Goethe-Universität Frankfurt

Bachelorarbeit

Eine Implementierung von Kantenlöschung für dynamische Breitensuche im Externspeicher

eingereicht bei

Prof. Dr. Ulrich Meyer

Professur für Algorithm Engineering

von

Fabian Knöller

04. April 2016

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Frankfurt, den 04.04.2016

_____

(Fabian Knöller)

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Der Anstieg der täglich produzierten Datenmenge, hat den Fortschritt bei der Entwicklung größerer interner Speicher von Rechnern überholt. Untersuchte Daten können daher nicht immer im internen Speicher gehalten werden, sondern liegen im Externspeicher. Zugriffe auf Daten im Externspeicher sind jedoch deutlich zeitintensiver als Rechenoperationen im Prozessor. Daher sind Algorithmen zur I/O-effizienten Verarbeitung von Daten, die nicht vollständig in den internen Speicher geladen werden können, in den Fokus der Algorithmenentwicklung gerückt. Ein Teilgebiet sind die Graphalgorithmen, insbesondere die Breitensuche.

Basierend auf vorangegangenen Arbeiten zur dynamischen Breitensuche im Externspeicher, setzt sich diese Arbeit mit dem Löschen von Kanten auseinander. Es wird gezeigt, dass die theoretische I/O-Grenze für Breitensuche im Externspeicher beim Einfügen von Kanten ebenso für das Löschen gilt. Außerdem erfolgt im Rahmen der Arbeit eine Implementierung der Kantenlöschung, die evaluiert wird. Die Experimente mit realen Graphen zeigen, dass die speziellen Algorithmen gegenüber einer wiederholten Anwendung statischer Breitensuche nach jeder Kantenlöschung deutliche zeitliche Vorteile bieten.

Kapitel 1

Einleitung

Das soziale Netzwerk Facebook zählt aktuell ca. 1 Milliarde täglich aktiver Mitglieder [10]. Viele dieser Mitglieder stehen zueinander in Beziehungen unterschiedlicher Art, seien es Freundschafts-, Familien- oder Partnerschaftsbeziehungen. Zusätzlich stellen viele Mitglieder Beziehungen zu Dingen her, die ihnen gefallen, z.B. Künstler, Spiele, Schriftsteller oder Filme. Auch können sich die Mitglieder in Gruppen zusammenfassen, die sich wiederum mit einem bestimmten Thema befassen. All diese mannigfaltigen Beziehungen lassen sich aus Datensicht als ein Graph darstellen, indem die Objekte als Knoten und Beziehungen als Kanten aufgefasst werden.
Eine Besonderheit eines solchen Graphen ist, dass er aufgrund der Anzahl der Objekte sehr groß wird. Eine weitere Herausforderung in der Betrachtung liegt zudem in der Dynamik des Graphen: Mitglieder des Netzwerkes verändern laufend ihre Beziehungen zu anderen Mitgliedern oder Dingen, die ihnen gefallen, sodass in diesem Graph ständig Kanten neu hinzukommen oder entfallen.

In den letzten Jahren ist die Größe der verarbeiteten Datenmenge immer weiter angewachsen und auch für die Zukunft wird ein Wachstum vorhergesagt. So soll die Menge der produzierten Daten von 2009 bis ins Jahr 2020 um einen Faktor 44 ansteigen [9]. Die Art der Daten ist dabei sehr vielfältig und beschränkt sich nicht auf nutzergenerierten Inhalt wie bei Facebook, sondern wird im industriellen Bereich vor allem aus Sensoren gewonnen. Beispiele sind hier die Fahrtaufzeichnungen von Straßenfahrzeugen im Bereich der Assistenzsysteme und des autonomen Fahrens, die Überwachung von Netzen zur Daten- oder Energieversorgung oder die Messung einzelner technischer Komponenten eines Gesamtsystems wie Verkehrssignalanlagen oder Weichenantriebe der Eisenbahn. Aber auch wissenschaftliche Untersuchungen liefern Anwendungsfälle mit Generierung großer Datenmengen in Experimenten, deren Zusammenhänge analysiert werden müssen – insbesondere im Bereich der Teilchenphysik. Um solche Zusammenhänge zu erkennen, werden aus den gewonnenen Daten Netzwerke gebildet – dies erfolgt im industriellen Bereich bevorzugt unter den Stichworten Data Mining und Big Data. Die Zahl der Graphen, die die oben genannten Besonderheiten – Größe und Dynamik – aufweisen, wächst ständig.
Auf diese Herausforderung müssen Algorithmen reagieren, da je nach verfügbarer Hardware und notwendiger Datenstruktur zur Ablage aller relevanten Informationen die Graphen so groß sind, dass sie

im internen Speicher, dem Arbeitsspeicher und Cache, nicht vollständig dargestellt werden können. Das bedeutet wiederum, dass Algorithmen, die auf diesen Graphen operieren, auf Informationen zugreifen müssen, die in einem externen Speichermedium, z.B. einer Festplatte, gespeichert sind. Solche Leseoperationen sind um ein Vielfaches langsamer als Rechenoperationen mit Zugriff auf Daten die im internen Speicher liegen.

In Bezug auf die Breitensuche wurde dieses Problem bereits mehrfach untersucht. Die vorliegende Arbeit baut auf den Grundlagen aus der generellen Untersuchung der dynamischen Breitensuche im Externspeicher (s. [13]) und der darauf aufbauenden Implementierung des Einfügens von Kanten (s. [4]) auf. Ziel dieser Arbeit ist die Bestätigung der oberen I/O-Schranke von

$$O\left(n\left(\frac{n}{B^{\frac{2}{3}}} + sort(n) \cdot log(B)\right)\right) \text{ I/O}$$

für dynamische Breitensuche im Externspeicher mit $\Theta(n)$ Kantenlöschungen. Parallel werden konkrete Algorithmen für dynamische Breitensuche im Externspeicher nach dem Löschen von Kanten entwickelt. Diese werden im bereits seit der Erarbeitung in [4] bestehenden Code ergänzt. Mittels Experimenten anhand synthetischer und realer Testdaten erfolgt die experimentelle Untersuchung des erweiterten Codes.
In Kapitel 2 werden zunächst die theoretischen Grundlagen geklärt und in Kapitel 3 der aktuelle Stand der Wissenschaft zur dynamischen Breitensuche im Externspeicher beim Einfügen von Kanten dargestellt. Kapitel 4 stellt die Theorie und praktische Implementierung des Löschens von Kanten bei dynamischer Breitensuche auf Graphen, die im Externspeicher liegen, vor. Die experimentelle Untersuchung der vorliegenden Implementierung wird in Kapitel 5 beschrieben und abschließend ein Ausblick auf die weitere Entwicklung in Kapitel 6 gegeben.

Kapitel 2

Grundlagen

Zur Beschreibung der vorliegenden Implementierung des Löschens von Kanten bei der dynamischen Breitensuche auf Graphen, die im Externspeicher gehalten werden müssen, werden die benötigten Grundlagen im folgenden Kapitel erläutert.

2.1 EM-Modell

Um das generelle Problem von Algorithmen, die auf im Externspeicher gehaltene Daten optimiert werden, zu erläutern, wird auf das Externspeichermodell von Aggarwal und Vitter zurückgegriffen [1]. In diesem Modell werden der interne Speicher (*engl.: internal memory; kurz IM*) und der externe Speicher (*engl.: external memory; kurz EM*) unterschieden. Die Größe des internen Speichers wird mit M angegeben. Der externe Speicher ist theoretisch unbegrenzt groß, hier wird statt der Gesamtgröße als relevante Messgröße die Zahl der Daten, die in einer Zugriffsoperation (*engl.: input/output; kurz I/O*) zurückgegeben werden kann, angegeben. Dies wird beschrieben durch die Blockgröße B.
Für Algorithmen, die auf im Externspeicher gehaltene Daten optimiert werden, gilt dann, dass die Problemgröße N deutlich größer ist als der verfügbare Platz im internen Speicher: $N \gg M$. In diesem Fall ist die Standardgröße zur Laufzeit eines Algorithmus, die Zahl der Berechnungsschritte, nicht mehr die alleinige ausschlaggebende Größe. Da Zugriffe auf Daten, die im Externspeicher liegen, deutlich länger dauern als eine Berechnung im Rechenkern, ist die Zahl der benötigten I/O hier die relevante Vergleichsgröße.
Relevante untere Schranken für die Betrachtung von I/O bei Externspeicher-Algorithmen sind die Schranken

$$scan(N) = \Theta(\tfrac{N}{B}) \text{ und}$$
$$sort(N) = \Theta(\tfrac{N}{B} \cdot log_{\frac{M}{B}}(\tfrac{N}{B}))$$

für das Lesen von aufeinanderfolgend abgelegten Daten im Externspeicher bzw. für das Sortieren von Daten im Externspeicher [13].

2.2 Graphen

Graphen im Sinne der vorliegenden Arbeit bestehen aus einer Menge von Knoten $V = \{v_1, \ldots, v_n\}$ mit $|V| = n$ und einer Menge von Kanten $E = \{e_1, \ldots, e_m\}$ mit $|E| = m$. Eine Kante $e \in E$ verbindet je zwei Knoten $u, v \in V$, in diesem Fall werden u und v als Nachbarn bezeichnet. Ein Graph wird folglich notiert als $G = (V, E)$.

