ein externer Sorter (vgl. Abschnitt 1.2.1) mit allen Kanten befüllt, welcher diese nach den neuen IDs der Start- und Zielknoten sortiert. Die Sortierung ist damit gleichbedeutend mit einer Sortierung nach Start- und Ziel-Cluster. Das Sortieren benötigt $\mathcal{O}(n \log n)$ Zeit und führt zu $\mathcal{O}\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$ I/Os. Sind die Kanten entsprechend sortiert werden sie sequentiell bearbeitet um sie zu den Kanten des neuen Graphen zusammenzuführen. Hierbei werden für die Kanten mit gleichen Endknoten lediglich die Kantengewichte aufaddiert und mit den Ergebnissen ein neuer externer Vektor gefüllt. Die dafür benötigte Zeit ist $\mathcal{O}(m)$. Das Schreiben einer neuen Kante auf die Festplatte benötigt jeweils einen I/O. In der Regel werden aber bei jedem Schreibzugriff auf die Festplatte direkt mehrere solcher neuen Kanten geschrieben. Für die gesamte Kontraktion ergibt sich somit insgesamt eine Komplexität von $\mathcal{O}(n \log n)$. Der Vorgang der Kontraktion wird in Abbildung 3.1 verdeutlicht.

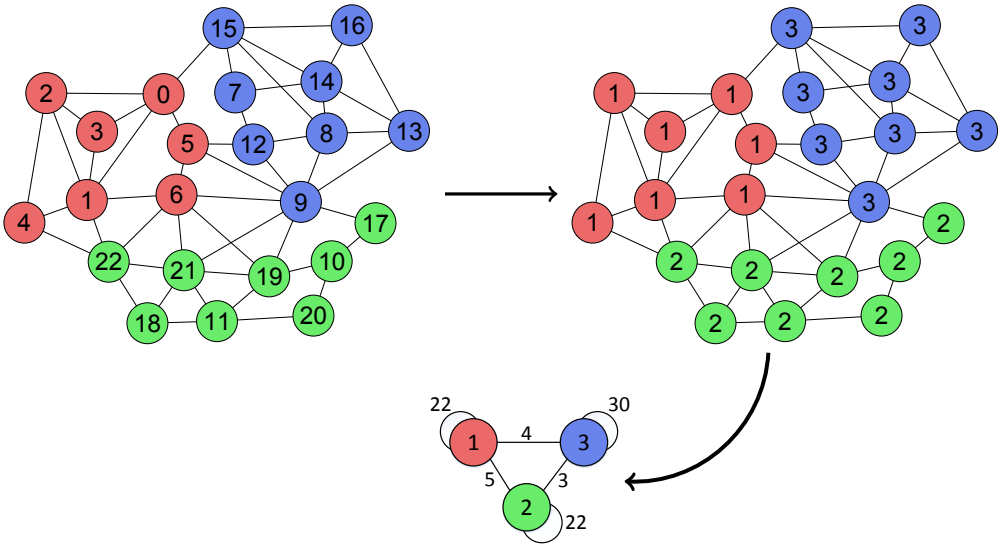


Abbildung 3.1: Veranschaulichung der Kontraktion. Die Farben dienen der Darstellung der Clusterzuordnung. Die Knoten-IDs werden zunächst entsprechend der Cluster neu vergeben und anschließend werden die Knoten mit gleicher ID kontrahiert. Die Kantengewichte werden bei der Kontraktion so aufaddiert, dass sich die Volumen der einzelnen Cluster nicht verändern.

Rückführen des Ergebnisses auf den Ursprungsgraphen

In diesem Schritt wird lediglich für jeden Knoten des Ursprungsgraphen das zugewiesenen Cluster entsprechend der Ergebnisse der Rekursion aktualisiert. Da die Clusterzuweisung in der Rekursion stets intern gespeichert wird, ist das Zuweisen der neuen Werte demnach einfach ein erneutes zuweisen der Cluster für jeden Knoten. Somit ergibt sich sofort die Komplexität $\mathcal{O}(n)$ für diesen Vorgang.

3.2 Voll-externe Variante

Die zweite in der Bachelorarbeit entwickelte Variante wurde so konzipiert, dass möglichst alle Daten auf den externen Speicher verlagert werden. Demnach werden nicht nur die Kantendaten extern gespeichert, sondern auch die Informationen zu den einzelnen Knoten, wie deren IDs oder Clusterzugehörigkeit.

Zu jeder Kante werden nun auch weitere für die Berechnung benötigten Informationen gespeichert, sodass beim Betrachten der Kanten auch Informationen über inzidente Knoten und Cluster verfügbar sind. Auch Knoten werden formal als Pseudokanten gespeichert, um Vorteile bei den Zugriffen auf den externen Speicher zu erzielen. Alle relevanten Daten sind somit strukturgleich abgelegt.

Um alle diese Informationen auf dem externen Speicher aktuell zu halten wird das sogenannte *Time Forward Processing* angewendet, welches Abläufe auf dem externen Speicher optimiert und in diesem Fall einen Informationsaustausch bezüglich Clusterzuweisungen zwischen Knoten ermöglicht.

Ein Problem, auf das später genauer eingegangen wird, sind die Daten zu den Volumen der einzelnen Cluster. Für sie ist eine Speicherung auf externem Speicher nicht sinnvoll. Eine genauere Erläuterung folgt in Abschnitt 3.2.5. Die Bezeichnung **voll-externe Variante** sollte deshalb so verstanden werden, dass zumindest alle Daten zu den ursprünglichen Elementen des Graphen (Knoten und Kanten) sowie die Clusterzuweisungen extern gehalten werden.

3.2.1 Time Forward Processing

Das *Time Foward Processing* (TFP) dient der I/O-Optimierung auf externem Speicher. Die Datenstruktur die für das TFP benötigt wird ist ein *gerichteter azyklischer Graph (DAG)*, wie zum Beispiel in Abbildung 3.2 zu sehen ist.

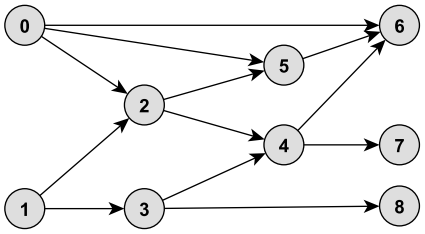


Abbildung 3.2: Beispiel eines einfachen gerichteten azyklischen Graphen.

Der Graph wird dabei in topologischer Ordnung extern gespeichert, das heißt, für zwei Knoten i und j die durch die Kante (i, j) verbunden sind, liegen die Daten für Knoten i im Speicher vor den Daten des Knoten j. Eine Topologische Ordnung des Graphen aus Abbildung 3.2 wäre dabei **0-1-2-3-4-5-8-7-6** oder auch **1-0-3-2-4-5-6-8-7**. Der Sinn des TFP liegt darin, dass man nun beim Bearbeiten eines Knotens i Daten an alle seine Folgeknoten j, gemäß den Kanten (i, j) schicken kann.

Im Weiteren werden die Begriffe *Betrachtungswert* und *Betrachtungsreihenfolge* verwendet. Die *Betrachtungsreihenfolge* gibt an, in welcher Reihenfolge die Knoten in einer Local-Moving-Runde betrachtet werden. Der *Betrachtungswert* ist ein Zufallswert, der einem Knoten zugewiesen wird. Durch Sortierung der Knoten nach diesem *Betrachtungswert* ergibt sich die *Betrachtungsreihenfolge*. Der Betrachtungswert zum Knoten i wird mit $Bet(i)$ bezeichnet.

Das DAG-Modell bildet die Grundlage für die Local-Moving-Heuristik in der voll-externen Implementierung: Um dieses Modell nutzen zu können, wird zunächst der vorliegende ungerichtete Graph zu einem doppelt gerichteten Graphen tansformiert, wobei jede ungerichtete

Kante $\{i, j\}$ durch zwei gerichtete Kanten (i, j) und (j, i) ersetzt wird. Bei der Randomisierung wird jedem Knoten i ein Betrachtungswert $Bet(i)$ zugewiesen. Um eine topologische Ordnung zu erhalten verwendet man diese Betrachtungswerte, indem aus dem doppelt gerichteten Graphen nur noch solche Kanten (i, j) für das TFP gespeichert werden, bei denen $Bet(i) < Bet(j)$ gilt. Zu jeder dieser Kanten (i, j) wird auch der Betrachtungswert des Startknotens $Bet(i)$ mit abgespeichert, da er für die spätere Sortierung der Kanten essentiell ist.

Das Vorgehen ist in Abbildung 3.3 verdeutlicht. Dort wird im ersten Schritt aus einem ungerichteten Graphen ein doppelt gerichteter Graph generiert, mit zugewiesenen Betrachtungswerten an den Knoten. Im zweiten Schritt wird eine topologische Ordnung gemäß der Betrachtungswerte hergestellt.

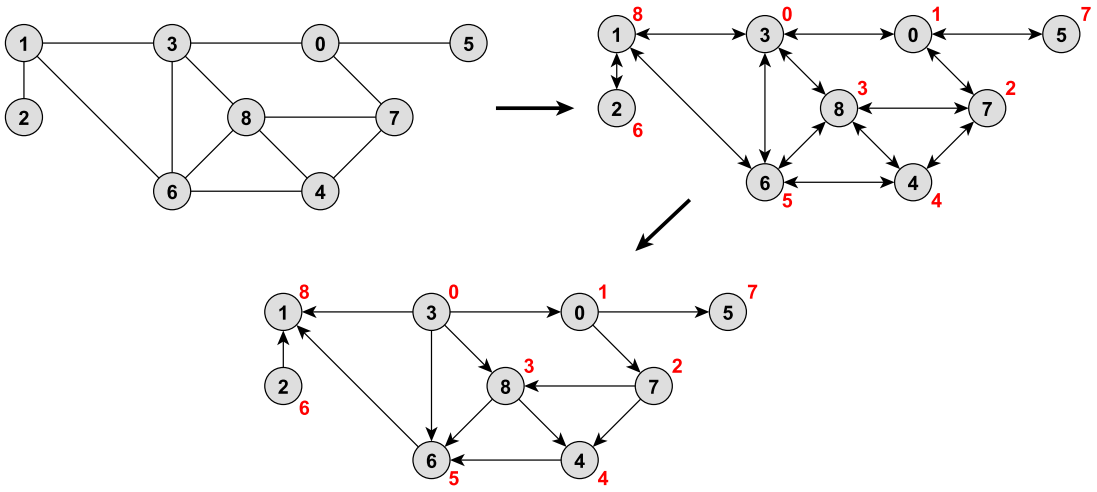


Abbildung 3.3: Veranschaulichung der Generierung eines DAG aus dem Ursprungsgraphen mit gesetzten Betrachtungswerten für die Knoten (**rot**).

Auf solchen Graphen kann nun die Local-Moving-Heuristik durchgeführt werden. Zunächst sind nicht für jeden Knoten alle benötigten Kanten gesetzt, um bei der Verschiebungsberechnung auch alle Möglichkeiten zu betrachten. Für den ersten zu betrachtenden Knoten i_1 sind aber alle Kanten gespeichert, da er den niedrigsten Betrachtungswert hat. Nach der Betrachtung von i_1 werden alle gespeicherten Kanten (i_1, j) gelöscht und stattdessen die Kanten (j, i_1) gespeichert. Dadurch stehen jetzt auch alle Kanten für Knoten i_2 zur Verfügung, denn bei diesem Knoten konnte wegen des Betrachtungswerts maximal die Kante (i_2, i_1) fehlen, sofern i_1 und i_2 überhaupt verbunden sind. Da diese Kantendrehungen nach der Bearbeitung jeden Knotens durchgeführt wird, stehen stets alle Kanten eines Knotens zur Verfügung sobald dieser betrachtet wird.

Ab dem zweiten Knoten der betrachtet wird, kann es vorkommen, dass Kanten (i, j) auftreten mit $Bet(i) > Bet(j)$. Beim Löschen einer solchen Kante muss die gedrehte Kante (j, i) nicht mehr als Ersatz eingefügt werden, da Knoten j bereits vor Knoten i betrachtet wurde. Stattdessen wird die Kante (j, i) in einen separaten DAG eingefügt, der für die nächste Local-Moving-Runde verwendet wird. Je nach Einstellung der Randomisierung wird dem Startknoten dabei ein neuer Betrachtungswert zugewiesen.

Nach einem vollständigen Durchgang einer Local-Moving-Runde sind mit dieser Vorgehensweise alle Kanten aus dem aktuellen DAG entfernt und der DAG für die nächste Runde ist vollständig aufgebaut.

Das beschriebene Vorgehen wird anhand des Beispiels aus Abbildung 3.3 zusammen mit Abbildung 3.4 noch einmal dargestellt: Zuerst wird der Knoten betrachtet, der den kleinsten

Betrachtungswert hat, im Beispiel aus Abbildung 3.3 wäre das Knoten ***3***. Für ihn sind alle ausgehenden Kanten deckungsgleich mit den Kanten des ursprünglichen Graphen. Ist die Verschiebungungsberechnung für den ersten Knoten (***3***) erfolgt, so wird der DAG aktualisiert. Alle ausgehenden Kanten des Knotens ***3*** werden umgedreht, womit nun für alle zuvor benachbarten Knoten (im Beispiel Knoten ***0***, ***1***, ***6*** und ***8***) bekannt ist, dass diese ebenfalls eine Kante zu Knoten ***3*** besitzen. Für den Knoten mit dem zweitniedrigsten Betrachtungswert (Knoten **0**) ergibt sich nun, dass er ebenfalls nur noch ausgehende Kanten besitzt und die Verschiebung berechnet werden kann.

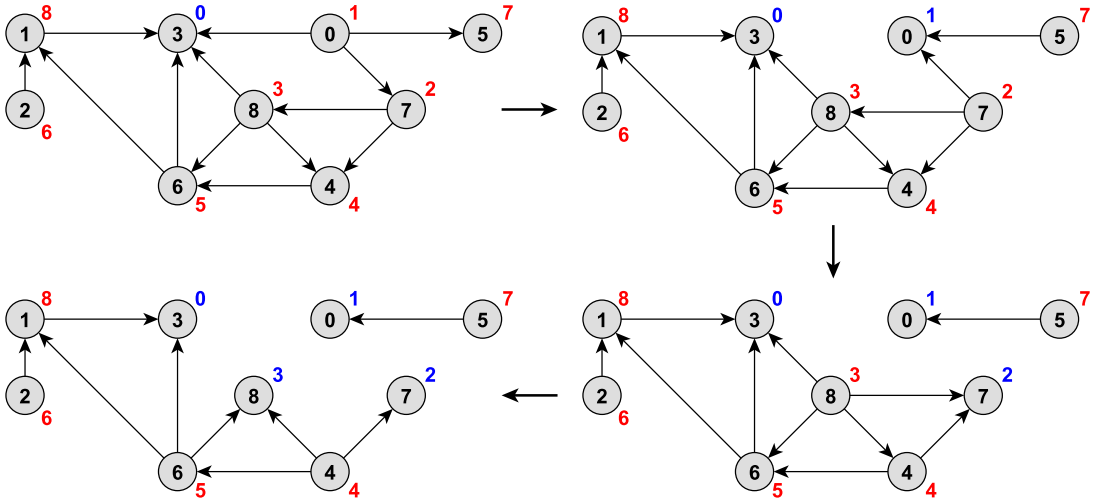


Abbildung 3.4: Nach jeder Betrachtung eines Knotens werden die Kanten umgedreht. Unbetrachtete Knoten haben einen **roten** Betrachtungswert, betrachtete Knoten einen **blauen**. Es ist zu sehen, dass der Knoten der als nächstes betrachtet wird stets nur ausgehende Kanten besitzt, keine eingehenden. Das zweite Umdrehen einer Kante wird in einem einzelnen Durchgang nicht benötigt, daher werden solche Kanten hier nicht angezeigt.

Nach der generellen Vorstellung der Vorgehensweise wird im folgenden auf die Aspekte einer effizienten Realisierung eingegangen. Dazu gehören die Anforderungen an die Randomisierung und an die Struktur der gespeicherten Daten, die erfüllt sein müssen, um die Laufzeit und die Anzahl der benötigten I/Os zu optimieren.

3.2.2 Aspekte der Randomisierung

Für die Betrachtungswerte $Bet(i)$ der Zufallsreihenfolge ist es wichtig, dass diese schnell generiert werden und auch für einen Knoten mehrfach abgefragt werden können. Letzteres spielt deshalb eine Rolle, weil beim Generieren des DAGs für die Kanten entschieden werden muss, in welche Richtung sie zeigen. Da sie von klein nach groß (Betrachtungswerte) zeigen sollen, wird jeweils $Bet(i)$ für Start- und Zielknoten generiert und geschaut, welcher kleiner ist.

Normale Zufallsgeneratoren sind zu langsam und haben das Problem, dass sie nur über Umwege einen Zufallswert in Abhängigkeit einer Knoten-ID generieren können, der bei mehrfacher Abfrage unverändert bleibt. In der Implementierung wurde für die Randomisierung daher eine Hashfunktion genutzt. Diese kann mit unterschiedlichen Seeds (für unterschiedliche Betrachtungsreihenfolgen) aufgerufen werden. Die Funktion bildet ihre Eingabewerte (Knoten-IDs) auf Hashwerte ab, welche als Betrachtungswerte für eine Local-Moving-Runde genutzt werden können. In Abbildung 3.5 wird das Randomisieren durch

Hashing verdeutlicht, die Knoten-IDs bekommen im ersten Schritt einen Hashwert zugewiesen, im zweiten Schritt wird nach diesen Hashwerten sortiert, was zu einer Permutation der Reihenfolge führt.

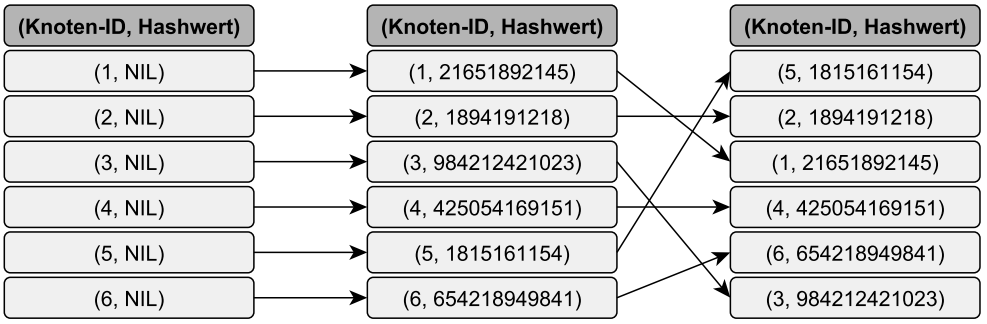


Abbildung 3.5: Generierung einer zufälligen Betrachtungsreihenfolge der Knoten durch die Verwendung einer Hashfunktion.

Nicht jede Hashfunktion bildet ihre Eingabe zufällig verteilt auf die Zielmenge ab. Deshalb wurde in der Arbeit zunächst der CityHash von Google [PA11] verwendet, welcher dahingehend konzipiert ist, dass die gehashten Werte auch wirklich zufällig verteilt sind. In den Experimenten hingegen wurde auf ein simpleres Hashing zurückgegriffen, das eine deutlich bessere Laufzeit aufweist und immer noch eine ausreichende Streuung der Hashwerte erzielt.

Mit der beschriebenen Funktion lassen sich die Kanten innerhalb des DAGs auch sehr leicht umdrehen. In der gespeicherten Kante müssen lediglich Start- und Zielknoten vertauscht und der Betrachtungswert mit dem Hashwert des ursprünglichen Zielknotens überschrieben werden. Auch bei der Generierung eines neuen DAGs für weitere Local-Moving Runden können die Betrachtungswerte schnell variiert werden, indem für die Hashfunktion ein neuer Seed genutzt wird. Zwar kann die in dieser Arbeit verwendete Hashfunktion nicht direkt mit einem Seed variiert werden, aber durch hashen einer Kombination aus Seed und Knoten-ID erhält man dennoch den gewünschten Effekt.

Die Betrachtungswerte müssen eigentlich eindeutig sein, um auch eine eindeutige Sortierung zu gewährleisten. Bei einer Hashfunktion sind Kollisionen aber prinzipiell möglich. Dem wird später bei der Sortierung Rechnung getragen, indem zum Betrachtungswert ein weiteres eindeutiges Sortierkriterium ergänzt wird.

3.2.3 Knotenverschiebung und Datenstruktur der Kanten

In der Local-Moving-Heuristik liegt das Hauptmerkmal auf der Verschiebungsberechnung der einzelnen Knoten. Im Folgenden wird erläutert, welche Daten gespeichert werden um effiziente Berechnungen zu gewährleisten.

Um zu prüfen, ob ein Knoten i zu verschieben ist, werden folgende Informationen benötigt:

- **Gesamtgewicht des Graphen:** Entspricht der Summe aller Kantengewichte im Graphen. Da dieser Wert sich nicht verändert wird er zu Beginn intern gespeichert.
- **Cluster $C_{c(i)}$:** Das Cluster des aktuellen Knoten. Damit kann festgestellt werden, ob ausgehende Kanten in das eigene Cluster $C_{c(i)}$ führen. Der Wert für $c(i)$ wird nicht in jeder Kante (i, j) abgespeichert, sondern einmalig in einer Pseudokante $(i, -1)$, wobei die -1 anzeigt, dass es sich nicht um eine normale Kante handelt. Durch die Sortierung wird sichergestellt, dass die Pseudokante $(i, -1)$ vor den echten Kanten (i, j) liegt und die Daten somit verfügbar sind, wenn Knoten i betrachtet wird.

- **Cluster $C_{c(j)}$:** In dieses Cluster kann potentiell verschoben werden, wenn es sich von $C_{c(i)}$ unterscheidet. Da $c(j)$ vom Endknoten abhängt, muss der Wert pro Kante extra gespeichert werden. Für die zuvor beschriebenen Pseudokanten gibt es kein $c(j)$, so dass stattdessen der Wert $c(i)$ gespeichert wird.
- **Volumen $vol(i)$:** Das Volumen des Knotens wird sowohl bei der Berechnung der Modularity-Änderung als auch später bei der eigentlichen Verschiebung benötigt. Der Wert wird wie $c(i)$ in der Pseudokante gespeichert.
- **Volumen $vol(j)$:** Das Volumen des Endknotens wird pro Kante gespeichert um den Hauptspeicherbedarf für die Clustervolumen zu reduzieren. Details dazu folgen in Kapitel 3.2.5. Für die Pseudokanten gibt es kein $vol(j)$, so dass in ihr stattdessen der oben erwähnte Wert von $vol(i)$ gespeichert wird.
- **$k_{i\_in}(j)$):** Hiermit ist die Summe der Kantengewichte gemeint, die von Knoten i aus in das benachbarte Cluster $C_{c(j)}$ führen. Dieser Wert kann bei Betrachtung der Kanten aufaddiert werden. Deshalb wird in jeder Kante auch das entsprechende Kantengewicht gespeichert. Eine weitere Sortierung der Kanten nach Zielclustern erweist sich hierbei als sinnvoll, damit Kanten beieinander liegen, die in dasselbe Cluster führen. Der Wert muss zwar für alle benachbarten Cluster berechnet werden, aber die Werte werden nicht alle gleichzeitig benötigt. Zu einem Zeitpunkt wird immer nur ein benachbartes Cluster betrachtet.
- **$k_{i\_in}(i)$):** Hiermit ist die Summe der Kantengewichte gemeint, die von Knoten i aus in das eigene Cluster $C_{c(i)}$ führen. Die Berechnung erfolgt analog zum vorhergehenden Punkt.
- **Clustervolumen:** Von allen Clustern werden die Volumen benötigt. Diese werden, wie zuvor erwähnt, im Hauptspeicher abgelegt. Weitere Details dazu folgen in Kapitel 3.2.5.

Um die benötigten Daten bereitzustellen ergibt sich für die Speicherung der Kanten die in Abbildung 3.6 gezeigte Struktur. Insgesamt ergibt sich für jede Kante ein Speicherbedarf von *288 Bit*.

Start	Ziel	Gewicht	Betrachtungs-wert	Ziel-cluster	Volumen Zielknoten
32 Bit	32 Bit	64 Bit	64 Bit	32 Bit	64 Bit

Abbildung 3.6: Datenstruktur einer einzelnen Kante.

In Algorithmus 3.1 wird die Verschiebungsberechnung eines Knotens und die damit verbundene Berechnung des Modularity-Gewinns dargestellt. Dabei wird bereits eine bestimmte Sortierfolge der Daten vorausgesetzt, die im folgenden beschrieben wird.

Im Vorfeld wurden bereits mehrere Kriterien genannt, nach welchen sortiert werden muss. Die folgende Aufzählung trägt alle Sortierregeln der verwendeten Prioritätswarteschlange, in absteigender Relevanz, zusammen:

1. **Betrachtungswert:** Sorgt dafür, dass alle ausgehenden Kanten eines Knotens beieinander liegen und gemäß des TFP abgearbeitet werden können und dass die gewünschte Betrachtungsreihenfolge (Randomisierung) der Knoten eingehalten wird.
2. **Startknoten:** Zwei unterschiedliche Knoten können denselben zufälligen Betrachtungswert bekommen, denn beim Hashing kann es Kollisionen geben. Durch die Sortierung nach den Startknoten gibt es auch bei solchen Kollisionen noch eine eindeutige Sortierung.

3. **Pseudokante:** Wird ein Knoten betrachtet, ist es zunächst notwendig zu wissen, welches Volumen er hat und in welchem Cluster er sich befindet. Die Pseudokante, die diese Informationen enthält, muss folglich an den Anfang sortiert werden. Die Pseudokante wird am Wert -1 des Zielknotens erkannt. (Hinweis: Es wird ein spezieller Vergleichoperator für die Sortierung genutzt, der eine solche Sortierung ermöglich. Es kann nämlich nicht einfach der Zielknoten als drittes Sortierfeld gewählt werden, weil für die normalen Kanten die Sortierung nach Zielclustern Vorrang hat.)
4. **Zielcluster:** Für die Verschiebung wird zu einem Zeitpunkt stets nur ein benachbartes Cluster betrachtet. Deshalb ist es günstig, wenn alle Kanten, die vom betrachteten Knoten in ein und dasselbe Cluster zeigen, nebeneinander liegen.
5. **Zielknoten:** Die Sortierung nach dem Zielknoten wird für den eigentlichen Ablauf nicht benötigt, aber nur durch Einbeziehung des Zielknotens wird die Sortierung eindeutig und die Menge der Kanten total geordnet.

Um die Sortierung zu veranschaulichen ist in Tabelle 3.1 eine Beispielsortierung aufgezeigt. Ausgangspunkt war der Graph aus Abbildung 3.1. Die aufgezeigten Einträge gehören hierbei zu Knoten ***6*** und ***9***, zur Vollständigkeit wurden den Knoten folgende Betrachtungswerte zugewiesen: $Bet(6) = 7$ und $Bet(9) = 4$.

Bet(Start)	Start	Ziel	Cluster	*Gewicht*	*vol*(*Ziel*)
⋮	⋮	⋮	⋮	⋮	⋮
4	9	-1	3	*1*	*8*
4	9	8	3	*1*	*5*
4	9	12	3	*1*	*4*
4	9	13	3	*1*	*4*
4	9	5	1	*1*	*4*
4	9	6	1	*1*	*6*
4	9	17	2	*1*	*2*
4	9	19	2	*1*	*5*
4	9	21	2	*1*	*6*
⋮	⋮	⋮	⋮	⋮	⋮
7	6	-1	1	*1*	*6*
7	6	1	1	*1*	*6*
7	6	5	1	*1*	*4*
7	6	19	2	*1*	*5*
7	6	21	2	*1*	*6*
7	6	22	2	*1*	*5*
7	6	9	3	*1*	*8*
⋮	⋮	⋮	⋮	⋮	⋮

Tabelle 3.1: Beispielsortierung der Kanten zu Knoten ***6*** und ***9*** aus Abbildung 3.1. Werte nach denen *nicht sortiert* wurde stehen am Ende.

Algorithmus 3.1 : VERSCHIEBUNGSBERECHNUNG

Input : **Prioritätswarteschlange** kanten, **Knoten** i

```
   // Verschiebungsberechnung auf einem Knoten
 1 while kanten.top().Startknoten == i do
 2     aktuelleKante = kanten.top()
 3     kanten.pop()
 4
       // Pseudokante: Alle nötigen Daten werden intern gespeichert
 5     if aktuelleKante.Zielknoten == -1 then
 6         knoten_clu ← c(i)
 7         knoten_vol ← vol(i)
 8         cluster_vol ← vol(knoten_clu)
 9
10     if aktuelleKante.Zielcluster == knoten_clu then
11         k_i_in(knoten_clu) ← aktuelleKante.Gewicht
12
           // Alle weiteren Kanten zum eigenen Cluster
13         while kanten.top().Zielcluster == knoten_clu do
14             aktuelleKante = kanten.top()
15             kanten.pop()
16             k_i_in(knoten_clu) ← k_i_in(knoten_clu) + aktuelleKante.Gewicht
17     else
           // Handelt ein benachbartes Cluster ab
18         z_Cluster ← aktuelleKante.Zielcluster
19         z_Cluster_vol ← vol(z_Cluster)
20         k_i_in(z_Cluster) ← aktuelleKante.Gewicht
21
           // Alle weiteren Kanten zu diesem Cluster
22         while kanten.top().Zielcluster == z_Cluster do
23             aktuelleKante = kanten.top()
24             kanten.pop()
25             k_i_in(z_Cluster) ← k_i_in(z_Cluster) + aktuelleKante.Gewicht
26
27         modGewinn = berechneModGew(knoten_vol,
           k_i_in(z_Cluster), cluster_vol, z_Cluster_vol)
28
29     if modGewinn > maxModGewinn then
30         maxModGewinn ← modGewinn
31         bestesCluster ← z_Cluster
32
33 realModGain = maxModGain - (k_i_in(knoten_clu) / totalWeight)
34 if realModGain > 0 then
35     verschiebeKnoten(i, bestesCluster)
```

3.2.4 Datenstrukturen

In der Implementierung wurde neben der bereits erwähnte **Prioritätswarteschlange (PQ)** zur Sortierung der Kanten auch ein **externer Sorter (EXS)** verwendet. Die Prioritätswarteschlange würde zwar für ein funktionierendes Programm vollkommen ausreichen, aber durch den Einsatz eines externen Sorters lässt sich eine deutliche Beschleunigung erzielen.