Es wird unterschieden zwischen gerichteten und ungerichteten Graphen. Kanten in gerichteten Graphen – die wie folgt notiert werden: (u, v) – verbinden die Knoten nur in einer Richtung, vom erstgenannten zum letztgenannten Knoten. Kanten in ungerichteten Graphen verbinden die Knoten immer in beiden Richtungen miteinander, die Notation ist in diesem Fall: $\{u, v\}$. In der vorliegenden Arbeit werden nur ungerichtete Graphen betrachtet.

Der Grad eines Knoten v gibt die Zahl der ein- bzw. ausgehenden Kanten dieses Knotens an. Bei gerichteten Graphen können Ein- und Ausgangsgrad eines Knotens verschieden sein, bei ungerichteten Graphen lässt sich je Knoten genau ein $Grad(v_i)$ angeben.

Die Darstellung eines Graphen mittels Datenstrukturen kann auf verschiedene Arten erfolgen. Ein bekanntes Verfahren ist die Darstellung in einer Adjazenzmatrix, in der je Knotenpaar des Graphen das Vorhandensein einer Kante zwischen diesen Knoten angegeben wird. In dünnen Graphen wird oft die Darstellung mittels Adjazenzlisten, also einer Liste je Knoten, in dem alle Nachbarn des Knotens angegeben sind, bevorzugt. Weitere geeignete Datenstrukturen existieren (s. z.B. [8, S. 211ff.]), sind im Rahmen dieser Arbeit jedoch nicht relevant.

2.3 Clusterung im Graphen

Knoten eines Graphen können nach diversen Kriterien als zusammengehörig bezeichnet werden. Im hier besprochenen Anwendungsfall der dynamischen Breitensuche auf Graphen, die im Externspeicher gehalten werden müssen, bezieht sich die Zusammengehörigkeit auf die räumliche Nähe der Knoten.

Die Clusterung in Graphen dient dem Ziel, zusammengehörige Knoten zu identifizieren und in Clustern zusammenzufassen. Das bedeutet, dass je ein Subset von Knoten $\{v_s, \ldots, v_t\} \in V$ einem Cluster C_i zugeordnet wird. Jeder Knoten befindet sich in genau einem Cluster. Nach welchen Kriterien diese Zuordnung erfolgt, hängt von der Definition der Zusammengehörigkeit im konkreten Fall ab. Im vorliegenden Anwendungsfall wird damit die hohe Wahrscheinlichkeit abgebildet, mit der alle Knoten des Clusters als nächstes besucht werden, sobald einer der Knoten besucht wurde.

Die Größe der Cluster kann unterschiedlich sein, oder aber möglichst gleich groß gewählt sein, so wie es für den vorliegenden Anwendungsfall am sinnvollsten ist. Eine Clusterung gleicher Größe ist zwar schwerer zu finden als ungleich verteilte Cluster, dafür kann mit einer Gleichverteilung beispielsweise die Arbeitslast bei Parallelisierung gleichmäßiger verteilt werden.

2.4 Breitensuche

Eine der grundlegenden Methoden, um einen zusammenhängenden Graphen zu traversieren, ist die Breitensuche *(engl. breadth-first search, kurz BFS)*. Die Breitensuche kann aufgefasst werden, als die Bestimmung der Länge kürzester Wege von einem Wurzelknoten w zu allen anderen Knoten des Graphen.
Die Wurzel w des Graphen erhält zunächst das BFS-Level 0. Anschließend läuft der Algorithmus einmal durch den Graphen, bis alle Knoten einmal besucht wurden. Dies erfolgt nach folgendem Schema:

Algorithm 1: Pseudocode für Breitensuche

```
1  for all nodes i in V do
2      besucht[i] = false;
3  Queue Q = {w};
4  Queue nextQ = {};
5  int level = 0;
6  while !Q.empty() do
7      for all nodes i in Q do
8          if !besucht[i] then
9              i.BFSlevel = level;
10             besucht[i] = true;
11             for all neighbours j from i do
12                 nextQ.push(j);
13     Q = nextQ;
14     level++;
```

Mit diesem Verfahren erhält man einen BFS-Baum, in dem das Level jedes Knotens die minimale Entfernung dieses Knotens vom Wurzelknoten w der Breitensuche angibt. Der vorgestellte Algorithmus löst das Problem mit einer Laufzeit von $O = (n+m)$ [5] im internen Speicher eines Rechners effizient.

2.5 Breitensuche im Externspeicher

Führt man eine Breitensuche auf einem Graphen durch, der nicht vollständig in den internen Speicher der Größe M passt, so müssen an zwei Stellen im Algorithmus die Adjazenzlisten des aktuell betrachteten Knotens möglicherweise aus dem Externspeicher geladen werden:

1. Überprüfung, ob ein Knoten bereits besucht wurde.
2. Laden der Nachbarn des aktuell betrachteten Knotens.

Da nicht garantiert werden kann, dass die Adjazenzlisten in derselben Reihenfolge sortiert sind, wie die Knoten im Graphen besucht werden, kann hier eine Vielzahl von unstrukturierten Speicherzugriffen in den Externspeicher entstehen. Besonders problematisch ist dies, wenn die jeweils aufeinander folgenden Zugriffe in unterschiedliche Blöcke des Externspeichers erfolgen müssen. Wenn der oben vorgestellten Algorithmus im worst-case $\Theta(n+m)$ I/O Operationen benötigt, verschlechtert sich die Laufzeit deutlich [13]. Für die statische oder einmalige Breitensuche auf einem ungerichteten Graphen, der mindestens teilweise im Externspeicher liegt, wird dieses Problem von zwei Algorithmen gelöst, die im Folgenden vorgestellt werden.

2.5.1 BFS-Algorithmus für ungerichtete dichte Graphen im Externspeicher

Munagala und Ranade setzten sich in [14] mit den unstrukturierten Zugriffen zur Prüfung ob ein Knoten bereits besucht wurde auseinander. Ihr Algorithmus, der im Folgenden als MR_BFS bezeichnet wird, macht sich die Tatsache zunutze, dass benachbarte Knoten in ungerichteten Graphen immer in benachbarten oder demselbem BFS-Level liegen. Um alle Knoten des Levels L_i mit der BFS-Tiefe i zu erhalten, werden zunächst alle Nachbarn der Knoten in Level L_{i-1} als Menge $potL_i$ der potentiellen Knoten des Level i geladen. Anschließend werden aus den Nachbarn alle Knoten, die in L_{i-1} oder L_{i-2} liegen, wieder entfernt: $L_i := potL_i \setminus \{L_{i-1} \cup L_{i-2}\}$
Dieser Vorgang wird dadurch I/O-effizient gestaltet, dass das Filtern der Nachbarknoten durch paralleles Scannen der drei Mengen $potL_i$, L_{i-1} und L_{i-2} erfolgt. Dieses erfordert, dass die Menge $potL_i$ zuvor sortiert und dabei von Duplikaten befreit wird. Insgesamt führt dies also zu einer oberen I/O-Schranke von $O(scan(\sum_i L_i) + sort(\sum_i potL_i))$, die sich als $O(n + sort(n+m))$ darstellen lässt. Um die Nachbarn der Menge L_{i-1} erstmalig nach $potL_i$ zu laden, sind allerdings weiterhin unstrukturierte Zugriffe auf die Adjazenzlisten der betroffenen Knoten notwendig.

2.5.2 BFS-Algorithmus für ungerichtete dünne Graphen im Externspeicher

Um auch den unstrukturierten Zugriff auf die Adjazenzlisten zur Bestimmung aller Nachbarn zu vermeiden, wurde MR_BFS von Mehlhorn und Meyer zu MM_BFS weiterentwickelt [12]. In ihrem Algorithmus findet eine Vorverarbeitung des Graphen statt, die dazu führt, dass in der BFS-Phase nicht mehr jede einzelne Adjazenzliste unstrukturiert geladen werden muss. Dadurch kann die Zahl der I/O maximal um einen Faktor von $\sqrt{B}$ verringert werden. Dies wird allerdings nur wirksam, sofern $m = O(n)$ gilt. Ist der Graph dichter, so kann aber asymptotisch dieselbe Zahl von I/O wie bei MR_BFS erreicht werden.
Kern der Vorverarbeitung ist es, Adjazenzlisten von Knoten, die nah beieinander liegen, zusammen im Externspeicher abzulegen. Es wird das Prinzip der Lokalität ausgenutzt: Sobald ein Knoten in der Breitensuche besucht wird, ist es sehr wahrscheinlich, dass als nächstes die Knoten in seiner Nachbarschaft besucht werden. Mit diesem Wissen werden alle Knoten des Graphen G in der Vorverarbeitung disjunkten Clustern (s. Kapitel 2.3) zugeordnet. Hierzu existieren zwei Vorgehensweisen:

1. Zufällige Clusterung: Bei der zufälligen Clusterung werden $k \leq n$ Knoten des Graphen G zufällig ausgewählt, wobei ein Knoten in jedem Fall der Wurzelknoten w der Breitensuche ist. Von jedem der ausgewählten Knoten aus wachsen die Cluster parallel und fügen sich die jeweils nächsten Nachbarn hinzu, bis alle Knoten einem Cluster zugeordnet sind. Dabei kann es notwendig sein an der Grenze zwischen Clustern eine zufällige Entscheidung zur Zuordnung zu treffen. Die Größe der Cluster hängt stark von der Struktur des Graphen und der Wahl der Startknoten ab, sie kann sehr ungleich verteilt sein.