Pro Durchgang der Local-Moving-Heuristik stehen somit zwei Container (PQ und EXS) mit Kantendaten zur Verfügung. Die nächste zu bearbeitende Kante wird jeweils aus diesen beiden Containern durch ein Merging bestimmt (zu Vergleichen mit dem Vorgehen bei einem Mergesort).

Die initiale Speicherung der Kantendaten für eine Runde der Local-Moving-Heuristik findet immer im EXS statt. Die Daten sind dabei zunächst unsortiert. Erst wenn sich alle nötigen Einträge im EXS befinden wird sortiert. Dieses Vorgehen ist signifikant schneller als die Verwendung der PQ, welche zu jedem Zeitpunkt die Einhaltung der Sortierreihenfolge garantiert muss.

Zu Beginn einer Runde der Local-Moving-Heuristik ist die PQ somit noch leer. Während der Bearbeitung werden Kanten (i, j) durch die umgedrehten Kanten (j, i) ersetzt. War dabei $Bet(i) < Bet(j)$, so wird die Kante (j, i) noch für die aktuelle Runde benötigt und deshalb in die PQ eingefügt. Dabei wird ausgenutzt, dass die PQ die neue Kante an die richtige Stelle sortiert. Für den Fall, dass $Bet(i) > Bet(j)$ ist, wird die Kante (j, i) erst wieder in der nächsten Runde der Heursitik benötigt. Diese Kante wird deshalb in einen EXS für den nächsten Durchgang eingefügt.

Pseudokanten befinden sich bei dieser Vorgehensweise nur im EXS. Wenn der durch eine Pseudokante beschriebene Knoten betrachtet wurde, wird eine neue Pseudokante mit aktualisierten Daten in den EXS für die nächste Runde eingefügt.

Dieser Hybrid aus PQ und EXS sei im Folgenden als **Kantenspeicher** bezeichnet und ist in Abbildung 3.7 dargestellt:

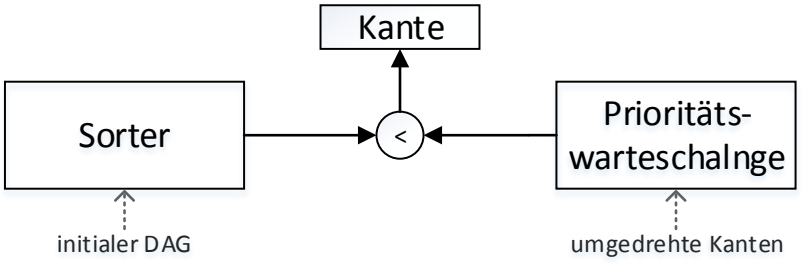


Abbildung 3.7: Veranschaulichung des Kantenspeichers.

Wie bereits erwähnt, wird in einer Local-Moving-Runde, die auf einem Kantenspeicher arbeitet, bereits ein weiterer Kantenspeicher für die nächste Runde gefüllt. Daraus folgt, dass es insgesamt notwendig ist, zwei solcher Kantenspeicher zu verwenden, jeweils einen für die aktuelle Runde und einen für die darauf folgende. Diese beiden Speicher können im Wechsel verwendet werden, sodass nicht jedes mal ein neuer Kantenspeicher instantiiert werden muss. Da sowohl PQ als auch EXS jeweils einen gewissen Anteil an internem Speicher benötigen fällt durch die doppelte Verwendung des Kantenspeichers auch ein höherer interner Speicherverbrauch an, der allerdings durch den resultierenden Performanzgewinn vollkommen gerechtfertigt ist.

3.2.5 Effiziente Speicherung der Clustervolumen

Die Volumen der einzelnen Cluster werden zur Berechnung von ΔQ (Formel 2.1) immer wieder benötigt. Für das Speichern der Volumen wird also eine Lösung benötigt, die trotz der großen Anzahl an Zugriffen (sowohl lesend als auch schreibend) performant ist. In der Arbeit wurden zwei Varianten zur externen Speicherung der Volumen untersucht.

Die erste Variante orientiert sich an der Implementierung ohne externe Daten und speichert die Volumen in einem externem Vektor. Die zweite Variante versucht die Informationen der Clustervolumen in die bereits existierende Datenstruktur für die Kanten mit einzubinden. Im Folgenden wird allerdings gezeigt, dass beide Varianten gravierende Probleme aufwerfen.

Variante 1: Externer Vektor

Die Zugriffsreihenfolge auf die Cluster-Volumen ist nicht geordnet. Jeder Knoten i, der in der Local-Moving-Heuristik betrachtet wird, kann mit beliebigen Clustern verbunden sein. Zwar impliziert die Sortierung der Startknoten auch eine Sortierung der Startcluster, aber eine Sortierung nach Zielknoten oder Zielclustern ist nicht gegeben. Jedes Cluster kann bei der Betrachtung der Knoten beliebig oft und an beliebiger Stelle als Zielcluster auftreten. Es ist folglich nicht möglich, den externen Vektor entsprechend der Zugriffsreihenfolge zu sortieren. Die benötigten Daten können also nicht sequentiell gelesen werden, stattdessen würden teure Einzelzugriffe benötigt. Das gleiche Problem stellt sich auch bei den benötigten Updates der Volumen nach der Verschiebung eines Knotens.

Variante 2: Erweiterung der Datenstruktur für Kanten

Bei einer Verwendung der existierenden Datenstruktur stellt sich ein anderes Problem. Hier wird zwar bereits abgespeichert, in welchem Cluster sich ein Knoten befindet und in welches Cluster eine Kante zeigt, aber zum Speichern der Volumen ist sie nicht geeignet. Bei einer Knotenverschiebung müssten alle Einträge aktualisiert werden, die eines der beiden betroffenen Cluster als Start- oder Zielcluster enthalten. Diese Einträge liegen verstreut in PQ als auch im EXS. Ein Aktualisierung der Daten im EXS wäre zwar denkbar aber teuer, eine Aktualisierung einzelner Einträge in einer PQ hingegen ist gar nicht unterstützt.

Clustervolumen im Hauptspeicher

Das Problem wird durch eine Hash-Map gelöst, welche im internen Speicher gehalten wird und einen schnellen Zugriff für beliebige Cluster ermöglicht. Die Anzahl der benötigten Einträge wird dabei noch durch eine Optimierung vermindert, so dass der benötigte interne Speicher für die Clustervolumen nicht zu stark ins Gewicht fällt.

Optimiert wurde durch die Sonderbehandlung von Singletons, also Clustern, die genau einen Knoten enthalten. Hier gilt, dass das Volumen der Knoten dem ihrer Cluster entspricht. Das Volumen der Knoten wird bereits in den Kanten gespeichert. Im Folgenden wird gezeigt, dass das ausreichend ist und somit keine separaten Einträge für Singletons erforderlich sind.

Zu Beginn einer Berechnung existieren ausschließlich Singletons, jeder Knoten ist seinem eigenen Cluster zugewiesen. Erst wenn sich Knoten verschieben, können Singletons verloren gehen und Cluster-Volumen müssen abgespeichert werden. Bei Verschiebung eines Knotens in das Cluster eines Singletons entsteht ein Cluster aus zwei Knoten. Die Cluster-ID wird zusammen mit seinem neuen Volumen (Volumen des ursprünglichen Singletons plus das Volumen des hinzugekommenen Knotens) in die Hash-Map eingetragen. Wird das Volumen eines Clusters benötigt so wird zunächst in der Hash-Map geschaut, ob für die ID des Clusters ein Eintrag vorhanden ist. Ist dies nicht der Fall, so handelt es sich um ein Cluster mit nur einem Knoten (Singleton).

Einträge von Clustern, die nach Verschiebungen von Knoten wieder zu Singletons werden, sollen auch wieder aus der Hash-Map gelöscht werden. Dazu enthält die Hash-Map neben dem Volumen des Clusters auch die Anzahl der sich darin befindenden Knoten. Fällt dieser Wert auf 1 zurück, so kann der entsprechende Eintrag gelöscht werden. Das Löschen von Einträgen aus der Hash-Map findet allerdings nur in der ersten Local-Moving-Phase einer Berechnung statt, da der Aufwand, den Wert für die Anzahl der Knoten in einem Cluster in der Rekursion aktuell zu halten, nicht mehr mit dem Nutzen korreliert. Denn ein Knoten eines kontrahierten Graphens beinhaltet in der Regel mehrere Knoten des Ursprungsgraphen.

3.2.6 Komplexitätsbetrachtung

Auch bei der voll-externen Variante können nur die Komplexitäten einzelner Teilschritte betrachtet werden. Wie in der semi-externen Variante auch sind das:

- Eine **einzelne Runde** der Local-Moving-Heuristik
- Kontraktion zu einem neuen Graphen
- Rückführen des Ergebnisses auf den Ursprungsgraphen

Local-Moving-Heuristik

Das Zuweisen der Betrachtungsreihenfolge geschieht beim Einfügen der einzelnen Knoten und Kanten in PQ oder EXS, wobei das Hashen einer Knoten-ID auf seinen Betrachtungswert in $\mathcal{O}(1)$ liegt. Das anschließende Sortieren (EXS) benötigt $\mathcal{O}((n+m)\log(n+m))$ Zeit und besitzt die I/O-Komplexität $Sort(n+m)$ aus Kapitel 1.2.1. Im Gegensatz zur semi-externen Variante kann das Sortieren allerdings nicht abgeschaltet werden, da die Sortierung nicht nur wegen der Randomisierung erfolgt.

Die einzelnen Knoten werden analog zur semi-externen Variante betrachtet. Berechnungen und potentielle Verschiebungen liegen pro Knoten in $\mathcal{O}(1)$. Auch das Umdrehen der Kante mit anschließendem Einfügen in EXS liegt in $\mathcal{O}(1)$. Das Einfügen in PQ hat die Ordnung $insert(x)$ aus Kapitel 1.2.1.

Alle Elemente des Kantenspeichers werden also in konstanter Zeit betrachtet. Insgesamt befindet sich jeder Knoten und jede Kante im Kantenspeicher und jede Kante wird maximal einmal umgedreht. Es ergibt sich also eine Zeitkomplexität für die Beabeitung im EXS von $\mathcal{O}(n+m)$ und von $\mathcal{O}(m \log m)$ für die PQ. Auch in der voll-externen Variante ist das Sortieren der Daten in $\mathcal{O}((n+m)\log(n+m))$ somit der dominierende Faktor. Im Gegensatz zur semi-externen Variante kann das Sortieren aber nicht für einzelne Runden abgeschaltet werden, sodass sich für eine einzelne Runde der Local-Moving-Heuristik **immer** eine Laufzeitkomplexität von $\mathcal{O}((n+m)\log(n+m))$ ergibt.

Kontraktion

Bei der Kontraktion wird ein separater EXS genutzt, um die zu kontrahierenden Daten zunächst passend zu sortieren. Der EXS wird zunächst in Zeit $\mathcal{O}(n+m)$ mit allen Knoten und Kanten gefüllt und anschließend in Zeit $\mathcal{O}((n+m)\log(n+m))$ nach Start- und Zielcluster sortiert. Für den EXS gilt die I/O-Komplexität von $Sort(n+m)$ aus Abschnitt 1.2.1.

Beim anschließenden Abarbeiten des EXS werden neue Knoten für die einzelnen Cluster und die Kanten zwischen den Clustern berechnet und in einen EXS eingefügt, welcher für die Rekursion genutzt wird. Die Berechnungen für die $n+m$ Einträge des EXS liegen insgesamt in $\mathcal{O}(n+m)$. Die Kontraktion wird demnach auch davon dominiert, dass Daten auf dem externen Speicher sortiert werden müssen. Hier zeigt sich kein großer Unterschied zu der semi-externen Umsetzung, welche ebenfalls die Sortierung benötigt, allerdings nur für n Kanten und nicht für $n+m$ Objekte. Die interne Implementierung hingegen zeigt hier klare Vorteile, da bei ihr die Sortierung wegfällt.

Abbilden der Rekursionsergebnisse auf den Ursprungsgraphen

Auch beim Zurückführen der Ergebnisse aus der Rekursion ist es in dieser Variante notwendig, die Daten entsprechend zu sortieren. Dazu werden im jeweiligen EXS die Daten des ursprünglichen Graphen nach Zielcluster und die Daten des Kontraktionsgraphen nach Startknoten sortiert. Der Startknoten repräsentiert dabei die ursprüngliche Cluster-ID die bei der Kontraktion gesetzt wurde. Beim Zusammenführen kann nun für jedes Zielcluster des ursprünglichen Graphen geprüft werden, ob der korrespondierende Knoten im Kontraktionsgraphen verschoben wurde. Ist dies der Fall so wird das Zielcluster des ursprünglichen Graphen auf den Wert des Zielclusters des korrespondierenden Knotens gesetzt. Der Aufwand des Zusammenführens liegt damit in $\mathcal{O}(n)$. Demnach stellt auch hier das Sotieren der Daten, mit Zeitkomplexität $\mathcal{O}(n \log n)$ und I/O-Komplexität von $Sort(N)$, erneut den Hauptaufwand dar.

4. Experimente

In diesem Kapitel werden die beiden in der Bachelorarbeit entwickelten externen Varianten der Louvain-Methode mit der ursprünglichen internen Implementierung von Blondel et al. [BGLL08] verglichen. Dazu werden zunächst Vorexperimente beschrieben, die zur Findung der besten Parameter für die beiden neuen Varianten durchgeführt wurden. Anschließend werden Ergebnisse auf Graphen verschiedener Größe und Herkunft dargestellt und ein Vergleich aller drei Varianten geliefert. Es folgt ein detaillierterer Vergleich nur zwischen den beiden externen Varianten um festzustellen, welche davon als sinnvoller erscheint. Abschließend wird noch untersucht, inwieweit Festplatten ein wesentlicher Grund dafür sind, dass die beiden entwickelten externen Varianten signifikant langsamer sind als eine interne Implementierung und ob durch Verwendung von schnelleren Solid-State-Drives ein merklicher Perfomanzgewinn erzielt werden kann.

4.1 Experimentelles Setup

Die Clusterberechnungen der Graphen wurde jeweils auf einem vom Institut für Theoretische Informatik (ITI) gestellten Rechner durchgeführt, der über 32 GB Hauptspeicher, einer Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz sowie zwei 1TB Festplatten mit 7200 RPM verfügt. Das verwendete Betriebssystem war openSUSE 13.2.

Es wurden diverse Scripte geschrieben, um die Testdurchführung größtenteils zu automatisieren. Die Scripte sammeln alle relevanten Ergebnisse der Algorithmen und erfassen zudem alle 0,1 Sekunden den vom jeweilig ausgeführten Programm belegten Hauptspeicher.

Zur Verwaltung der externen Container (Vektoren, EXS und PQs) in den beiden in der Bachelorarbeit entwickelten Varianten wurde die *Standard Template Library for Extra Large Data Sets* (STXXL) [DKS08] verwendet. Sie sorgt für eine möglichst optimale Nutzung der beiden Festplatten und liefert den hier benötigten Funktionsumfang, wobei die bereitgestellte PQ nicht unbedingt für die Art der Nutzung optimiert ist, die in dieser Bachelorarbeit implementiert ist.

Die in den Tests verwendeten Graphen stammen aus der *10th DIMACS Implementation Challenge* [BMSW13], dem *Stanford Network Analysis Project* [LK14] sowie aus dem *WebGraph Framework* [BV04] des *Labratory for Web Algorithmics* der *Universität Mailand*. Die ausgewählten Graphen repräsentieren unterschiedliche Bereiche. Die Auswahl umfasst Webgraphen, die Verlinkungen zwischen einzelnen Webseiten darstellen, Graphen zu sozialen Netzwerken und Graphen zu Straßennetzwerken.

Die Graphen lagen dabei in unterschiedlichen Formaten vor und mussten teilweise konvertiert werden. Eine Auflistung der in den Experimenten verwendeten Graphen findet sich in Tabelle 4.1. Neben der Anzahl von Knoten und Kanten sind zu jedem Graphen Angaben zur Speichergröße auf der Festplatte (bei Speicherung in einem einheitlichen Textformat) enthalten. Sofern bekannt, wurde auch angegeben welche Art von Netzwerk repräsentiert wird. Um mit den Tests eine möglichst große Bandbreite abzudecken, wurden Graphen sehr unterschiedlicher Größe (unter 100 bis 3 Milliarden Kanten) und unterschiedlicher Relationen Knotenzahl : Kantenzahl (ca. 1 : 1,2 bis ca. 1 : 30) gewählt.

Graph	Anzahl Knoten	Anzahl Kanten	Typ	Speicherplatz
Karate	34	78	-	450 Byte
Football	115	613	-	3,8 KByte
Jazz	198	2 742	-	19 KByte
Smallworld	100 000	499 998	-	5,8 MByte
com-Amazon	334 863	925 872	*Produktnetzwerk*	15 MByte
cnr-2000	325 557	2 738 969	*Webgraph*	36 MByte
eu-2005	862 664	16 138 468	*Webgraph*	214 MByte
com-youtube	1 134 890	2 987 624	*Soziales Netzwerk*	46 MByte
in-2004	1 382 908	13 591 473	*Webgraph*	191 MByte
road_central	14 081 816	16 933 413	*Straßennetz*	279 MByte
road_usa	23 947 347	28 854 312	*Straßennetz*	493 MByte
uk-2002	18 520 486	261 787 258	*Webgraph*	4,2 GByte
Twitter-2010	41 652 230	1 202 513 195	*Soziales Netzwerk*	18 GByte
uk-2007	105 896 555	3 301 876 564	*Webgraph*	56 GByte

Tabelle 4.1: Die in den Experimenten verwendeten Graphen. Der jeweils angegeben Speicherplatz soll veranschaulichen um welche Datenmengen es sich handelt.

4.2 Vorexperimente

Die entwickelten Algorithmen besitzen mehrere Parameter, über die das Berechnungsverfahren eingestellt werden kann. So kann zum Beispiel angegeben werden, wie viele Runden der Local-Moving-Phase maximal berechnet werden sollen, bevor der Graph kontrahiert wird. Im Falle der voll-externen Variante kann vorab festgelegt werden, wie viel interner Speicher dem EXS und der PQ zur Verfügung stehen sollen. Bei der semi-externen Variante kann vorgeben werden, wie oft eine Randomisierung erfolgen soll.

Das Ziel der Vorexperimente ist es, mit verschiedenen Parameternsätzen auf einer Teilmenge der Graphen (unter Weglassen der größeren Graphen) die Schnelligkeit der Berechnung und die Qualität der erzielten Modularity zu ermitteln und anhand der Ergebnisse einen Parametersatz festzulegen, mit dem dann alle Graphen durchgerechnet werden. Die Anwendung aller Parametersätze auf alle Beispielgraphen ist wegen der insgesamt benötigten Rechenzeit (auf dem größten Graphen benötigt ein Programmlauf ggf. mehrere Tage) im Rahmen der vorliegenden Arbeit kaum praktikabel.

Sollten bei den finalen Messungen, aufgrund von Problemen mit den in den Vorexperiment bestimmten Parametern, anschließend abweichende Parameter verwendet werden, wird dies explizit angegeben.

4.2.1 Parameteroptimierung für die Local-Moving-Phase

In diesem Vorexperiment gilt es zu klären, ob es notwendig ist, die Berechnung der Local-Moving-Phase stets komplett durchlaufen zu lassen (also bis keine Verbesserung mehr gefunden wird), und in jeder Runde eine neue randomisierte Reihenfolge festzulegen. Dazu wurden verschiedene Testläufe mit der semi-externen Variante gemacht.

Es gibt mehrere Möglichkeiten, die Local-Moving-Phase frühzeitig abzubrechen. Standardmäßig wird das Local-Moving solange durchgeführt, bis keine weitere Verbesserung gefunden wird. Anschließend wird der Graph kontrahiert und die Berechnung wird rekursiv auf dem gröberen Graphen weitergeführt. Bei Graphen mit sehr vielen Knoten und Kanten ist es allerdings auch nach zahlreichen Local-Moving-Runden noch sehr wahrscheinlich, dass weiterhin einzelne Verschiebungen gefunden werden. Diese bringen jedoch nur noch sehr geringen Gewinn bei der Modularity. Die ungünstige Relation von benötigter Zeit pro Runde der Local-Moving-Phase zum erzielten Modularitygewinn legt es nahe, das Local-Moving an geeigneter Stelle frühzeitig abzubrechen.

Es gibt prinzipiell die Möglichkeit die Local-Moving-Phase nach einer gewissen Anzahl an Runden abzubrechen oder nach einer Runde, in der ein vorgegebener Mindestwert für den Modularitygewinn nicht erreicht wurde. In der Bachelorarbeit wurde die einfachere Möglichkeit, die Beschränkung der Rundenzahl, realisiert, da sinnvolle Vorgaben bei der Verwendung des anderen Abbruchkriteriums stärker von der Größe des betrachteten Graphen abhängt.

Zusammen mit der Anzahl an Runden der Local-Moving-Phase wurde auch untersucht, wie sehr die Randomisierung die Berechnung beeinflusst. In Tabelle 4.2 sind durchschnittliche Zeiten für Local-Moving-Runden auf ausgewählten Graphen mit und ohne Randomisierung angegeben. Es handelt sich jeweils um Runden auf dem Ursprungsgraphen (noch nicht kontrahiert). Die Ergebnisse zeigen, dass Runden mit Randomisierung um den Faktor 1,7-3,7 langsamer sind, was im Umkehrschluss bedeutet, dass die Randomisierung ungefähr 40-70% der Laufzeit ausmacht.

Daher wurde untersucht, ob und wie sehr die Qualität der Ergebnisse leidet, wenn man zum Beispiel nur jede vierte oder achte Runde randomisiert. So würde die Randomisierung bei der Laufzeit nur noch zwischen 10% und 35% ausmachen.

Graph	ohne R.	mit R.
com-youtube	0,5s	1,3s
in-2004	3,2s	6,3s
road_usa	7,9s	13,9s
uk-2002	67,6s	116,4s
Twitter-2010	314,5s	560,0s
uk-2007	925,5s	3459,0s

Tabelle 4.2: Laufzeiten einzelner Local-Moving-Runden mit und ohne Randomisierung.

Um die Anzahl der Runden und Häufigkeit der Randomisierung für die finalen Testläufe festzulegen wurden folgende Tests mit der semi-externen Variante durchgeführt:

- **Ursprünglich:** Die Local-Moving-Phase besitzt keine besondere Abbruchbedingung, sie wird ausgeführt, bis keine weitere Verbesserung durch Verschiebung eines Knotens erreicht werden kann. Zudem wird die Betrachtungsreihenfolge der Knoten in jeder Runde randomisiert.
- **32 R1:** In jeder Local-Moving-Phase werden maximal 32 Runden durchlaufen. Anschließend wird der Graph kontrahiert und die Berechnung wird rekursiv weitergeführt. „R1“ besagt, dass nach jeder Runde randomisiert wird.
- **32 R4:** Ebenfalls maximal 32 Runden pro Local-Moving-Phase. Allerdings wird erst nach jeder vierten Runde randomisiert (R4).
- **32 R8:** Maximal 32 Runden pro Local-Moving-Phase mit Randomisierung nach jeder achten Runde.
- **16 R4:** Maximal 16 Runden mit Randomisierung nach jeder vierten Runde.
- **16 R8:** Maximal 16 Runden mit Randomisierung nach jeder achten Runde.

In Tabelle 4.3 sind die in den verschiedenen Testläufen erzielten Modularity-Werte zu den betrachteten Graphen aufgezeigt. Es ist zu sehen, dass die Qualität der Ergebnisse nicht erheblich schwankt, die jeweils besten Ergebnisse aber ziemlich gestreut liegen. Allgemein stellt sich allerdings keine Spalte eindeutig als die beste dar. Die Werte sagen aber offensichtlich aus, dass kein Qualitätsverlust mit den gemachten Einschränkungen verbunden sind. Der fehlende Eintrag bei *uk-2002* ist der Laufzeit zu schulden, die Berechnung wurde nach 13 Stunden abgebrochen, da nach dieser Zeit in der ersten Local-Moving-Phase immer noch neue Verschiebungen gefunden wurden.

Graph	Ursprünglich	32 R1	32 R4	32 R8	16 R4	16 R8
Karate	0,419790	0,419790	0,419790	0,419790	0,419790	0,419790
Football	0,604570	0,604570	0,604570	0,604570	0,604570	0,604570
Jazz	0,445144	0,445144	0,445144	0,445144	0,445144	0,445144
Smallworld	0,793072	0,793072	**0,793083**	0,793074	**0,793083**	0,793074
com-amazon	0,931947	0,931947	**0,932122**	0,931927	**0,932122**	0,931927
cnr-2000	0,912921	**0,912960**	0,912877	0,912948	0,912941	0,912929
eu-2005	0,939834	0,939834	0,939752	**0,940501**	0,940498	0,940470
com-youtube	0,728453	0,728453	**0,728765**	0,728494	0,728701	0,728735
in-2004	**0,980366**	0,980332	0,980323	0,980317	0,980339	0,980331
road_central	0,997548	0,997548	0,997543	0,997551	**0,997552**	0,997549
road_usa	0,998160	0,998160	**0,998163**	0,998160	**0,998163**	0,998158
uk-2002	-	0,990072	0,990087	0,990101	**0,990102**	0,989995

Tabelle 4.3: Erzielte Modularity auf den Graphen mit den verschiedenen Testdurchläufen. Der Maximalwert pro Zeile ist **hervorgehoben**, bei Zeilen ohne Markierung haben alle Varianten den gleichen Wert für die Modularity erzielt.

Für ausgewählte Graphen sind in Abbildung 4.1 zudem die Laufzeiten der verschiedenen Tests dargestellt. Die Liste A.1 im Anhang führt die exakten Ergebnisse aller Graphen auf. Es ist zu sehen, dass wie erwartet die Laufzeiten in der Regel abnehmen, sofern die maximale Rundenanzahl verkleinert und die Randomisierungshäufigkeit verringert wird.

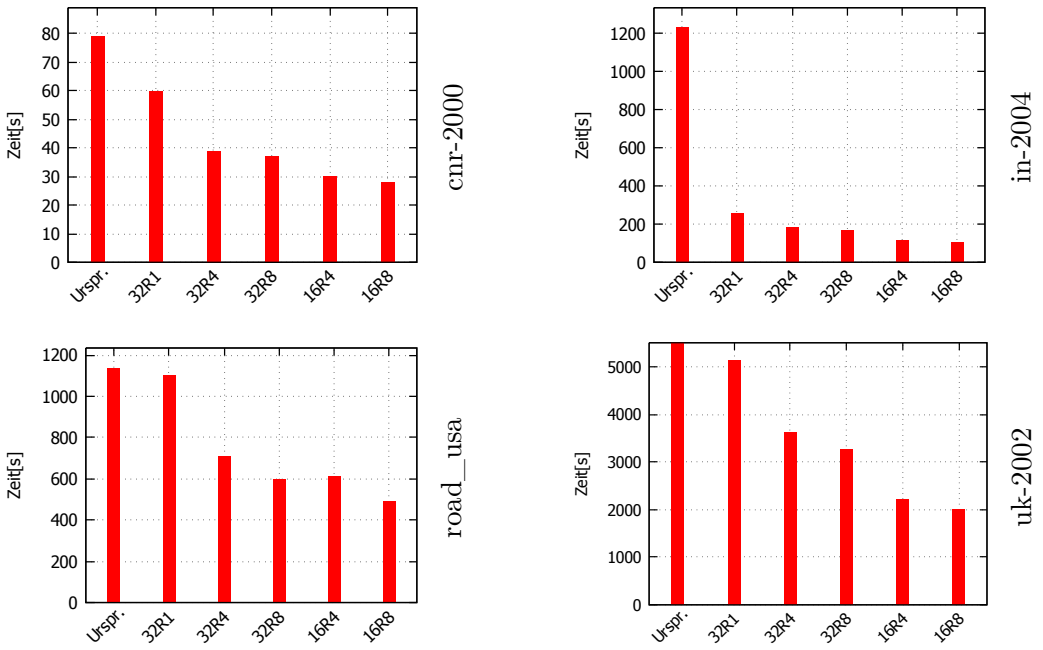


Abbildung 4.1: Laufzeiten der Tests auf ausgewählten Graphen.

Der Laufzeitunterschied ist bei den mittelgroßen bis großen Graphen bereits erheblich, führt aber offenbar zu keinem schwerwiegenden Qualitätsverlust der Berechnung. Es erscheint demnach sinnvoll, die Anzahl der Runden sowie die Häufigkeit der Randomisierung zu beschränken.

Es sind allerdings keine Parameter erkennbar, die durchweg die besten Ergebnisse liefern. Da die in der Bachelorarbeit entwickelten Varianten der Louvain-Methode durch die Nutzung des externen Speichers hohe Laufzeiten besitzen, ist es sinnvoll, die Parameter mit den geringsten Laufzeiten zu wählen. Die Parameterwahl für die folgenden Tests fällt daher auf **16 R 4**, also 16 Runden für die Local-Moving-Phase mit einer Randomisierung nach jeweils 4 Runden.

Es ist anzumerken, dass in der voll-externen Variante lediglich die maximale Anzahl der Runden für die Local-Moving-Phase gesetzt werden kann. Die Häufigkeit der Randomisierung fällt bei dieser Variante so gut wie nicht ins Gewicht (vgl. Abschnitt 3.2.2), da die zeitintensive Sortierung der Knoten und Kanten nach jeder Runde auch ohne Randomisierung stattfinden muss.

4.2.2 Speicherzuweisung für die voll-externe Variante

Die voll-externe Variante lagert zwar die meisten anfallenden Daten auf externe Speichermedien aus, aber um die einzelnen Dateneinträge zu verarbeiten, müssen diese in den internen Speicher geladen werden. Aus diesem Grund haben sowohl die verwendeten externen Sorter als auch die Prioritätswarteschlangen einen Parameter, der angibt, wie viel Hauptspeicher ihnen zugewiesen ist. Dieser Wert muss vor der Berechnung angegeben werden und ist anschließend nicht mehr variierbar. Die Erwartung ist, dass die Verarbeitung umso schneller ist, je mehr Hauptspeicher dem EXS und der PQ zugewiesen werden. Auch hier wurde eine Testreihe durchgeführt um diese Annahme zu überprüfen und sich auf einen Wert für die folgenden Experimente festzulegen.