2. Deterministische Clusterung: Um eine gleichmäßigere Aufteilung der Knoten in Cluster zu gewährleisten, wurde in [12] zusätzlich eine deterministische Variante entwickelt. Hierzu wird zunächst ein Spannbaum T_s der verbundenen Komponente des Graphen G berechnet, in dem der Wurzelknoten w der Breitensuche liegt. Dies ist gemäß [3] mit $O((1+loglog(B\cdot\frac{n}{m}))\cdot sort(n+m)$ I/O erreichbar. Alle Kanten des ungerichteten Spannbaumes werden durch zwei entgegengesetzte gerichtete Kanten ersetzt, anschließend wird eine Eulertour auf dem Graphen erstellt. Die Eulertour wiederum wird in gleich große Abschnitte der Größe $\mu = max\{1, \sqrt{\frac{n\cdot B}{n+m}}\}$, beginnend in w, geteilt, die die Cluster darstellen. Da Knoten durch die Eulertour mehrfach besucht werden und somit zunächst in mehreren Clustern liegen können, wird hier eine Entscheidung getroffen, in welchem Cluster diese verbleiben.

Unter Berücksichtigung der einzelnen Vorverarbeitungsschritte (z.B. Berechnung der verbundenen Komponente im Externspeicher oder der Eulertour im Externspeicher) ergibt sich gemäß [12] als obere wahrscheinliche I/O-Grenze bei zufälliger Clusterung, bzw. worst-case I/O-Grenze bei deterministischer Clusterung $O(\sqrt{\frac{n\cdot(n+m)}{D\cdot B}}\cdot sort(n+m))$, wobei D für die Zahl der unabhängigen gleichzeitig les-/beschreibbaren Externspeichermedien steht. Für dichte Graphen kann die worst-case I/O-Grenze von MR_BFS immer erreicht werden, indem die Clustergröße auf 1 begrenzt wird.

Kapitel 3

Einfügen von Kanten bei dynamischer Breitensuche im Externspeicher

Wie bereits in der Einleitung, Kapitel 1, beschrieben, lohnt es sich, neben der statischen, einmaligen Breitensuche, auch die dynamische Breitensuche zu untersuchen. Bei der dynamischen Breitensuche verändert sich der zugrundeliegende Graph G durch Einfügen oder Löschen von Kanten im Laufe der Zeit in k Veränderungsschritten. Der jeweils aktuelle Zustand ist $G_i = (V_i, E_i)$ mit $0 \leq i \leq k$, wobei G_0 der Ursprungsgraph ist. In der realen Anwendung kann eine Veränderung des Graphen mehrere Kanten gleichzeitig beinhalten. Dies soll in der theoretischen Betrachtung jedoch durch mehrere nacheinander durchgeführte Veränderungen dargestellt werden. Um von G_{i-1} zu G_i zu gelangen, wird die Kante $e_c = \{u, v\}$ mit $u, v \in V$ zur Menge E_{i-1} hinzugefügt oder gelöscht. Beim Einfügen gilt also $E_i = E_{i-1} \cup \{e_c\}$, beim Löschen hingegen $E_i = E_{i-1} \setminus \{e_c\}$.

Das Hinzufügen oder Löschen einer Kante im Graphen führt dazu, dass sich das Ergebnis der Breitensuche ggf. verändert. Beispielsweise verkürzt sich der kürzeste Weg vom Wurzelknoten w der Breitensuche zu einem der beiden Knoten und ggf. auch seinen Nachbarn beim Einfügen einer Kante zwischen zwei Knoten, die in vorheriger Iteration $i-1$ mehr als 1 BFS-Level voneinander getrennt waren. Dies drückt sich in verringerten BFS-Leveln aus. Mit $d_i(v)$ wird die Distanz des Knoten v zur Wurzel w der Breitensuche in Iteration i ausgedrückt. Die Veränderung des BFS-Levels eines Knotens wird dann durch $\Delta d_i(v) = |d_{i-1} - d_i|$ ausgedrückt.

Die Breitensuche muss nach einer Veränderung des Graphen angepasst werden. Wie auch in [4] argumentiert wird, lohnt es sich bei der Breitensuche im internen Speicher nicht, hierfür besondere Vorgehensweisen zu etablieren. Asymptotisch ist es nicht aufwendiger, eine vollständig neue Breitensuche ausgehend vom Wurzelknoten durchzuführen, als die Änderungen durch die Einfügung oder Löschung einer Kante zu identifizieren. Dies gilt jedoch nicht für die dynamische Breitensuche auf einem Graphen, der im Externspeicher gehalten werden muss. Hier stellt man fest, dass sich mit geeigneten Methoden der I/O-Aufwand reduzieren lässt, wenn nur die relevanten Änderungen berechnet werden, anstatt eine vollständig neue Breitensuche durchzuführen.

3.1 Theoretische Betrachtung

In [13] wird dargelegt, wie die dynamische Breitensuche auf Graphen, die im Externspeicher liegen, I/O-effizient durchgeführt werden kann. Die dort durchgeführte Betrachtung konzentriert sich dabei auf das Einfügen von Kanten, kann aber analog auch auf das Löschen von Kanten angewandt werden.

Clusterung

Grundlage des Papiers ist der bereits in Kapitel 2.5.2 vorgestellte Algorithmus MM_BFS. Um sicherzustellen, dass die Clustergrößen im Erwartungsfall bei Anwendung der deterministischen Clusterung nicht zu unterschiedlich groß sind, werden mehrfach besuchte Knoten nicht mehr dem ersten sie besuchenden Cluster zugeordnet, sondern zufällig mit einer Wahrscheinlichkeit von je $\frac{1}{2}$ dem ersten bzw. letzten sie besuchenden Cluster. Durch diese Maßnahme ist die erwartete tatsächliche Clustergröße jedes Clusters – mit Ausnahme des letzten Clusters – mindestens $\frac{\mu}{8}$.
In Erweiterung dessen stellt der Artikel [4] eine weitere Möglichkeit zur Clusterung vor, mit der für alle μ, für die $1 \leq \mu = 2^q \leq \sqrt{B}$ gilt, $\Theta(\frac{n}{\mu})$ Cluster mit einer Größe von $\Theta(\mu)$ und Durchmesser $O(\mu)$ erzeugt werden. Hierzu wird eine levelweise hierarchische Clusterung durchgeführt. Kern dessen ist, dass jeder Knoten eine neue Bezeichnung in Form eines Bitvektors $(b_r,\cdots,b_{q+1},b_q,\cdots,b_1)$ erhält. Dieser wird als Kombination aus einem Präfix $(b_r,\cdots,b_{q+1})$, der das Cluster definiert, und einem den konkreten Knoten identifizierenden Suffix $(b_q,\cdots,b_1)$ interpretiert. Die Wahl von q korrespondiert dabei mit der gewählten Clustergröße.
Die levelweise hierarchische Clusterung baut ebenfalls auf dem Spannbaum T_s von G auf. In p Phasen wird dabei jeweils der Spannbaum T_s^j zum Spannbaum T_s^{j+1} verkleinert, wobei die Zahl der Knoten halbiert wird. In $p = \lceil log(\sqrt{B}) \rceil$ Phasen werden so Cluster von maximaler Größe $\mu = \sqrt{B}$ gebildet. Die Clusterbildung erfolgt dabei von den Blättern des Baumes hin zur Wurzel, indem jeweils Geschwister- oder Kind- und Elternknoten zu einem Cluster zusammengefasst werden, das in der nächsten Phase als ein Knoten des Spannbaumes dargestellt wird. Die im Cluster zusammengefassten Knoten haben dabei zwar nicht zwingend jeweils eine Verbindung, aber immer einen kürzesten Weg von maximal $2^{q+1}-2$ zwischen einander. Nach Bildung aller p Spannbäume werden die Knoten mit den erzeugten Bitvektoren benannt und entsprechend umsortiert. Die gesamte Clusterung für alle μ, für die $1 \leq \mu = 2^q \leq \sqrt{B}$ gilt, benötigt $O(sort(n+m))$ I/O.

In [13] wird ohne Nutzung der levelweisen hierarchischen Clusterung gezeigt, dass für allgemeine ungerichtete dünne Graphen mit n Knoten und initial $O(n)$ Kanten bei $\Theta(n)$ Kanteneinfügungen die Anpassung des BFS-Ergebnisses mit hoher Wahrscheinlichkeit mit

$$O\left(\frac{n}{B^{\frac{2}{3}}} + sort(n) \cdot log(B)\right) \text{ I/O}$$

erfolgen kann.