In der Testreihe wurde den externen Sortern und Prioritätswarteschlangen jeweils 32 MB, 64 MB, 128 MB, 256 MB, 512 MB, 1 GB oder 2 GB an Hauptspeicher gewährt. Anschließend wurden drei Runden des Local-Movings durchgeführt und der Mittelwert über die benötigte Zeit pro Runde gebildet. In Abbildung 4.2 sind diese Zeiten dargestellt. In Liste A.2 im Anhang befinden sich zudem die Werte für alle Graphen.

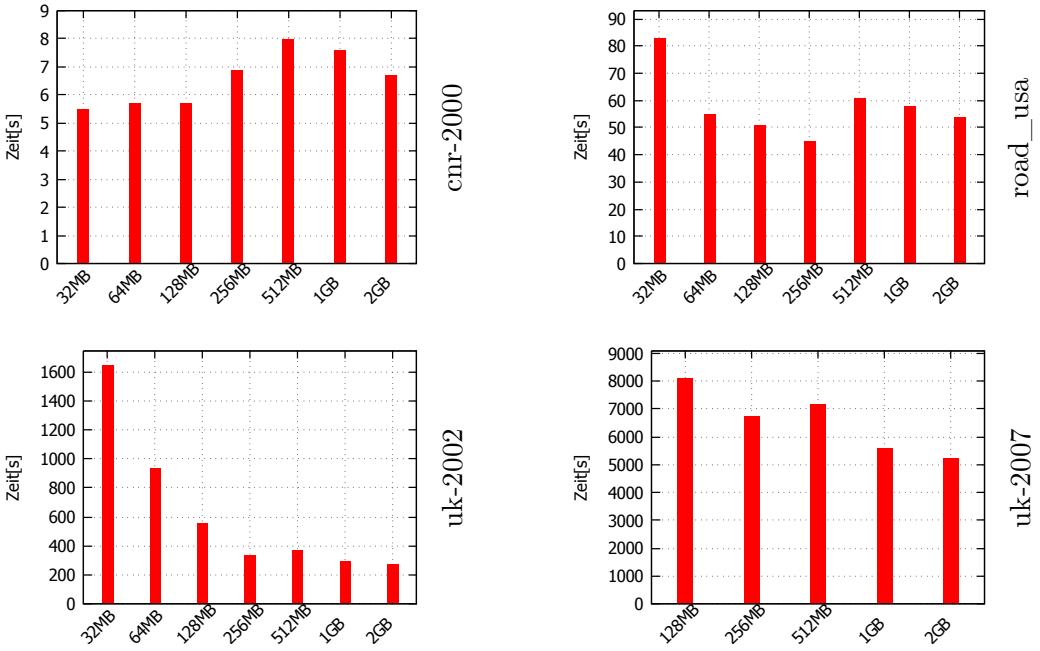


Abbildung 4.2: Laufzeiten einer **einzelnen** Local-Moving-Runde der voll-externen Variante mit verschiedenen Speicherzuweisungen auf ausgewählten Graphen.

Es ist zu sehen, dass bei kleinen bis mittelgroßen Graphen die Speicherzuweisung keinen großen Einfluss auf die Rechenzeit einzelner Runden hat, da selbst 32 MB oder 64 MB ausreichen, damit Sorter und Prioritätswarteschlange alle Daten im internen Speicher halten können. Bei manchen dieser Graphen, wie zum Beispiel dem *cnr-2000*-Graphen, erhöht sich sogar die Laufzeit zunächst. Eine mögliche Erklärung ist hier, dass mit zunehmenden Speicher der von der eigentlichen Bearbeitung unabhängige Verwaltungsaufwand (z.B. Initialisierungen) dominiert. Das Absinken der Zeiten bei 1 GB oder 2 GB ist nicht ganz geklärt. Die gesamte Schwankung bei diesem Graphen beträgt allerdings auch nur wenige Sekunden.

Ein anderes Phänomen ist dagegen wohl nur mit Besonderheiten bei der internen Implementierung des EXS und/oder der PQ zu erklären. Bei allen Messungen zeigt sich beim Übergang von 256 MB zu 512 MB eine geringfügige Verschlechterung.

Bei größeren Graphen spiegeln die Messungen im Wesentlichen die Erwartungen wieder. Beim *uk-2002* kann die Rechenzeit von über 26 Minuten mit 32 MB Hauptspeicher pro Container auf 4,5 Minuten mit 2 GB Hauptspeicher gedrückt werden. Beim noch größeren *uk-2007* ergibt sich ebenfalls eine enorme Zeitersparnis von knapp 48 Minuten oder ca. 35% zwischen der Durchführung mit 128 MB und 2 GB. Hier waren die Tests mit 32 MB und 64 MB aufgrund der großen Laufzeiten nicht sinnvoll, da das Programm bereits mit 128 MB über 2 Stunden lief.

Insgesamt ergibt sich folgendes Bild: Je größer die zu berechnenden Graphen sind, desto sinnvoller ist es, den Sortern und Prioritätswarteschlangen mehr Hauptspeicher zu geben. Damit im folgenden nicht alle Experimente mit verschiedenen Werten durchgeführt werden müssen, soll ein gemeinsamer Wert für alle Testläufe mit der voll-externen Variante festgelegt

werden. Zunächst scheint die Wahl von 2GB sinnvoll, bei den späteren Experimenten stellte sich allerdings heraus, dass durch die Rekursion bei einer Wahl von 1GB oder 2GB der interne Speicher frühzeitig ausgeschöpft wurde und die Berechnung nicht zu Ende geführt werden konnte. Weil der Wert von 512 MB bei den erfolgten Messungen eine Anomalität gezeigt hat, wurde schließlich **256 MB** pro Sorter und Prioritätswarteschlange zugewiesen.

4.3 Vergleich der drei Varianten

In den folgenden Experimenten wurden alle drei Varianten der Louvain-Methode auf den verschiedenen Graphen aus Tabelle 4.1 ausgeführt. Dabei wurden Messungen zur benötigten Laufzeit sowie des benötigten Hauptspeichers gemacht und die auf einem Graphen jeweils erreichte Modularity abgespeichert. Zum Hauptspeicherverbrauch wurden zudem noch theoretische Betrachtungen des Speicherbedarfs angestellt.

4.3.1 Modularity

Die Ergebnisse zu den erzielten Modularity-Werten befinden sich in Tabelle 4.4. Es ist zu sehen, dass zwischen den Varianten keine zu großen Unterschiede vorhanden sind. Alle Varianten verfahren nach dem gleichen Algorithmus, bei den in der Bachelorarbeit entwickelten Varianten kommt allerdings noch das Multilevel-Refinement hinzu, welches jeweils noch einen kleinen Gewinn an Modularity bringt. Die tatsächlich erreichte Qualität der erzielten Ergebnisse hängt zudem von der Randomisierung ab. Dadurch sind kleine Schwankungen der Modularity möglich.

Graph	intern	semi-extern	voll-extern
Karate	0,418803	0,419790	0,419790
Football	0,604570	0,604570	0,604407
Jazz	0,439804	0,445144	0,445027
Smallworld	0,792970	0,793083	0,793014
com-amazon	0,925966	0,932122	0,931636
cnr-2000	0,912745	0,912941	0,912947
eu-2005	0,938726	0,940498	0,940053
com-youtube	0,723519	0,728701	0,728522
in-2004	-	0,980339	0,980346
road_central	0,997380	0,997552	0,997365
road_usa	0,998031	0,998163	0,998027
uk-2002	-	0,990102	0,990098
Twitter-2010	zu groß	0,491438	0,487544
uk-2007	zu groß	0,996309	0,996318

Tabelle 4.4: Auflistung der mit den verschiedenen Varianten erzielten Modularity-Werte. Die Messwerte der internen Variante für *in-2004* und *uk-2002* fehlen aufgrund von Programmabstürzen.

Aus Zeitgründen wurden keine mehrfachen Durchläufe mit unterschiedlichen Seeds durchgeführt um diese Schwankungen weiter zu eliminieren. Die geringen Unterschiede zwischen berechneten Modularitys lassen dafür aber kaum eine Notwendigeit erkennen.

Insgesamt lässt sich festhalten, dass alle Varianten ähnlich gute Clusterungen auf den getesteten Graphen erzielen. Wie nahe die Werte an die maximal erzielbaren Modularitys bei einem theoretischen Verfahren mit voller Kombinatorik über alle Clustermöglichkeiten heranreicht, kann nicht gesagt werden (zumindest nicht bei den größeren Graphen). Da der

maximal erreichbare Wert aber fast 1 ist, deuten die Ergebnisse einiger großen Graphen mit Werten über $0,99$ aber daraufhin, dass der Abstand zum bestmöglichen Wert sehr klein ist.

4.3.2 Laufzeiten

Bei den Laufzeiten der verschiedenen Varianten lassen sich, wie erwartet, erhebliche Unterschiede feststellen. Die Messergebnisse dazu befinden sich in Tabelle 4.5. Alle Messungen wurden erst gestartet, nachdem die Graphen in das Programm eingelesen waren. Die benötigte Zeit zum Einlesen der Graphen hängt zu sehr vom Format ab, in dem die Graphen vorliegen.

Graph	intern	semi-extern	voll-extern
Karate	<1s	<1s	<1s
Football	<1s	1s	<1s
Jazz	<1s	<1s	<1s
Smallworld	1s	29s	46s
com-amazon	1s	33s	137s
cnr-2000	4s	30s	165s
eu-2005	11s	148s	439s
com-youtube	6s	38s	201s
in-2004	-	118s	429s
road_central	116s	335s	1910s
road_usa	201s	615s	3016s
uk-2002	-	2213s	9243s
Twitter-2010	zu groß	13871s	88 829s
uk-2007	zu groß	46 473s	190 480s

Tabelle 4.5: Laufzeiten der verschiedenen Varianten auf den Graphen in Sekunden.

Es ist zu sehen, dass die beiden externen Varianten auf den kleineren Graphen (*Karate* bis *Jazz*) absolut kaum länger benötigen als die interne Variante. Erst wenn die Graphen größer und die Festplatten durch EXS und PQs wirklich genutzt werden, benötigen die externen Varianten deutlich mehr Zeit.

Sowohl beim Vergleich intern $\leftrightarrow$ semi-extern als auch bei intern $\leftrightarrow$ voll-extern ergibt sich eine starke Streuung der Laufzeitfaktoren. Die semi-externe Variante benötigt das 3-fache bis 30-fache der internen Variante, die voll-externe Variante das 15-fache bis über 100-fache. Betrachtet man die später in Abbildung 4.5 und 4.6 dargestellte Verteilung der Gesamtlaufzeit pro Graph, scheint sich herauszustellen, dass die Faktoren dann besonders hoch sind, wenn die in der Rekursion entstehenden kontrahierten Graphen noch verhältnismäßig groß und die Local-Moving-Phasen während der Rekursion dementsprechend teurer sind. Details dazu folgen in Kapitel 4.4. Das deutet darauf hin, dass sich insbesondere die Performanz der Local-Moving-Phasen durch Verwendung des externen Speichers verschlechtert. Diese Aussage ist allerdings unter dem Vorbehalt zu sehen, dass die vorliegenden Vergleichsdaten unvollständig sind, weil insbesondere für die größten Graphen keine Tests mit der internen Variante möglich waren, da dafür 32 GB Hauptspeicher nicht ausreichend waren.

4.3.3 Hauptspeicherverbrauch

Im nachfolgenden Abschnitt wird untersucht, wie viel internern Speicher die verschiedenen Varianten der Louvain-Methode benötigen. Dazu wird zunächst theoretisch betrachtet, wie viel Hauptspeicher die verschiedenen Implementierungen aufgrund der verwendeten

Datenstrukturen benötigen sollten. Hierbei wird sich auf die Betrachtung der Local-Moving-Phase auf dem Ursprungsgraphen beschränkt, da eine Gesamtbetrachtung aufgrund der nicht vorhersehbaren Rekursionstiefe nicht möglich ist. Bei der späteren Messung des tatsächlichen Speicherverbrauchs ist die gesamte Programmausführung und damit auch die Rekursion berücksichtigt.

Theoretische Betrachtung

In diesem Abschnitt wird grob berechnet, wie viel Platz die verschiedenen Varianten für die wesentlichen Daten der ersten Local-Moving-Phase im internen Speicher benötigen. Hierzu wird für jede Variante eine Formel angegeben, die den Platzbedarf der für die Berechnung wichtigsten Objekte in Abhängigkeit von der Knotenanzahl n und der Kantenanzahl m eines Graphen berechnet. Der ermittelte Wert gibt allerdings nur eine Orientierung für den Platzbedarf innerhalb der ersten Local-Moving-Phase auf dem Originalgraphen. In jeder Rekursionsstufe tritt weiterer entsprechender Platzbedarf auf, allerdings mit kleiner werdenden Werten für n und m.

In der ersten Tabelle 4.6 befinden sich eine Auflistungen der wichtigsten Daten, die Hauptspeicher bei der internen Implementierung der Louvain-Methode belegen. Addiert man die genannten Werte auf, so ergibt sich für einen Graphen mit n Knoten und m Kanten ein Speicherbedarf von $256 \cdot n + 192 \cdot m$ Bit oder $32 \cdot n + 24 \cdot m$ Byte.

Bezeichnung	Speicher pro Eintrag	Anzahl Einträge
Knoten	32 Bit	n
Kanten	$2 \cdot (32 + 64)$ Bit	m
Cluster	$3 \cdot 32$ Bit	n
Zusätzl.-Werte	$(32 + 64)$ Bit	n
Zufallswerte	32 Bit	n

Tabelle 4.6: Auflistung wesentlicher Komponenten des internen Speicherverbrauchs der internen Louvain-Methode.

In Tabelle 4.7 befindet sich eine entsprechende Auflistung für die semi-externe Variante. Es ist zu sehen, dass die Anzahl der verschiedenen Einträge hier jeweils nur von der Knotenanzahl n abhängt. Dadurch ergibt sich insgesamt ein Speicherbedarf von lediglich $320 \cdot n$ Bit oder $40 \cdot n$ Byte. Hinzu kommt der Hauptspeicher der für den verwendeten externen Sorter anfällt. Dieser Wert ist allerdings nicht fix und kann im Vorhinein nicht genau beziffert werden.

Bezeichnung	Speicher pro Eintrag	Anzahl Einträge
Knoten	64 Bit	n
Cluster	$(32 + 64)$ Bit	n
Knotengew.	32 Bit	n
k_{i_in}*-Werte*	32 Bit	n
Zufall	32 Bit	n
zusätzlich	64 Bit	n

Tabelle 4.7: Auflistung wesentlicher Komponenten des internen Speicherverbrauchs der semi-externen Variante.

Bei der voll-externen Variante ist keine Tabelle zur Darstellung des internen Speicherverbrauchs erforderlich. Hier wird lediglich die Hash-Map der Cluster-Volumen im Hauptspeicher gehalten. Jeder Eintrag benötigt dabei 64 Bit. Da jeder Knoten zunächst sein eigenes

Cluster darstellt und die Volumen dieser Singletons nicht explizit abgespeichert werden muss, ergibt sich, dass die Hash-Map nie mehr als $\frac{n}{2}$ Einträge enthält. Der Platzbedarf ist demnach $64 \cdot \frac{n}{2}$ Bit oder $8 \cdot \frac{n}{2}$ Byte.

In Abbildung 4.3 ist für ausgewählte Graphen der jeweils theoretisch benötigte Hauptspeicher für die einzelnen Varianten visualisiert. In Tabelle A.3 finden sich die entsprechenden Werte für alle in der Bachelorarbeit betrachteten Graphen. Es handelt sich jeweils nur um den Speicherbedarf in der ersten Local-Moving-Phase, der zusätzlich benötigte Speicher für externe Sorter und Prioritätswarteschlangen sowie der Hauptspeicherbedarf der durch die Rekursionen entsteht wurde nicht beachtet.

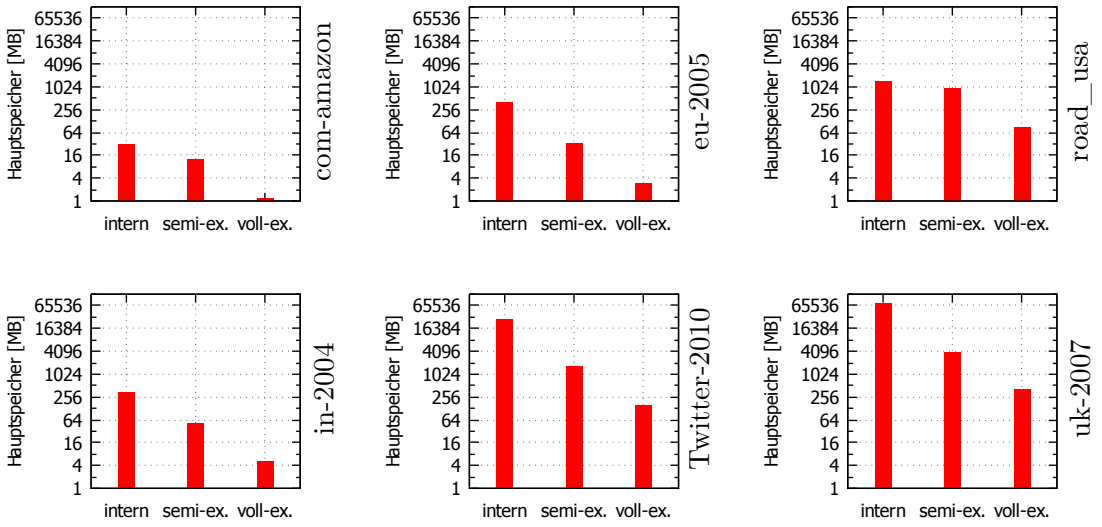


Abbildung 4.3: Theoretisch benötigter Hauptspeicher der ersten Local-Moving-Phase der verschiedenen Varianten auf ausgewählten Graphen. Die y-Achse ist jeweils **logarithmisch skaliert**.

Bei Betrachtung der verschiedenen Graphen kann bereits beim Vergleich der semi-externen und der internen Variante ein sehr großer Unterschied festgestellt werden. Je nach Graph benötigt die interne Implementierung zwischen 1,5-mal (vgl. *road_usa*) und 20-mal so viel Speicher (vgl. *uk-2007*). Der genaue Faktor hängt vom Verhältnis Kantenzahl : Knotenanzahl im Graphen ab. Jedoch bringt allein die Auslagerung der Kantendaten bereits bei den meisten Graphen eine erhebliche Minderung des theoretisch benötigten Hauptspeichers. Beim Schritt von semi- auf voll-externer Variante spart man erneut einen Faktor zehn an Hauptspeicher, was bedeutet, dass die voll-externe Variante **theoretisch** nur zwischen einem $\frac{1}{30}$ und $\frac{1}{200}$ des Hauptspeichers der internen Implementierung benötigt. Dass es sich hierbei nur um eine theoretische Betrachtung handelt wird bei den folgenden tatsächlichen Messergebnissen ersichtlich.

Messungen des Speicherverbrauchs

Bei der Durchführung der verschiedenen Experimente wurde jeweils alle 0,1 Sekunden der aktuelle Hauptspeicherverbrauch des ausgeführten Programms gemessen, um so einen Maximalwert zu ermitteln. Die interne Variante konnte hierbei leider nicht auf jedem Graphen ausgeführt werden. Wie in Abbildung 4.3 zu sehen ist benötigt sie für den *uk-2007*-Graphen bereits aufgrund der theoretischen Betrachtung mindestens 77GB Hauptspeicher, im Testsystem stehen aber nur 32 GB zur Verfügung.

Außerdem hatte die Implementierung von Blondel et al. [BGLL08] teilweise Probleme mit anderen Graphdaten und stürzte des Öfteren ab (so zum Beispiel auf dem *in-2004*-Graphen und *uk-2002*-Graphen).

In Abbildung 4.4 sind Messergebnisse für eine Auswahl von Graphen dargestellt. Die Liste A.4 im Anhang beinhaltet wieder alle verfügbaren Messwerte. Es ist zu sehen, dass für die kleineren und mittelgroßen Graphen die tatsächlichen Messwerte stark von den Werten der theoretischen Betrachtungen abweichen. Die Begründung, externe Varianten zu verwenden um Hauptspeicher zu sparen, wäre bei alleiniger Betrachtung dieser Graphen ad absurdum geführt. Erst bei einem der größten dargestellten Graphen (*Twitter-2010*), der bei Betrachtung des unkomprimierten Speicherplatzes (Tabelle 4.1) ca. 36 mal so groß ist wie der nächstkleinere (*road_usa*) ergibt sich bei der Abnahme des Hauptspeicherverbrauchs annähernd das erwartete Bild. Auch zeigt sich erst beim größten betrachteten Graphen (*uk-2007*) ein geringere Speicherverbrauch der voll-externen Variante gegenüber der semi-externen. Dabei wird sogar minimal weniger Hauptspeicher benötigt als beim *Twitter-2010*-Graphen, was allerdings nur an der Rekursionstiefe sowie den verwendeten PQs und EXs liegt.

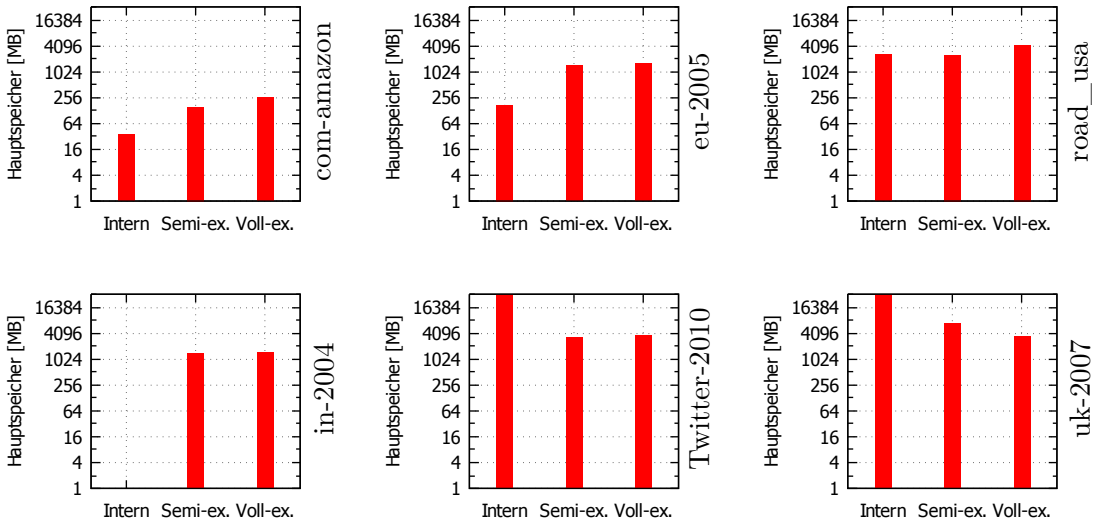


Abbildung 4.4: Tatsächlich gemessener Hauptspeicherbedarf der verschiedenen Varianten auf ausgewählten Graphen. Die y-Achse ist jeweils **logarithmisch skaliert**.

Twitter-2010 und alle größeren Graphen konnten nicht mehr mit der internen Implementierung geclustert werden. Formal wurde deshalb in der graphischen Darstellung der maximal Wert des Darstellungsbereichs gewählt.

Bei der voll-externen Variante wurden zusätzlich Messungen durchgeführt zu der Anzahl der Einträge die die Hash-Map bei den verschiedenen Graphen maximal hat. Diese Anzahl repräsentiert auch die maximale Anzahl an Clustern die gleichzeitig existierten, ohne die Singletons. Im theoretischen Teil wurde der maximale Füllstand bereits mit $\frac{n}{2}$ angegeben. In Tabelle 4.8 finden sich zu einer Auswahl an Graphen die tatsächlich maximale Anzahl an Einträgen gegenüber der Anzahl an Knoten. Es ist zu sehen, dass in den meisten Fällen weitaus weniger Einträge für die Hash-Map im internen Speicher benötigt werden als maximal möglich. Nur auf Graphen, bei denen ungefähr so viele Knoten wie Kanten existieren (z.B. *road_usa*) wird ansatzweise das Maximum ausgereizt. Demnach fällt der hierfür benötigte Hauptspeicher in der Regel deutlich kleiner aus als der berechneten Maximalwert.

Folgende Erkenntnisse lassen sich nach den Messungen festhalten:

- Semi- und voll-externe Variante erlangen zwar einen geringeren Hauptspeicherverbrauch gegenüber der internen Implementierung, aber erst auf sehr großen Graphen.
- Die theoretischen Werte bei der voll-externen Variante sind zumindest in dem Bereich, der durch Messungen abgedeckt wurde weitgehend wertlos, die errechneten Zahlen

Graph	n	max. Anzahl Einträge	Anteil an n
com-amazon	334 863	92 321	27,57%
in-2004	1 382 908	201 250	14,55%
road_usa	23 947 347	10 331 269	43,14%
Twitter-2010	41 652 230	6 034 082	14,49%
uk-2007	105 896 555	9 260 720	8,75%

Tabelle 4.8: Maximale Anzahl der Einträge innerhalb der Hash-Map bei Ausführung der voll-externen Variante auf ausgewählten Graphen.

sind bereits so niedrig (z.B. < 300 MB bei *Twitter-2010*), dass sich hier eine klare Dominanz von EXS und PQs ergibt. Zudem gilt nach Betrachtung der maximalen Einträge innerhalb der Hash-Map, dass die theoretischen Werte für die Hash-Map meist sogar zu hoch angesetzt sind.

- Die Container (PQs und EXS) können anscheinend einen zusätzlichen Hauptspeicherbedarf von insgesamt mehr als 2 GB erzeugen (z.B. bei *road_usa* steigt der belegte Hauptspeicher beim Übergang von semi-extern zu voll-extern um 1,8 GB an, obwohl er theoretisch zurückgehen müsste). Je größer die Rekursionstiefe, desto mehr Speicher wird durch die Container belegt. Eine Optimierung bezüglich der Speichernutzung wäre hier durchaus angebracht.

4.4 Weitere Laufzeitanalysen und Betrachtung der I/Os

Als Ergebnisse der Experimente wurde in den vorangegangenen Abschnitten die absoluten Werte für Modularity, Laufzeit und Speicherverbrauch dargestellt. In diesem Kapitel folgt eine weitergehende Analyse der Laufzeiten der externen Varianten hinsichtlich verschiedener Programmphasen und im Zusammenhang mit I/O-Statistiken. Unter Verwendung dieser Statistiken werden schließlich noch Betrachtungen zu möglichen Performanzgewinnen durch Austausch des externen Speichermediums gemacht.

4.4.1 Laufzeitanteile verschiedener Programmphasen

In den Abbildungen 4.5 und 4.6 sind die Anteile der wesentlichen Programmphasen an der Gesamtlaufzeit dargestellt. Dabei wurde jeweils in die vier Phasen: Erste Local-Moving-Phase, Kontraktion, Rekursion und Refinement aufgeteilt. Bei der voll-externen Variante gibt es als weitere Phase das Zurückführen der Ergebnisse. Diese Phase beansprucht in der semi-externen Variante zu wenig Zeit um sie dort ebenfalls zu berücksichtigen. Es ist anzumerken, dass sich Kontraktion, Zurückführen der Ergebnisse und Refinement nur auf die Berechnungen des Ursprungsgraphen beziehen. Die Rekursions-Phase beinhaltet alle weiteren Local-Moving-Phasen, Kontrahierungen, Zurückführeungen und Refinements innerhalb der Rekursion.

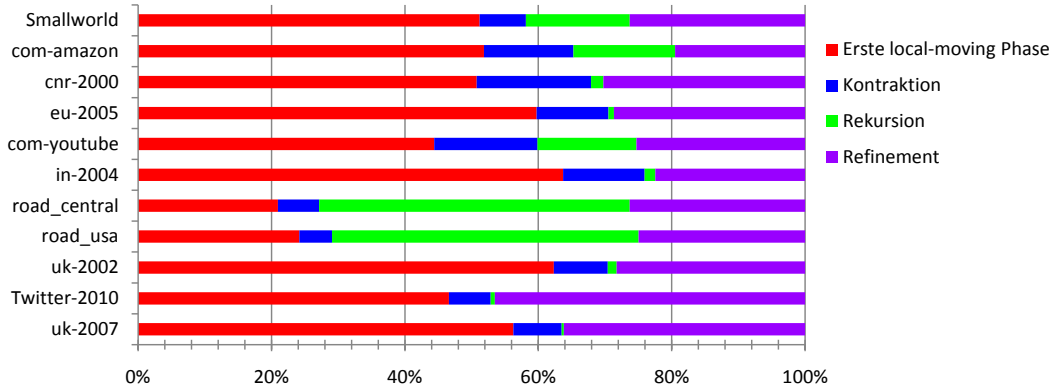


Abbildung 4.5: Prozentuale Anteile der verschiedenen Phasen an der Gesamtlaufzeit der **semi-externen Variante** für ausgewählte Graphen.

Vergleicht man die Anteile der Phasen für einen Graphen bei beiden Varianten, so zeigt sich, abgesehen von der Rückführung, dass die Verteilung der gesamten Laufzeit ähnlich ist. Dieses Verhalten war durchaus auch zu erwarten, da beide Algorithmen ja nach dem gleichen Prinzip funktionieren.

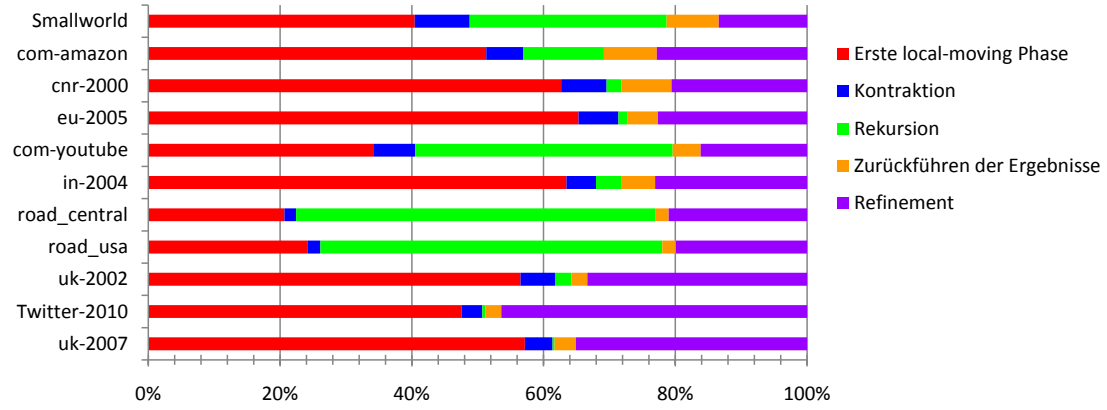


Abbildung 4.6: Prozentuale Anteile der verschiedenen Phasen an der Gesamtlaufzeit der **voll-externen Variante** für ausgewählte Graphen.