Dazu werden folgende Fälle des Einfügens von Kanten unterschieden:

1. Fall A : Einfügen einer Kante zu einer bisher unverbundenen Graphkomponente
2. Fall B : Einfügen einer Kante innerhalb der verbundenen Graphkomponente der Wurzel w

Fall A

Um zu unterscheiden, in welchem der beiden Fälle man sich bei der Einfügung der aktuellen Kante $e_c = \{u, v\}$ befindet, wird zunächst ein Algorithmus zur Feststellung der Verbundenheit der Graphkomponenten angewandt. Stellt man dabei fest, dass es sich mit e_c um eine Kante des Fall A handelt, so verbindet e_c nun die Komponente C_w, in der sich die Wurzel w befindet, mit der neu hinzugekommenen Komponente. Sei $u \in C_w$ und v in der neu hinzugekommenen Komponente C_v. Da e_c die einzige beide Komponenten verbindende Kante ist, kann in diesem Fall auf C_v ausgehend von Knoten v eine Breitensuche mit MR_BFS durchgeführt werden. Die dabei ermittelten Level aller Knoten n_v der Komponente C_v werden um $d_{i-1}(u) + 1$ erhöht und man erhält den neuen BFS-Baum für G_i. Die Zahl der I/O ergibt sich aus der Berechnung der Verbundenheit mit maximal $O(sort(n) \cdot log(B))$ I/O sowie $O(n_v \cdot sort(n))$ I/O für die Breitensuche auf C_v. In $m' = \Theta(n)$ Experimenten kann Fall A maximal $n-1$ mal auftreten, woraus sich als obere Grenze für die notwendigen I/O $O(n \cdot sort(n) \cdot log(B))$ ergibt.

Fall B

Handelt es sich bei der Einfügung um einen Fall B, so müssen die BFS-Level der Knoten in der bestehenden Komponente C_s neu berechnet werden. Um dies effizient mit möglichst geringer Zahl von I/O durchzuführen, wird ein nach Knotennummern sortierter Hotpool H benutzt, in den die Adjazenzlisten eines Knotens w genau dann geladen werden, wenn das BFS-Level $max\{0, d_{i-1}(w) - \alpha\}$ neu berechnet wird. $\alpha > 1$ ist dabei ein wählbarer Wert für die Vorausschau des Hotpools. Alle Adjazenzlisten, die nicht in H geladen sind, werden in einer nach Knotenlevel sortierten Liste gehalten, so dass das Nachladen von Adjazenzlisten eines weiteren BFS-Levels mittels $O(scan(n))$ I/O möglich ist.

Einfache Fälle B

Liegen die beiden Knoten u, v der Kante e_c im selben BFS-Level, so sind keine Änderungen von BFS-Leveln und somit maximal $O(1)$ I/O notwendig. Gilt für alle Knoten des Graphen nach der Einfügung $\Delta d_i(v) < \alpha$, können alle zur Bestimmung der neuen BFS-Level notwendigen Adjazenzlisten aus H gelesen werden, ohne dass weitere unstrukturierten Zugriffe auf Adjazenzlisten notwendig wären. In diesem Fall wird jede Adjazenzliste maximal einmal in den Hotpool geladen und dort maximal α oft gescannt. Insofern ist die Gesamtzahl der I/O geringer als bei MM_BFS, solange $\alpha = o(\sqrt{B})$ gewählt wird.

I/O-intensive Fälle B

Der komplexeste Fall liegt vor, wenn für manche Knoten $\Delta d_i(v) > \alpha$ gilt. Wird ein Knoten nicht in H gefunden, so werden in diesem Fall die Adjazenzlisten des Knotens und aller anderen Knoten seines Clusters nach H geladen. Die Clustergröße wird auf $\mu = \frac{\alpha}{4}$ festgesetzt. Um zu vermeiden, dass dieses

Vorgehen in $\Theta(\frac{n}{\alpha})$ Clusterzugriffen resultiert, wird die Zahl der unstrukturierten Clusterzugriffe auf maximal $\alpha \cdot \frac{n}{B}$ begrenzt. Wird diese Grenze überschritten, so wird α um den Faktor 2 vergrößert und eine neue Clusterung mit verdoppelter Clustergröße durchgeführt. Die Breitensuche erfolgt nun ebenfalls mit doppelter Hotpoolgröße. Dies resultiert im Versuch $j, j \geq 1$ in einem α_j mit $\alpha_j := 32 \cdot 2^j$ und somit Clustergröße $\frac{\alpha_j}{4} = 8 \cdot 2^j$, sowie einem Hotpoolumfang von α_j BFS-Leveln.
Es gilt $j \leq log(\sqrt{B})$, sodass die Zahl der Fehlversuche mit zu kleinem α auf $O(log(B))$ begrenzt ist. Bei genügend großem j werden alle Adjazenzlisten der n Knoten geladen, so dass die Zahl der I/O maximal dem statischen MM_BFS entspricht. Um zu verhindern, dass Adjazenzlisten, die durch Clusterzugriffe nach H geladen wurden, durch das sequenzielle Nachladen erneut geladen werden, wird jede Adjazenzliste mit einem Zeitstempel versehen. So kann jede Adjazenzliste nach höchstens $\alpha_j = O(2^j)$ betrachteten BFS-Leveln wieder aus H entfernt werden. Dadurch wird die Größe von H begrenzt; die Suche in H bleibt effizient.

I/O-Schranke für alle Fälle B

Die Gesamtzahl der BFS-Level-Veränderungen von Knoten $v \in V$ nach Einfügung der i-ten Kante wird wie folgt definiert: $\Delta D_i = |D_{i-1} - D_i|$ wobei gilt: $D_i = \sum_{v \in V \setminus \{w\}} (d_i(v))$. Es wird gezeigt, dass mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$ $\Delta D_i \geq 2^{3 \cdot j+5} \cdot \frac{n}{B} =: Y_j$ für den Erfolg der i-ten Einfügung beim j-ten Versuch gelten muss. Der Eintritt dieses Ereignisses wird *hoher j-Ertrag* genannt.
Mithilfe der Chernoff-Ungleichung wird gezeigt, dass aus $k \geq 16 \cdot c \cdot ln(n)$ Einfügungen, die nach der gleichen Zahl j an Versuchen erfolgreich sind, mit einer Wahrscheinlichkeit von $1 - n^{-c}$ mindestens $\frac{k}{4}$ einen *hohen j-Ertrag* haben, wobei c eine beliebige positive Zahl ist. Gleichzeitig ist jedoch bei maximal $m' = \Theta(n)$ Einfügungen die Zahl der möglichen Ereignisse mit *hohem j-Ertrag* durch $\frac{n^2}{Y_j} = \frac{n \cdot B}{2^{3 \cdot j+5}}$ begrenzt. Dies resultiert aus der monotonen Reduktion der Summe aller BFS-Level nach $\Theta(n)$ Einfügungen: $n^2 > D_0 \geq D_1 \geq \cdots \geq D_{m'} > 0$. Um auf die maximale Zahl an Ereignissen mit *hohem j-Ertrag* zu kommen, sind mit hoher Wahrscheinlichkeit also maximal $k_j := \frac{4 \cdot n \cdot B}{2^{3 \cdot j+5}}$ Ereignisse notwendig, die im j-ten Versuch die Breitensuche abschließen.
Die auftretenden Fälle B werden nun unterteilt in die Fälle B_1, in denen die Breitensuche mit einem Wert $\alpha_j < B^{\frac{1}{3}}$ abgeschlossen wird und die maximal $\Theta(n)$ mal auftreten können, sowie die Fälle B_2, in denen α_j höher liegt. Für die Fälle B_2 wurde gezeigt, dass diese höchstens $O\left(\frac{n \cdot B}{2^{3 \cdot j}}\right)$ mal auftreten können. Somit argumentiert [13] mittels der Booleschen Ungleichung, dass für alle Fälle B bei $m' = \Theta(n)$ Einfügungen in einen dünnen Graphen die I/O auf $O\left(n\left(\frac{n}{B^{\frac{2}{3}}} + sort(n) \cdot log(B)\right)\right)$ mit hoher Wahrscheinlichkeit begrenzt sind. Als allgemeine I/O-Grenze für eine einzelne Einfügung einer Kante in einen dünnen Graphen wird somit mit hoher Wahrscheinlichkeit $O\left(\frac{n}{B^{\frac{2}{3}}} + sort(n) \cdot log(B)\right)$ gezeigt.

3.2 Implementierung

Aufbauend auf den vorangehend dargelegten Grundlagen erfolgte im Rahmen der Arbeit an [4] eine Implementierung des Einfügens von Kanten bei dynamischer Breitensuche auf Graphen, die im Externspeicher gehalten werden müssen. Dabei wurden die algorithmischen Ideen aus [13] zugrunde

gelegt. Für die Clusterung wurde allerdings die oben bereits erläuterte alternative Vorgehensweise der levelweisen hierarchischen Clusterung verwendet.

Eine weitere Abweichungen gegenüber dem theoretischen Algorithmusdesign ist die Verwendung zweier getrennter Hotpools H und HC. Der Hotpool H wird nur für die sequentiell geladenen Adjazenzlisten verwendet, während die explizit per Cluster geladenen Adjazenzlisten in Hotpool HC gelagert werden. Dies hat den Vorteil, dass unterschiedliche, auf die jeweilige Anwendung zugeschnittene Datenstrukturen verwendet werden können. So werden Cluster-ID und Timer nur für Einträge im Hotpool HC benötigt. Auch ist dieser Zuschnitt für das Durchführen von Experimenten geeignet, da die sequentiellen I/O von den Clusterzugriffen getrennt gemessen werden können.