Ein Vergleich zwischen den verwendeten Graphen zeigt jedoch deutliche Unterschiede. So ist der relative Anteil der Rekursion bei *road_central* und *road_usa* auffällig hoch, bei *com-youtube*, *com-amazon* und *Smallworld* immer noch deutlich höher als bei den anderen Graphen. *Twitter-2010* und *uk-2007* stehen in dieser Rangliste ganz am Ende.

Die Ursache ist in der Relation von Kantenanzahl zu Knotenanzahl zu suchen. Bei *road_central* und *road_usa* besteht ein Faktor von $1,1 - 1,2 : 1$, bei *com-youtube*, *com-amazon* und *Smallworld* etwa 2,5-5 : 1 und bei *Twitter-2010* und *uk-2007* ca. 30 : 1.

Für die beiden Straßengraphen bedeutet das, dass jeder Knoten im Schnitt nur etwas mehr als zwei Nachbarn hat. Damit existieren für die erste Local-Moving-Phase nur relativ wenig Verschiebungsmöglichkeiten. Als Zwischenergebnis entstehen dadurch eher viele kleinere Cluster und damit insgesamt eine hohe Anzahl an Clustern. Das bedeutet wiederum, dass der kontrahierte Graph in der erste Rekursion noch verhältnismäßig viele Knoten hat. Auch das Verhältnis zwischen Kanten und Knoten bleibt in der ersten Rekursion so noch relativ klein.

Insgesamt ergibt sich, dass sich die Größe der Graphen (gemessen an der Knotenanzahl) in der Rekursion umso langsamer verringert, je geringer das Verhältnis von Kanten zu Knoten

ist. Im Umkehrschluss folgt daraus, dass die Local-Moving-Phase während der Rekursion dann einen größeren Anteil an der Gesamtzeit beansprucht.

In Tabelle 4.9 lassen sich die unterschiedlich schnellen Abnahmen der Anzahl an Clustern für die beiden Graphen *road_usa* und *uk-2007* im Verlauf der Berechnung gut nachvollziehen.

Das Kontrahieren des Graphen beansprucht bei der semi-externen Variante einen größeren Zeitanteil als bei der voll-externen Variante. Insgesamt verläuft die Kontraktion auf der semi-externen Variante dennoch ca. zwei- bis dreimal so schnell wie auf der voll-externen. Der größere Zeitanteil kommt daher, dass die semi-externe Variante insgesamt schneller ist.

Die Rückführung der Ergebnisse aus der Rekursion auf den Ursprungsgraphen kostet bei der semi-externen Variante nicht viel Zeit, da nur der Vektor mit den Clusterzuweisungen im Hauptspeicher einmal aktualisiert werden muss. Bei der voll-externen Variante hingegen sind dafür aufwändige Sortierungen erforderlich. Der Aufwand liegt dabei zwischen 2% und 10% des Gesamtaufwands.

Level	Runde	*uk-2007* Anzahl Cluster	*road_usa* Anzahl Cluster
0	0	105 896 555	23 947 347
	1	13 229 496	10 932 573
	2	3 701 025	10 350 464
	3	2 792 419	10 278 207
	4	2 610 911	10 266 631
	5	2 536 312	10 264 023
	⋮	⋮	⋮
	16	2 388 375	10 262 910
1	16	897 769	4 003 608
2	11 | 16	771 697	1 408 424
3	6 | 16	763 631	447 954
4	4 | 16	762 994	125 737
5	3 | 12	762 964	30 629
6	2 | 9	762 963	6 692
7	1 | 8	762 963	2 005
8	5	-	1 560
9	2	-	1 550
10	1	-	1 550
9	Refinement	-	1 550
8	Refinement	-	1 510
7	Refinement	-	1 486
6	Refinement	762 963	1 486
5	Refinement	762 697	1 486
4	Refinement	762 593	1 486
3	Refinement	762 397	1 486
2	Refinement	762 332	1 486
1	Refinement	762 324	1 486
0	Refinement	762 320	1 486

Tabelle 4.9: Anzahl der Cluster von *uk-2007* und *road_usa* zu verschiedenen Zeiten. Die Daten stammen von der voll-externen Variante, sind aber auch für die semi-externe Variante repräsentativ. Sind zwei Rundenzahlen angegeben, so bezieht sich die erste Zahl auf die maximal erreichte Runde innerhalb dieses Levels für *uk-2007* und die zweite Zahl für *road_usa*.

4.4.2 I/O-Statistiken

Bei den Experimenten der semi- und voll-externen Variante wurden zusätzlich *I/O-Statistiken* erstellt. Darin finden sich hilfreiche Informationen zu den Zugriffen auf den externen Speicher. Da die gesamte Aufführung der Daten zu umfangreich ist, wird sich auf die folgenden Messerwerte innerhalb der *I/O-Statistiken* beschränkt:

- Anzahl Lesezugriffe
- Anzahl Schreibzugriffe
- Anzahl von der Festplatte gelesener Bytes
- Anzahl auf die Festplatte geschriebener Bytes
- I/O Wartezeit

Diese Daten werden im Folgenden nur für zwei Graphen komplett aufgezeigt, da durch eine Aufzählung aller Werte für alle Graphen keine zusätzliche Erkenntnisse gewonnen werden. Die aufgeführten Graphen sind *eu-2005* und *Twitter-2010*. Die Messwerte befinden sich in Tabelle 4.10

	eu-2005		*Twitter-2010*	
Messwert	semi-extern	voll-extern	semi-extern	voll-extern
Lesezugriffe	10 113	16 111	961 182	4 376 759
Schreibzugriffe	3 161	18 316	314 547	4 390 076
Bytes (gelesen)	19,7 GiB	24,3 GiB	1,8 TiB	5,9 TiB
Bytes (geschrieben)	6,2 GiB	26,4 GiB	614,4 GiB	5,9 TiB
I/O Wartezeit	106s	219s	6 653s	49 644s

Tabelle 4.10: Messwerte der I/O-Statistiken von semi- und voll-externe Variante bei ausführung auf *eu-2005* und *Twitter-2010*.

Bei den Messwerten für den *eu-2005*-Graphen ist zu sehen, dass die voll-externe Variante eine klar höhere Anzahl an Lese- und Schreibzugriffen aufweist. Vor allem die Anzahl der Schreibzugriffe hat sich aufgrund der aufwändigeren Sortierung deutlich erhöht. Der Vergleich der gelesenen und geschriebenen Datenmengen zeigt diesen Unterschied ebenfalls. Die voll-externe Variante hat hier mehr als vier mal so viele Daten auf die Festplatte geschrieben, was auch zu einem entsprechend höheren Zeitverbrauch führt. Die I/O Wartezeit gibt Aufschluss über die kumulierte Wartezeit, die benötigt wird, bis Daten von der Festplatte im Hauptspeicher zugreifbar sind.

Für den *Twitter-2010*-Graphen zeigt sich ein ähnliches Bild. Allerdings wird hier ersichtlich, welche Dimensionen die Werte für große Graphen annehmen. Die Differenz der Werte zwischen semi- und voll-externer Variante wird hier noch einmal deutlicher. Anhand des *Twitter-2010*-Graphen zeigt sich, dass die Anzahl der Zugriffe deutlich voneinander abweicht. Die Datenmengen die dabei insgesamt geschrieben und gelesen werden liegen im Bereich von Tebibyte (TiB). Die Festplatten an sich besitzen gerade einmal 0,9 TiB. Leider liegen keine Messwerte dazu vor, wie voll die Festplatten zu einem Zeitpunkt maximal beschrieben waren. Zusammen mit den anderen Werten steigt auch die *I/O Wartezeit*. Bei Ausführung der voll-externen Variante wurde somit 13h 47min allein dafür verwendet um auf die Zugriffe der Festplatte zu warten.

4.4.3 Festplatten als Flaschenhals

Mit dem letzten Experiment sollte geklärt werden, wie sehr die Festplatten die beiden externen Varianten verlangsamen. Wie im vorhergehenden Abschnitt bereits zu sehen, fallen erhebliche Zeiten für das Warten auf die I/O-Operationen der Festplatte an. Rechnet man diese Werte aus den Gesamtlaufzeiten der verschiedenen Varianten heraus, so ergeben sich theoretische Laufzeiteinsparungen von ca. 30% bis 70%. In Abbildung 4.7 sind diesbezüglich für weitere ausgewählte Graphen die jeweilgen Anteile der I/O Wartezeiten an der Gesamtlaufzeit für semi- und voll-externe Variante dargestellt.

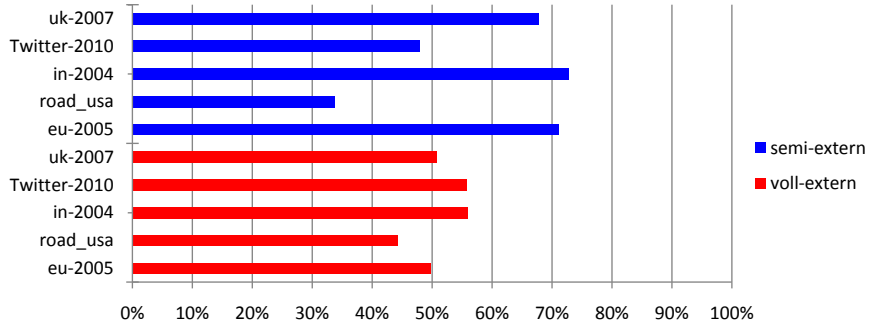


Abbildung 4.7: Prozentuale Anteile der I/O Wartezeit auf ausgewählten Graphen für die Berechnung mit semi- und voll-externer Variante.

Im Folgenden wird in Experimenten untersucht, welche genauen Laufzeiten erzielt werden, wenn die Festplatten die Berechnungen nicht verlangsamen. Die verwendete STXXL-Library ermöglicht es hierfür, anstelle der Festplatten den Hauptspeicher als Backend zu nutzen, sodass die PQs und EXS ihre Daten stets im Hauptspeicher bewahren. Mit diesem Vorgehen wurden die verschiedenen Testläufe der semi- und voll-externen Variante erneut durchgeführt. Die verschiedenen Laufzeiten dazu finden sich in Tabelle 4.11. Zum Vergleich wurden auch die Zeiten der internen Implementierung erneut aufgelistet.

Graph	intern	semi-extern	voll-extern
Karate	<1s	<1s	<1s
Football	<1s	<1s	<1s
Jazz	<1s	<1s	<1s
Smallworld	1s	1s	6s
com-amazon	1s	3s	17s
cnr-2000	4s	5s	31s
eu-2005	11s	32s	206s
com-youtube	6s	13s	57s
in-2004	-	26s	176s
road_central	116s	209s	891s
road_usa	201s	388s	1 764s

Tabelle 4.11: Zeiten der verschiedenen Varianten bei Nutzung des Hauptspeichers als Backend.

Zunächst ist zu sehen, dass dieses Experiment nur mit den kleinen bis mittelgroßen Graphen durchgeführt werden konnte, da sonst der Hauptspeicher vollläuft. Da sowohl die semi- als auch voll-externe Variante insgesamt kompliziertere Datenstrukturen aufbauen als die ursprüngliche Variante, benötigen sie bei diesen Tests im Vergleich zur internen Implementierung entsprechend mehr Hauptspeicher.

Bei den Laufzeiten lassen sich immer noch größere Unterschiede feststellen. So benötigt die interne Variante auf dem *eu-2005*-Graphen gerade einmal 11 Sekunden, die semi-externe bereits 32 Sekunden, obwohl alle Daten im Hauptspeicher gehalten werden. Die voll-externe Variante benötigt auf diesem Graphen sogar 206 Sekunden, was eine klare Aussage darüber ist, dass die Verwendung von PQ und EXS auch dann einen erkennbaren Mehraufwand erzeugt, wenn im Hintergrund der Hauptspeicher steht. Vergleicht man aber die hier gemessenen Zeiten für den Graphen mit den Zeiten bei Ausführung auf den Festplatten, so zeigt sich allerdings auch, dass die Festplatten die Laufzeit spürbar verlängern. Die semi-externe Variante benötigt bei der Ausführung mit Festplatten 116 Sekunden mehr, die voll-externe Variante sogar 233 Sekunden. Diese Werte lassen sich nun mit den, in den I/O-Statistiken für diesen Graphen, angegebenen Werten für die *I/O Wartezeit* vergleichen. Diese Beträgt bei der semi-externe Variante 106 Sekunden und bei der voll-externen Variante 220 Sekunden. Das heißt, dass tatsächlich die meiste Zeitersparnis dadurch erreicht wird, dass nicht mehr darauf gewartet werden musste, bis die Daten von der Festplatte in den Hauptspeicher geladen wurden.

Zusammenfassend lässt sich somit festhalten, dass die I/Os auf Festplatten einen großen Anteil an der Gesamtlaufzeit haben und die Verwendung schnelleren Speichers durchaus sinnvoll ist. Die einzelnen Berechnungen könnten dabei um jeweils mindestens den Anteil der I/O Wartezeit, welche in Abbildung 4.7 dargestellt ist beschleunigt werden.
Realistisch betrachtet ist es demnach eine gute Option auf sogenannte Solid-State-Drives zurückzugreifen, da diese erheblich schnellere Zugriffszeiten aufweisen und auch bei der Datenübertragung deutlich schneller sind. Aber selbst mit ihnen lassen sich nicht die Laufzeiten einer internen Implementierung erreichen.

5. Fazit

In dieser Arbeit wurden eine semi- und eine voll-externe Variante der Louvain-Methode entwickelt, theoretisch betrachtet und anschließend experimentell evaluiert. Die Erkenntnisse aus den Messungen für die beiden vorgestellten Varianten zur Berechnung von Clusterungen lassen sich wie folgt zusammenfassen:

Sofern die zu clusternden Graphen klein genug sind um mit einem internen Verfahren betrachtet zu werden, so ist dieses von der Laufzeit her überlegen. Eine Anwendung der externen Varianten auf diesen Graphen zeigt keinen niedrigeren Hauptspeicherbedarf und ist deshalb bei gleichzeitig höheren Laufzeiten nicht sinnvoll. Sobald die Graphen für die interne Louvain-Methode zu groß werden (bei vorgegebener verfügbarer Hauptspeichergröße wie etwa 32 GB in den Testläufen), ist der Umstieg auf eine externe Variante notwendig.

Beim Vergleich zwischen der semi- und voll-externen Variante weißt die semi-externe Variante deutlich kürzere Laufzeiten auf und sollte verwendet werden, sofern der vorgesehene Hauptspeicher das zulässt. Der Hauptspeicherbedarf wächst für große Graphen nahezu linear mit der Anzahl der Knoten. Für den Graphen *uk-2007* mit 105 Millionen Knoten wurden 7 GB Hauptspeicher benötigt. Bei verfügbaren 32 GB Hauptspeicher wäre demnach bei ca. 400-500 Millionen Knoten auch die Grenze für die semi-externe Variante erreicht. Ähnliche Überlegungen für die voll-externe Variante zeigen, dass die Grenze hier bei einer Größenordnung von ca. 4-5 Milliarden Knoten erreicht wäre (Bei der Hochrechnung wird davon ausgegangen, dass bei den 3,5 GB Hauptspeicherbedarf für *uk-2007* mindestens 2-3 GB für die PQs und EXS anfallen, deren Speicherbedarf bei größeren Graphen nicht groß zunehmen würde).