Ebenfalls erfolgte eine Anpassung hinsichtlich der Verwendung von α: Für den sequentiellen Hotpool H gibt es einen eigenen Wert α_1, wohingegen die Clustergröße über den Parameter α_2 gesteuert wird. Diese Trennung war ein Ergebnis der praktischen Erprobung der Implementierung. Hierbei zeigte sich, dass ein zu großer Wert α für den sequentiellen Hotpool H dazu führt, dass zu viele Level gleichzeitig darin vorgehalten werden und somit die Suche nach Elementen in diesem sehr lange dauert. Gleichzeitig führte ein kleiner Wert α für die Clustergröße dazu, dass es kaum vorkam, dass die Zahl der tolerierten Cluster-I/O überschritten und α neu berechnet wurde. Dies führte zu einer hohen I/O-Zahl durch das Laden vieler kleiner Cluster. Um den Effekten entgegenzuwirken, wurde α in zwei Werte aufgeteilt. Als Initialwert für jede dynamische Breitensuche bestimmt ein kleiner Wert α_1 die Levelzahl in H und ein um Faktor $2^k, k \geq 0$ größerer Wert α_2 bestimmt die initiale Clustergröße der nach HC zu ladenden Cluster. Bei einer notwendigen Anpassung von α werden jedoch beide Werte wie oben beschrieben verdoppelt, die Clusterung neu berechnet und beide Hotpools neu initialisiert.

Eine weitere Aufwandsreduktion war möglich, da die Verbesserung der BFS-Level nach dem Einfügen einer Kante nur für Knoten mit mindestens BFS-Level $min_{level\_improv} = d_{i-1}(u) + \lfloor \frac{d_{i-1}(v) - d_{i-1}(u)}{2} \rfloor + 1$ für $d_{i-1}(v) > d_{i-1}(u)$ möglich ist. Somit müssen nicht die α_1 Level ab $d_{i-1}(u)$ nach H geladen werden, sondern α_1 Level ab min_{level_improv}. Falls $min_{level\_improv} - d_{i-1}(u) > \alpha_1$, kann die Zahl der Cluster-I/O reduziert werden.

Die Implementierung erfolgte in C++ und stützt sich zur Verwaltung des externen Speichers auf die STXXL-Bibliothek [7]. Zur Überprüfung des Nutzens der modifizierten Algorithmen zur dynamischen gegenüber der statischen Breitensuche mittels MM_BFS (s. 2.5.2) wurden Experimente mit ausreichend großen Graphen durchgeführt. Für die verschiedenen Graphen wurde zunächst eine statische Breitensuche durchgeführt und die benötigte Zeit für die Vorverarbeitung sowie der Breitensuche selbst gemessen. Für die dynamische Breitensuche wurde ebenfalls die Zeit für die Vorverarbeitung (Clusterung, Sortieren der Adjazenzlisten) gemessen sowie die eigentliche Zeit der dynamischen Breitensuche für das Einfügen von Kanten zwischen der Wurzel und einem zufällig gewählten Knoten in $10\%, 20\%, \ldots, 90\%, 100\%$ Tiefe des initialen BFS-Baumes.

Die Experimente zeigten, dass die Vorverarbeitung für die dynamische Breitensuche aufwendiger ist als für das statische BFS. Dies liegt ursächlich in den $p = O(log(B))$ statt nur einer Phase der Clus-

terung begründet. Die Breitensuche nach Einfügung einer Kante kann jedoch bei einer Vielzahl der Tests dank geringerer I/O-Zahl deutlich schneller durchgeführt werden. Die konkreten Testergebnisse sind [4] zu entnehmen.

Kapitel 4

Löschen von Kanten bei dynamischer Breitensuche im Externspeicher

Basierend auf der bereits bestehenden Implementierung für die dynamische Breitensuche im Externspeicher, die bislang nur das Hinzufügen von Kanten ermöglicht, wurde das Löschen von Kanten ergänzt. Durch diese Funktionserweiterung bildet der Code nunmehr das vollständige Spektrum der dynamischen Breitensuche ab. Um dieses Ziel zu erreichen, wurden für das Löschen von Kanten zunächst Algorithmen entwickelt und theoretisch untersucht, bevor die Implementierung erfolgte.

4.1 Algorithmenentwicklung

Im Weiteren gehen wir davon aus, dass im jeweils betrachteten Durchlauf einer dynamischen Breitensuche die Kante $e_c = \{v_1, v_2\}$ gelöscht wurde. Das gleichzeitige Löschen mehrerer Kanten lässt sich für die theoretische Betrachtung durch eine Abfolge von Einzellöschungen abbilden. Die Level der Knoten v_1 bzw. v_2 im BFS-Baum vor Löschen der Kante werden im Folgenden mit $level_{v_1}$ und $level_{v_2}$ bezeichnet.
Ebenso wie beim Einfügen von Kanten (s. Kapitel 3) kann man beim Löschen einige Fälle unterscheiden. Diese unterscheiden sich wesentlich hinsichtlich des entstehenden Aufwandes zur Neuberechnung des BFS-Baumes. Folgende, für die Algorithmenentwicklung relevant unterscheidbare Fälle, die in einem Durchlauf der dynamischen Breitensuche auftreten können, wurden identifiziert:

1 Gelöschte Kante zwischen zwei Knoten desselben BFS-Level

Verläuft die gelöschte Kante e_c zwischen zwei Knoten, die demselben BFS-Level angehören, so hat diese Löschung keine Auswirkung auf das Ergebnis der Breitensuche. Da beide Knoten im selben BFS-Level $level_{v_1} = level_{v_2}$ liegen, müssen beide Knoten einen Nachbarn im BFS-Level $level_{v_1,v_2} - 1$ besitzen. Daher verändert sich durch das Löschen der Kante nichts an der Leveleinordnung beider Knoten. Daraus folgt, dass sich auch für den restlichen BFS-Baum keinerlei Änderung ergibt.

2 Gelöschte Kante zwischen zwei Knoten, die mehr als ein BFS-Level auseinander liegen

Dieser Fall kann in ungerichteten Graphen, wie sie im Rahmen dieser Arbeit betrachtet werden (s. Kapitel 2), nicht vorkommen. In gerichteten Graphen wäre dieser Fall hingegen möglich und müsste gesondert gelöst werden.

3 Gelöschte Kante zwischen zwei Knoten, die in benachbarten Leveln liegen

Liegen die beiden Knoten, die durch Kante e_c verbunden waren, in benachbarten BFS-Leveln, so lassen sich wiederum weitere Fälle unterscheiden. Sei für diese Fälle der Knoten v_1 derjenige Knoten, der sich in einem kleineren BFS-Level befindet, also $level_{v_1} < level_{v_2}$. Für Knoten v_1 ändert sich durch die Löschung der Kante das BFS-Level nicht. Es gilt Für den Knoten v_2 und von ihm im BFS-Baum abhängige Knoten muss ggf. ein neues BFS-Level bestimmt werden. Hierbei können folgende mögliche Konstellationen unterschieden werden:

3.1 Isolation eines Knotens

Besitzt Knoten v_2 außer der Kante e_c im Zustand G_{i-1} des Graphen keine weiteren ein- bzw. ausgehenden Kanten, hat also $Grad_{i-1}(v_2) = 1$, so wurde durch die Löschung von e_c dieser Knoten isoliert. Dies bedeutet, dass der Knoten nicht mehr zur Komponente C_w der Wurzel w der Breitensuche gehört und somit durch die Breitensuche von w aus nicht mehr erreicht wird. Der Knoten v_2 wird aus dem BFS-Baum entfernt; weitere Änderungen ergeben sich nicht.

Eine besondere Betrachtung erfolgt im Falle der Isolation der Wurzel w. In diesem Fall handelt es sich nicht um den Knoten v_2, der zu betrachten ist, sondern den Knoten v_1. Immer wenn $v_1 = w$ gilt, muss geprüft werden, ob w neben v_2 noch weitere Nachbarn besitzt. Diese Feststellung wird getroffen indem geprüft wird, ob $Grad_{i-1}(w) > 1$ oder $|\{v|v \in V, d_{i-1}(v) = 1\}| > 1$ gilt. Ist dies nicht der Fall, so wurde die Wurzel durch die Löschung der Kante e_c isoliert, bzw. alle anderen Knoten aus C_w entfernt. Der BFS-Baum besteht in diesem Fall also nur noch aus der Wurzel und eine weitere Untersuchung ist auf dieser Basis nicht sinnvoll. Eventuell könnte nun eine neue Wurzel w bestimmt werden, um für eine der noch zusammenhängenden Komponenten des Graphen mit der Breitensuche fortzufahren. Dies erfordert allerdings einem Neuanstoß der dynamischen Breitensuche mit einer erneuten Initialisierung (initiale Breitensuche, Clusterung, Sortierung der Adjazenzlisten), sodass es sich im Wesentlichen um eine vollständig neue dynamische Breitensuche handelt.

3.2 Bestehende Kante in vorhergehendes BFS-Level

Besitzt Knoten v_2 nach Löschung von e_c noch mindestens eine Kante, die ihn mit einem Knoten aus dem vorhergehenden BFS-Level $level_{v_2} - 1 = level_{v_1}$ verbindet, so ändern sich die BFS-Level der Knoten nicht. Es existiert also weiterhin ein kürzester Weg von Wurzel w zu v_2, für dessen Länge $d_{i-1}(v_2) = d_i(v_2) = level_{v_2}$ gilt. Der kürzeste Weg zu Knoten v_2 führt nun lediglich über eine andere Knotenabfolge.

3.3 Bestehende Kante in das gleiche BFS-Level

Besitzt der Knoten v_2 keine Kante in ein vorhergehendes BFS-Level, aber mindestens eine Kante, die ihn mit einem Knoten aus dem eigenen BFS-Level $level_{v_2}$ verbindet, so erhöht sich das BFS-Level von v_2 um 1. Dies wiederum kann für einige Knoten, die in nachfolgenden BFS-Leveln liegen und deren kürzester Weg über v_2 führt, ebenfalls zu einer Anpassung ihrer BFS-Level führen.

Existieren in G_{i-1} von w zu beliebigem Knoten $v_k \in V$ $l \geq 1$ kürzeste Wege der Länge $d_{i-1}(v_k)$, so muss überprüft werden, ob v_2 in jedem der l kürzesten Wege enthalten ist. Ist dies nicht der Fall, so existiert mindestens ein kürzester Weg der Länge $d_{i-1}(v_k)$, der durch die Leveländerung von v_2 unverändert ist; es gilt also $d_{i-1}(v_k) = d_i(v_k)$. Ist v_2 jedoch in jedem der l kürzesten Wege enthalten, so gilt $d_i(v_k) = d_{i-1}(v_k) + 1$. Der Grad der Anpassung ist jedoch minimal; das Level kann sich für jeden Knoten v_k maximal um 1 erhöhen.

Das Vorgehen wird durch den folgenden Pseudocode abgebildet:

Algorithm 2: Pseudocode für lineare BFS-Level-Anpassung

```
1  int aktuellesLevel = level_{v_2} + 1;
2  int [] verboteneKnoten = {v ∈ V | d_{i-1}(v) = level_{v_2}} \ {v_2};
3  int [] aktuelleKnoten = {v_2};
4  int [] naechsteKnoten = {};
5  while !aktuelleKnoten.empty() do
6      for all nodes v in aktuelleKnoten do
7          d_i(v) =aktuellesLevel;
8          naechsteKnoten.add(v.getNeighbours());
9      naechsteKnoten = naechsteKnoten \(aktuelleKnoten ∪ verboteneKnoten);
10     for all nodes v in naechsteKnoten do
11         for all nodes u in v.getNeighbours() do
12             if d_{i-1}(u) = aktuellesLevel − 1 then
13                 naechsteKnoten.remove(v);
14     verboteneKnoten = aktuelleKnoten;
15     aktuelleKnoten = naechsteKnoten;
16     naechsteKnoten = {};
17     aktuellesLevel++;
```

Um die BFS-Level-Anpassungen aller betroffenen Knoten zu bestimmen, wird zunächst das neue BFS-Level von Knoten v_2 mit $level_{v_2} + 1$ festgesetzt. Um die weiteren Anpassungen zu ermitteln, ist es notwendig, eine Breitensuche in höhere Level ausgehend vom Knoten v_2 durchzuführen. Alle gefundenen Nachbarn im nachfolgenden BFS-Level müssen darauf überprüft werden, ob sie außer

der Kante zu v_2 noch eine andere Kante zu einem nicht im Level angepassten Knoten im vorhergehenden Level besitzen. Ist dies nicht der Fall, so muss ihr eigenes Level entsprechend ebenfalls um eins erhöht werden. Diese Breitensuche wird rekursiv auf allen Knoten mit verändertem BFS-Level durchgeführt, bis keiner der gefundenen Nachbarn mehr angepasst wird.

In Abbildung 4.1 wird eine solche Anpassung beispielhaft für das erste BFS-Level dargestellt. Nachdem die rote Kante $\{v_1, v_2\}$ gelöscht wurde, erhöht sich das BFS-Level von v_2 um 1, da noch Nachbarn im eigenen $level_{v2}$ vorhanden sind. Für die Nachbarn von v_2 in nachfolgenden Leveln muss entschieden werden, ob ihr BFS-Level ebenfalls angepasst werden muss. Für v_j trifft dies nicht zu, da noch ein anderer Nachbar als v_2 im vorhergehenden Level existiert. v_k hingegen hat keinen weiteren Nachbarn außer v_2 in diesem Level, so dass das BFS-Level von v_k ebenfalls um 1 erhöht werden muss. Somit setzt sich die Breitensuche rekursiv auf den Nachbarn von v_k, die sich in höheren BFS-Leveln befinden, fort.

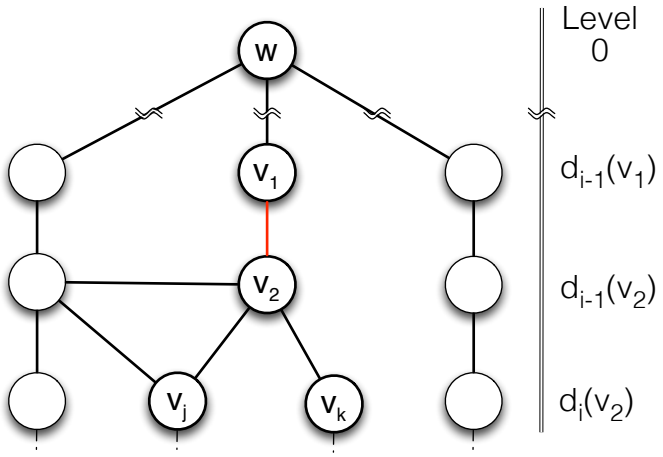

Abbildung 4.1: Beispiel für eine Levelerhöhung um maximal 1

3.4 Bestehende Kante in ein nachfolgendes BFS-Level

Fällt die Kantenlöschung unter keine der vorangegangenen Konstellationen, so hat Knoten v_2 zwar noch weitere ein- bzw. ausgehende Kanten außer e_c, diese führen jedoch alle nur in das nachfolgende BFS-Level. In diesem Fall ist die Veränderung am BFS-Baum und an den BFS-Leveln nicht mehr einfach ersichtlich und erfordert einen deutlich höheren Aufwand als alle anderen Fälle.

Die Neuberechnung des BFS-Baumes verläuft in zwei Stufen. Zunächst erfolgt eine Voruntersuchung ausgehend von Knoten v_2, um zu überprüfen, ob durch die Kantenlöschung eine Graphkomponente C_{v_2} isoliert wurde. Ist dies nicht der Fall, so wird im Rahmen der Voruntersuchung das BFS-Level identifiziert, ab dem die Neuberechnung des BFS-Baumes erfolgen muss. Das genaue Vorgehen wird in den nächsten beiden Abschnitten erläutert.

4.1.1 Voruntersuchung auf Verbundenheit und Verbindungslevel

Ziel der Voruntersuchung ist die Feststellung, ob durch die Löschung der Kante e_c eine Graphkomponente C_{v_2} isoliert wurde oder alternativ in G_i immer noch ein Weg von v_2 zu w existiert. Um dies festzustellen, wird eine Breitensuche ausgehend von v_2 in seinem alten Level gestartet. Diese Breitensuche hat zwei natürliche Abbruchbedingungen:

- Sobald in einem der nachfolgenden Level ein Knoten v_j besucht wird, der eine Kante zu einem bis dahin in dieser Breitensuche noch nicht besuchten Knoten v_k in einem kleineren BFS-Level $d_{i-1}(v_j) - 1$ hat, kann die Breitensuche abgebrochen werden. Der Knoten v_k muss mindestens einen kürzesten Weg der Länge $d_{i-1}(v_k) = level_{v_j} - 1$ zur Wurzel w der ursprünglichen Breitensuche besitzen, der nicht über den Knoten v_2 führt.