Der Bereich oberhalb von 400-500 Millionen Knoten ist aber eigentlich bei 32 GB Hauptspeicher auch durch die voll-externe Variante nicht mehr vernünftig abzudecken. Bereits für 105 Millionen Knoten ergab sich bei *uk-2007* eine Laufzeit von über zwei Tagen. Bei ähnlicher Relation von Kanten- zu Knotenzahl wäre für 4 Milliarde Knoten bereits mit dem Faktor $40 \cdot \log 40$ zu rechnen, also einer Laufzeit in der Größenordnung von über einem Jahr, was sicherlich inakzeptabel ist. Der Ansatz der voll-externen Variante, möglichst viele anfallende Daten auf den externen Speicher zu verschieben, ist nicht von Erfolg gekrönt und kann anhand der Testergebnisse nicht als sinnvoll erachtet werden.

Die semi-externe Variante hat hingegen deutlich besser abgeschnitten. Nicht nur dass bereits durch die einfache Optimierung der Kantenauslagerung der Hauptspeicherbedarf erheblich reduziert werden konnte spricht für sie, sondern auch die im Vergleich zur voll-externen Variante deutlich besseren Laufzeiten. Die Laufzeit für *uk-2007* war ca. 13 Stunden, für

einen maximal möglichen Graphen mit cs. 500 Millionen Knoten kommt man mit dem Faktor $5 \cdot \log 5$ in die Größenordnung von 5-6 Tagen.

Durch externe Speichermedien mit schnelleren Zugriffszeiten würden sich die hier dargestellten Laufzeiten noch einmal deutlich verringern lassen. Der Einsatz von SSDs ist demnach sehr sinnvoll, vor allem bei häufigen Berechnungen auf große Graphen. (Anbei bemerkt: Auch das Einlesen von Graphen in die Programme und das Speichern der Ergebnisse funktioniert auf SSDs deutlich schneller, mit ihnen profitieren also auch interne Clusterverfahren.)

Zusätzlich zu den Erkenntnissen bei den externen Umsetzungen der Louvain-Methode wurden auch allgemeine Erkenntnisse zu der Methode selbst gemacht: Die Local-Moving-Phase muss nicht immer bis zum Ende durchgeführt werden und auch die Randomisierung muss nicht in jeder Runde erfolgen. Durch beide dieser Vorgänge verliert das Ergebnis der Methode nicht an Qualität.

Ausblick
Gerade gegen Ende der Arbeit, nachdem alle Testergebnisse vorlagen, entstanden immer wieder neue Ideen für weitergehende Optimierungen, für deren Umsetzung es allerdings an Zeit gemangelt hat. Dennoch soll ein kurzer Überblick über diese Optimierungsmöglichkeiten gegeben werden.

Eine potentielle Verbesserung, die erst gegen Ende erkannt wurde, ist der Gebrauch der Prioritätswarteschlangen in der voll-externen Variante. Statt pro Runde zwei PQs und zwei EXS zu verwenden würde eine PQ vollkommen ausreichen, da die PQ für die Folgerunde zu Beginn jeweils leer ist. Auch könnte in den Rekursionen noch erheblich an Speicherplatz gespart werden, durch temporäres deallokieren von Speicher.

Für die interessantere semi-externe Variante könnte man einen Ansatz mit einer abgewandelten Randomisierung austesten, bei der jegliche Umsortierung der einmal gespeicherten Kanten vermieden werden kann. Die Idee lässt sich am Beispiel der 16 R4 Variante (16 Local-Moving-Runden, Änderung der Reihenfolge nach vier Runden) gut erläutern. Offenbar benötigt man hier pro Kontraktionsstufe nur vier verschiedene Reihenfolgen. Wenn man die Ausgangsdaten (Knoten und Kanten) in jeweils drei Teile aufteilt (also drei interne Vektoren für die Knoten und drei dazu passende externe Datenbestände für die Kanten), könnte man diese drei Teile in sechs unterschiedlichen Reihenfolgen abarbeiten, was für 16 R4 ausreichen würde. Jeder Bestand behält aber seine interne Sortierung. Man kann nicht sagen, ob die dadurch erzeugten Reihenfolgen besser oder schlechter sind als die Randomisierung über ein Hash-Verfahren, die Bearbeitung würde aber sicherlich deutlich beschleunigt, da das Sortieren und die resultierenden Schreibzugriffe entfallen.

Die Einführung einer Hybrid-Variante, die zunächst als semi-externe Variante startet, aber ab einer bestimmten Kontraktionsstufe (wenn der kontrahierte Graph klein genug ist) mit der internen Variante weiterrechnet könnte die Rekursionsphase weiter beschleunigen.

Konzepte bei denen stets ganze Teilgraphen zur Berechnung in den Hauptspeicher geladen werden, sodass auf ihnen die Berechnungen schnell durchgeführt werden und die Ergebnisse der verschiedenen Teilgraphen am Ende zu einem großen Ergebnis zusammengeführt werden können scheinen ebenfalls erfolgversprechend.

Die in dieser Bachelorarbeit entwickelten Varianten halten sich somit eventuell doch etwas zu strikt an die ursprüngliche Louvain-Methode, welche als internes Verfahren zu Graphclusterung entwickelt wurde. Dennoch wurde mit der semi-externen Variante das Ziel erreicht, die Clusterungen großer Graphen auf Systemen mit vergleichsweise wenig Hauptspeicher durch die Verwendung externen Speichers mit annehmbaren Rechenzeiten zu ermöglichen.

Literaturverzeichnis

[AV87] A. Aggarwal und J. S. Vitter: *Automata, Languages and Programming: 14th International Colloquium Karlsruhe, Federal Republic of Germany, July 13–17, 1987 Proceedings*, Kapitel The I/O complexity of sorting and related problems, Seiten 467–478. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987, ISBN 978-3-540-47747-1. http://dx.doi.org/10.1007/3-540-18088-5_40.

[BDG+08] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski und D. Wagner: *On Modularity Clustering*. Knowledge and Data Engineering, IEEE Transactions on, 20(2):172–188, Feb 2008, ISSN 1041-4347.

[BGLL08] V. Blondel, J. L. Guillaume, R. Lambiotte und E. Lefebvre: *Fast unfolding of communities in large networks*. Journal of Statistical Mechanics: Theory and Experiment, 10:10008, October 2008.

[BK98] G. S. Brodal und J. Katajainen: *Algorithm Theory — SWAT'98: 6th Scandinavian Workshop on Algorithm Theory Stockholm, Sweden, July 8–10, 1998 Proceedings*, Kapitel Worst-case efficient external-memory priority queues, Seiten 107–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, ISBN 978-3-540-69106-8. http://dx.doi.org/10.1007/BFb0054359.

[Blo14] V. Blondel: *Computational complexity of the Louvain method*, 2011 (Abgerufen am 26.02.2014). https://perso.uclouvain.be/vincent.blondel/research/louvain.html.

[BMSW13] D. A. Bader, H. Meyerhenke, P. Sanders und D. Wagner (Herausgeber): *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, Band 588 der Reihe *Contemporary Mathematics*. American Mathematical Society, 2013, ISBN 978-0-8218-9038-7. http://dx.doi.org/10.1090/conm/588.

[BV04] P. Boldi und S. Vigna: *The WebGraph Framework I: Compression Techniques*. In: *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, Seiten 595–601, Manhattan, USA, 2004. ACM Press.

[DKS08] R. Dementiev, L. Kettner und P. Sanders: *STXXL: standard template library for XXL data sets*. Software: Practice and Experience, 38(6):589–637, 2008, ISSN 1097-024X. http://dx.doi.org/10.1002/spe.844.

[Gö10] R. Görke: *An algorithmic walk from static to dynamic graph clustering*. Dissertation, Karlsruhe Institute of Technology, 2010. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000018288, http://d-nb.info/1003331343.

[LK14] J. Leskovec und A. Krevl: *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data, June 2014.

[NG04] M. E. J. Newman und M. Girvan: *Finding and evaluating community structure in networks*. Physical Review, E 69(026113), 2004.

[PA11] G. Pike und J. Alakuijala: *Introducing CityHash*, 2011. `http://google-opensource.blogspot.de/2011/04/introducing-cityhash.html`, besucht: 16.03.2016.

[RN11] R. Rotta und A. Noack: *Multilevel Local Search Algorithms for Modularity Clustering*. J. Exp. Algorithmics, 16:2.3:2.1–2.3:2.27, July 2011, ISSN 1084-6654. `http://doi.acm.org/10.1145/1963190.1970376`.

[SM16] C. L. Staudt und H. Meyerhenke: *Engineering Parallel Algorithms for Community Detection in Massive Networks*. IEEE Transactions on Parallel and Distributed Systems, 27(1):171–184, Jan 2016, ISSN 1045-9219.

[Sot14] Sotera: *Distributed Graph Analytics - GraphX*, 2014. `https://github.com/Sotera/distributed-graph-analytics/tree/master/dga-graphx#louvain-modularity`, besucht: 12.04.2016.

[Vit08] J. S. Vitter: *Algorithms and Data Structures for External Memory*. Found. Trends Theor. Comput. Sci., 2(4):305–474, January 2008, ISSN 1551-305X. `http://dx.doi.org/10.1561/0400000014`.

Anhang

A Exakte Testergebnisse

Graph	Ursprünglich	32 R1	32 R4	32 R8	16 R4	16 R8
Karate	0,29s	0,32s	0,14s	0,14s	0,14s	0,54s
Football	0,32s	0,71s	0,33s	0,80s	0,93s	0,60s
Jazz	0,61s	0,36s	0,32s	0,82s	0,39s	0,64s
Smallworld	54,46s	54,83s	29,17s	26,79s	28,91s	26,33s
com-amazon	60,33s	60,03s	32,23s	26,88s	32,86s	25,40s
cnr-2000	79,42s	59,64s	38,87s	37,12s	30,31s	28,37s
eu-2005	337,87s	342,89s	232,52s	205,09s	147,64s	145,99s
com-youtube	60,69s	61,62s	39,35s	36,37s	37,61s	34,26s
in-2004	1235,8s	260,25s	185,18s	170,64s	118,01s	108,92s
road_central	600,52s	658,412s	424,98s	469,173s	335,43s	342,32s
road_usa	1134,06s	1100,09s	708,01s	598,86s	615,21s	491,73s
uk-2002	-	5134,52s	3618,55s	3253,15s	2212,68s	1994,97s

Tabelle A.1: Exakte Laufzeiten der verschiedenen Testläufe, zur Parameterwahl der Rundenanzahl und Randomisierungshäufigkeit, auf allen Graphen.

Graph	32MB	64MB	128MB	256MB	512MB	1GB	2GB
Karate	0,0001s	0,0001s	0,0001s	0,0001s	0,0001s	0,0001s	0,0001s
Football	0,0006s	0,0005s	0,0006s	0,0006s	0,0005s	0,0006s	0,0005s
Jazz	0,0033s	0,0133s	0,0023s	0,0018s	0,0029s	0,0022s	0,0016s
Smallworld	1,34s	1,83s	1,80s	1,54s	1,87s	1,70s	1,41s
cnr-2000	5,53s	5,71s	5,68s	6,98s	8,09s	7,56s	6,71s
eu-2005	49,43s	26,97s	18,30s	17,64s	25,79s	19,94s	20,27s
in-2004	39,67s	23,33s	16,41s	16,01s	20,37s	20,05s	17,90s
road_central	83,85s	54,50s	31,60s	28,20s	36,98s	33,19s	29,47s
road_usa	147,67s	91,78s	51,79s	45,02s	61,00s	58,83s	54,15s
uk-2002	1643,33s	933,78s	556,17s	331,11s	370,57s	294,18s	271,18s
uk-2007	-	-	8085,04s	6737,89s	7148,86s	5567,66s	5211,01s

Tabelle A.2: Durchschnittliche Laufzeit einer *local-moving Runde* auf verschiedenen Grahpen bei unterschiedlicher Arbeitsspeicherzuweisung für Sorter und Prioritätswarteschlange in der Voll-Externen Variante.

Graph	n	m	Intern	Semi-extern	Voll-Extern
Karate	34	78	2,9 KB	1,3 KB	0,1 KB
Football	115	613	18,0 KB	4,5 KB	0,4 KB
Jazz	198	2 742	70,5 KB	7,7 KB	0,8 KB
smallworld	100 000	499 998	14,5 MB	3,8 MB	0,4 MB
com-amazon	334 863	925 872	31,4 MB	12,8 MB	1,2 MB
cnr-2000	325 557	2 738 969	72,6 MB	12,4 MB	1,2 MB
eu-2005	862 664	16 138 468	395,7 MB	32,9 MB	3,3 MB
com-youtube	1 134 890	2 987 624	103,0 MB	43,3 MB	4,3 MB
in-2004	1 382 908	13 591 473	353,3 MB	52,8 MB	5,3 MB
road_central	14 081 816	16 933 413	817,3 MB	537,2 MB	53,7 MB
road_usa	23 947 347	28 854 312	1,4 GB	913,5 MB	91,4 MB
uk-2002	18 520 486	261 787 258	6,4 GB	706,5 MB	70,7 MB
twitter-2010	41 652 230	1 202 513 195	28,1 GB	1,6 GB	159 MB
uk-2007	105 896 555	3 301 876 564	77,0 GB	3,9 GB	404,0 MB

Tabelle A.3: Auflistung des theoretischen internen Speicherverbrauchs der verschiedenen Variante bei Berechnungen der *local-moving Phase* auf den aufgelisteten Graphen.

Graph	n	m	Intern	Semi-extern	Voll-Extern
Karate	34	78	234 KB	23,1 MB	10,8 MB
Football	115	613	728 KB	25,5 MB	15,3 MB
Jazz	198	2 742	732,0 KB	25,6 MB	13,4 MB
Smallworld	100 000	499 998	17,6 MB	90,8 MB	259,1 MB
com-amazon	334 863	925 872	35,9 MB	155,0 MB	259,6 MB
cnr-2000	325 557	2 738 969	40,5 MB	341,2 MB	581,5 MB
eu-2005	862 664	16 138 468	173,1 MB	1,4 GB	1,5 GB
com-youtube	1 134 890	2 987 624	113,4 MB	426,8 MB	897,9 MB
in-2004	1 382 908	13 591 473	-	1,4 GB	1,4 GB
road_central	14 081 816	16 933 413	1,6 GB	2,0 GB	3,5 GB
road_usa	23 947 347	28 854 312	2,5 GB	2,4 GB	4,2 GB
uk-2002	18 520 486	261 787 258	-	2,1 GB	2,7 GB
Twitter-2010	41 652 230	1 202 513 195	zu groß	3,2 GB	3,5 GB
uk-2007	105 896 555	3 301 876 564	zu groß	7,0 GB	3,4 GB

Tabelle A.4: Auflistung des gemessen Speicherverbrauchs der verschiedenen Variante bei Ausführung auf den verschiedenen Graphen.