  Dies liegt darin begründet, dass der Knoten v_k in der Breitensuche ab v_2 (die bis Level $d_{i-1}(v_j)$ stets in jeweils höhere BFS-Level der ursprünglichen Breitensuche führt) nur über den Knoten v_j erreicht wird, der in Level $d_{i-1}(v_k) + 1$ liegt. Somit hätte ein potentieller kürzester Weg von w zu v_k über v_2 in G_{i-1} mindestens die Länge $d_{i-1}(v_2) + (d_{i-1}(v_k) + 1 + 1 - d_{i-1}(v_2)) = d_{i-1}(v_k) + 1 + 1$. Da v_k jedoch in $d_{i-1}(v_k)$ liegt, muss ein Weg dieser Länge existieren, der somit nicht über v_2 führen kann. Somit ist sichergestellt, dass mit der Löschung von e_c keine Graphkomponente abgetrennt wurde, da über den Knoten v_j noch ein Weg $w, \ldots, v_k, v_j, \ldots v_2$ von der Wurzel des BFS-Baumes zu Knoten v_2 besteht. Eine solche Kante $\{v_j, v_k\}$ kann dabei in zwei möglichen Knotennachbarschaftsbeziehungen zum von v_2 aus durchlaufenen Teilgraphen auftreten, die in der Abbildung 4.2 visualisiert werden. Entweder besitzt einer der besuchten

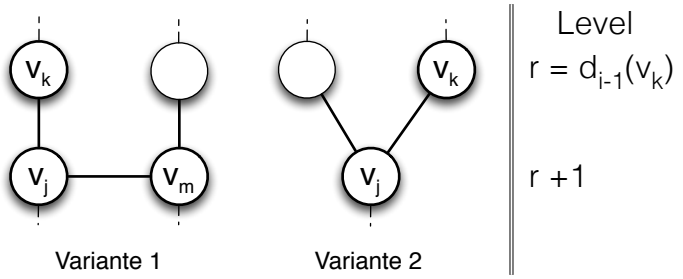

Abbildung 4.2: Mögliche Knotennachbarschaftsbeziehungen zum Verbindungslevel

  Knoten direkt einen noch nicht besuchten Nachbarn im vorangegangenen Level, oder er besitzt einen Nachbarn im eigenen Level, der wiederum einen Nachbarn im vorangegangenen Level hat.

- Existieren in der Suche keine weiteren zu besuchenden Knoten mehr und hat keiner der besuchten Knoten die zuvor genannte Abbruchbedingung erfüllt, so wurde mit der Löschung von Kante e_c in G_i eine Graphkomponente C_{v_2} abgetrennt. Es besteht kein Weg von Knoten v_2 mehr zur Wurzel w der ursprünglichen Breitensuche.

Um die Voruntersuchung durchzuführen, wird eine leicht abgewandelte Version der Breitensuche zur linearen BFS-Level-Anpassung (s. Algorithm 2) verwendet:

Algorithm 3: Pseudocode für Voruntersuchung

```
1  bool verbindungGefunden = false;
2  int verbindungsLevel = ∞;
3  int aktuellesLevel = level_{v2} + 1;
4  int [] verboteneKnoten = {v2};
5  int [] aktuelleKnoten = {};
6  for all nodes u in v2.getNeighbours() do
7      if d_{i-1}(u) > level_{v2} then
8          aktuelleKnoten.add(u);
9  int [] naechsteKnoten = {};
10 while !aktuelleKnoten.empty() && !verbindungGefunden do
11     for all nodes v in aktuelleKnoten do
12         if d_{i-1}(v) ≤ aktuellesLevel −2 then
13             verbindungGefunden = true;
14             verbindungsLevel = d_{i-1}(v);
15         naechsteKnoten.add(v.getNeighbours());
16     naechsteKnoten = naechsteKnoten \(aktuelleKnoten ∪ verboteneKnoten);
17     verboteneKnoten = aktuelleKnoten;
18     aktuelleKnoten = naechsteKnoten;
19     naechsteKnoten = {};
20     aktuellesLevel++;
21 return verbindungGefunden;
```

Wenn kein Weg von v_2 zur Wurzel der ursprünglichen Breitensuche existiert, so hängen alle in der Graphkomponente C_{v_2} enthaltenen Knoten nicht mehr mit der Graphkomponente C_w der Wurzel zusammen. Dies bedeutet, dass alle Knoten $v \in C_{v_2}$ aus dem BFS-Baum zu entfernen sind. Um zu diesem Zweck nicht ein zweites Mal eine Breitensuche auf C_{v_2} durchführen zu müssen, wird die Löschung aller besuchter Knoten im Rahmen der Voruntersuchung bereits vorgemerkt. Tritt der Fall der Abtrennung der Graphkomponente ein, so muss dieses Ergebnis nur übernommen werden. Greift jedoch das andere Abbruchkriterium, so wird dieses vorläufige Ergebnis verworfen und gemäß den im nächsten Abschnitt beschriebenen Schritten vorgegangen.

4.1.2 Neuberechnung der BFS-Level ab Verbindungslevel

Ist das Ergebnis der Voruntersuchung, dass keine Graphkomponente durch die Löschung von Kante e_c abgetrennt wurde, so existiert noch ein Weg von Knoten v_2 zur Wurzel w. Die Änderungen im BFS-

Baum müssen durch eine weitere Breitensuche angepasst werden. Diese Breitensuche kann von dem in der Voruntersuchung ermittelten Verbindungslevel, also Level $d_{i-1}(v_k)$, das den dort gefundenen Knoten v_k enthält, gestartet werden. Für Level $< d_{i-1}(v_k)$ gilt, dass aus diesen nur solche Knoten in höhere Level verschoben werden, die durch die Voruntersuchungs-Breitensuche bereits besucht wurden. Die betroffenen Knoten werden aber durch die neue Breitensuche ab dem Verbindungslevel explizit erneut besucht.

Ausgangsknoten der Breitensuche sind alle Knoten des Level $d_{i-1}(v_k)$ mit Ausnahme derjenigen Knoten dieses Levels, die bereits in der Voruntersuchungs-Breitensuche besucht wurden. Ab diesem Level wird nun eine normale Breitensuche durchgeführt. Analog zum Vorgehen beim Einfügen von Kanten werden hierzu Hotpools verwendet. Der lineare Hotpool H hält immer die Adjazenzlisten der nächsten α_1 Level vor. Da die Breitensuche ab dem Verbindungslevel jedoch sowohl Knoten aus steigender wie fallender BFS-Level-Zahl besucht, wird zusätzlich der Cluster-Hotpool benötigt, in den Cluster von Knoten geladen werden, die nicht im linearen Hotpool gefunden wurden. Die Clustergröße wird durch α_2 bestimmt und analog zur Implementierung für das Einfügen von Kanten bei Überschreiten der tolerierten I/O-Grenze vergrößert.

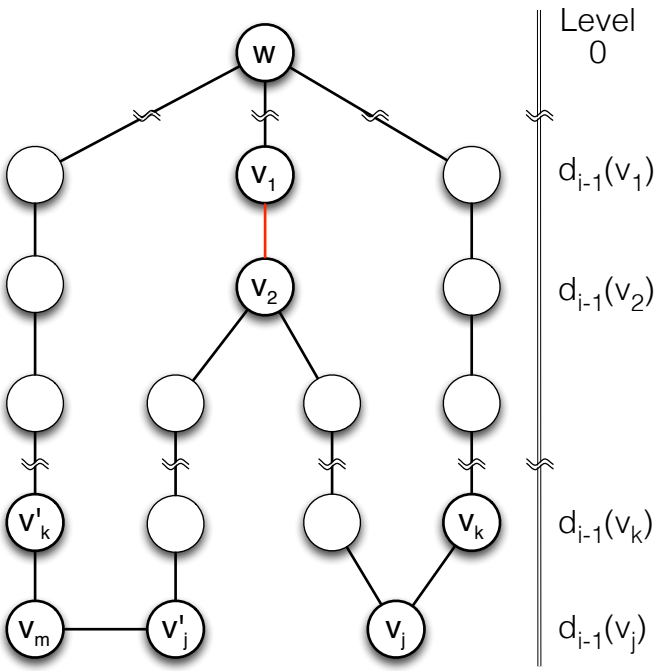

Abbildung 4.3: Mehrere Verbindungslevel für die Neubestimmung der BFS-Level auf G_{i-1} mit gelöschter roter Kante e_c

Im Unterschied zur Breitensuche beim Einfügen von Kanten können besuchte Knoten, deren Level sich nicht verändert, nicht als Endpunkt der Breitensuche angesehen werden. Seien v_j, v'_j beide Knoten des Levels $d_{i-1}(v_j) = d_{i-1}(v'_j)$, die eine Entfernung von $2 \cdot (d_{i-1}(v_j) - d_{i-1}(v_2))$ voneinander haben. Sie könnten also beide in jeweils einer eigenen Liste von Knoten an v_2 hängen. Sei v_j der Knoten, dessen Nachbar v_k in der Voruntersuchung zum Abbruch und Bestimmung des Verbindungslevels

$d_{i-1}(v_k)$ geführt hat. Sei weiterhin v'_k ebenfalls im Verbindungslevel $d_{i-1}(v_k)$ ohne eine Kante zu einem in der Voruntersuchung besuchten Knoten des Levels $d_{i-1}(v_j)$. Existiert nun ein Knoten v_m in Level $d_{i-1}(v_j)$, der sowohl eine Kante zu v'_k wie auch zu v'_j hat, so wird dieser Knoten in der Breitensuche ausgehend von v'_k besucht. Das BFS-Level von v_m ändert sich nicht. Würde man v_m daher als Endpunkt der Breitensuche ansehen, so würde v'_j sein BFS-Level möglicherweise erst über die Suche von v_k über $v_j, \ldots, v_2, \ldots v'_j$ erhalten, also ein BFS-Level $d_i(v'_j) = d_{i-1}(v'_j) + (2 \cdot (d_{i-1}(v_j) - d_{i-1}(v_2)))$, was nicht seinem eigentlichen BFS-Level $d_i(v'_j) = d_{i-1}(v_m) + 1 = d_{i-1}(v'_j) + 1$ entsprechen würde. Der entsprechende Sachverhalt wird für den Graph G_{i-1} auch in Abbildung 4.3 dargestellt.
Dies bedeutet, dass in der Breitensuche zur Anpassung im schlimmsten Fall viele Knoten besucht werden, deren Level sich nicht verändert und deren Besuch keinen Mehrwert für die dynamische Anpassung bringt. Die praktischen Auswirkungen zeigen sich in den Experimenten, s. Kapitel 5. In Abbildung 4.4 wird das Prinzip der Breitensuche ab Verbindungslevel zur Anpassung der BFS-Level dargestellt. Die blauen Pfeile kennzeichnen die Ausbreitung dieser Breitensuche ab dem Verbindungslevel mit Knoten v_k (waagrechter blauer Strich). Dreiecke stellen Knotenbereiche des Graphen dar, wobei graue Dreiecke durch die dynamische Breitensuche nicht besucht werden, orange Dreiecke notwendigerweise besucht werden und Knoten in braunen Dreiecken zwar besucht werden, dies aber keinen Mehrwert bietet. $v_{k'}$ steht stellvertretend für weitere Verbindungslevel.

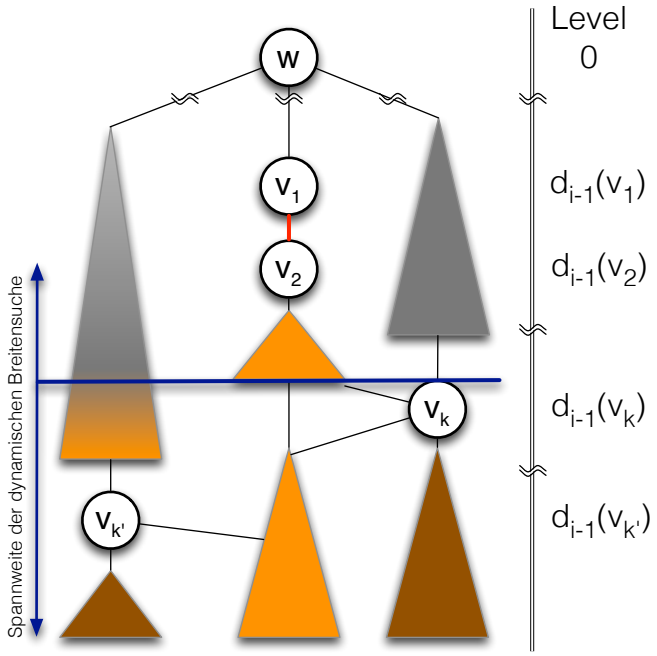

Abbildung 4.4: Neubestimmung der BFS-Level ausgehend vom Verbindungslevel $d_{i-1}(v_k)$

4.2 Theoretische Betrachtung

Im folgenden Abschnitt wird gezeigt, dass die obere I/O-Schranke für die dynamische Breitensuche im Externspeicher beim Löschen von Kanten dieselbe ist wie beim Einfügen von Kanten. Die Her-

angehensweise beim Nachweis liegt dabei in der Umkehrung des Beweises aus [13], s. auch Kapitel 3. Hierzu werden zunächst die möglichen auftretenden Fälle unterschieden. Um die Unterscheidung bezogen auf die Löschung der aktuellen Kante $e_c = \{v_1, v_2\}$ zu treffen, wird ein Algorithmus zur Feststellung der Verbundenheit der Graphkomponenten angewandt. Mögliche Ergebnisse sind:

1. Fall A : Löschen einer Kante führt zum Abtrennen einer bisher verbundenen Graphkomponente
2. Fall B : Löschen einer Kante ohne Veränderung der Graphkomponente der Wurzel w

Fall A

Stellt man dabei fest, dass es sich bei e_c um eine Kante des Falls A handelt, so ist die Komponente C_{v_2}, in der sich v_2 befindet, von der Komponente C_w der Wurzel w getrennt worden. Somit werden alle Knoten der Komponente C_{v_2} aus dem BFS-Baum entfernt, was durch eine Breitensuche auf der Komponente ausgehend von v erfolgen kann. Die Zahl der I/O ergibt sich aus der Berechnung der Verbundenheit mit maximal $O(sort(n) \cdot log(B)$ I/O sowie $O(n_v \cdot sort(n)$ I/O für die Breitensuche auf C_{v_2}.
In $m' = \Theta(n)$ Experimenten kann Fall A maximal $n-1$ Mal auftreten, woraus sich als obere Grenze für die notwendigen I/O $O(n \cdot sort(n) \cdot log(B))$ ergibt.

Fall B

Handelt es sich bei der Löschung jedoch um einen Fall B, so müssen die BFS-Level der Knoten in der bestehenden Komponente C_w neu berechnet werden. Analog zum Einfügen von Kanten wird der nach Knotennummern sortierte Hotpool H benutzt, in den die Adjazenzlisten eines Knotens w genau dann geladen werden, wenn das BFS-Level $max\{0, d_{i-1}(w) - \alpha\}$ neu berechnet wird.

Einfache Fälle B

Eine Untermenge des Fall B sind die oben beschriebenen Fälle **1 Gelöschte Kante zwischen zwei Knoten desselben BFS-Level**, **3.1 Isolation eines Knotens** und **3.2 Bestehende Kante in vorhergehendes BFS-Level**. In diesen Fällen sind keine Änderungen von BFS-Leveln außer ggf. bei Knoten v_2 notwendig. Somit sind hier lediglich $O(1)$ I/O erforderlich.

I/O-intensive Fälle B

Im oben als **3.3 Bestehende Kante in das gleiche BFS-Level** beschriebenen Fall des verbleibenden Nachbarn im selben BFS-Level ist eine Erhöhung des BFS-Levels von $O(n)$ Knoten von maximal 1 notwendig. Es gilt also $\Delta d_i(v) \leq 1, v \in V_i$. Das bedeutet, dass für α mit $1 < \alpha < \sqrt{B}$ die Änderung der BFS-Level unter alleiniger Nutzung des sequentiellen Hotpools möglich ist. Die Gesamtzahl der I/O ist also – analog zum Einfügen von Kanten mit $\Delta d_i(v) < \alpha$ für alle Knoten – geringer als bei MM_BFS.
Hat der Knoten v_2 außer v_1 nur mindestens eine Kante zu einem Knoten im nachfolgenden Level, so müssen die für den Fall **3.4 Bestehende Kante in ein nachfolgendes BFS-Level** beschriebenen Vorgehensweisen angewandt werden. Dabei muss davon ausgegangen werden, dass für manche Knoten $\Delta d_i(v) > \alpha$ gilt, da die Breitensuche sowohl in steigender als auch fallender Richtung gleichzeitig

verläuft. Insbesondere kann die Breitensuche auch Knoten aus BFS-Leveln in nicht monotoner Folge besuchen. Wird ein Knoten in H nicht gefunden, so werden die Adjazenzlisten des Clusters des Knotens nachgeladen. Wie beim Einfügen von Kanten ist die Clustergröße auf $\frac{\alpha}{4}$ festgesetzt und die Zahl der unstrukturierten Clusterzugriffe auf maximal $\alpha \cdot \frac{n}{B}$ begrenzt. Sobald die Grenze überschritten wird, wird α um den Faktor 2 vergrößert und eine neue Clusterung mit verdoppelter Clustergröße, sowie die Breitensuche mit doppelter Hotpoolgröße durchgeführt. Dies resultiert im Versuch $j, j \geq 1$ in einem α_j mit $\alpha_j := 32 \cdot 2^j$ und somit Clustergröße $\frac{\alpha_j}{4} = 8 \cdot 2^j$, sowie einem Hotpoolumfang von α_j BFS-Leveln. Es gilt $j \leq log(\sqrt{B})$, sodass die Zahl der Fehlversuche mit zu kleinem α auf $O(log(B))$ begrenzt ist. Bei genügend großem j werden alle Adjazenzlisten der n Knoten geladen, womit die Zahl der I/O maximal dem statischen MM_BFS entspricht. Dank der Verwendung von Zeitstempeln kann jede Adjazenzliste nach höchstens $\alpha_j = O(2^j)$ betrachteten BFS-Leveln wieder aus H entfernt werden.

Gesamtzahl der I/O

Zur Ermittlung, dass es sich nicht um Fall A handelt (Abtrennen einer bisher verbundenen Graphkomponente), werden $O(sort(n) \cdot log(B))$ I/O benötigt. In der BFS-Phase sind die I/O asymptotisch begrenzt durch die Zahl der I/O im letzten, erfolgreichen Versuch $j : O(2^j \cdot \frac{n}{B})$. Die weiteren im Algorithmus je Versuch $j, j = O(log(B))$ vorgesehenen Tätigkeiten sind in $O(sort(n))$ I/O möglich. Dies umfasst die Clusterung des Graphen, die BFS-Level- und Cluster-sequentielle Sortierung der Adjazenzlisten sowie die Initialfüllung des Hotpools H. Somit ist die Zahl der I/O für alle Löschungen vom Typ Fall B im j-ten Versuch der Breitensuche begrenzt durch $O(2^j \cdot \frac{n}{B} + sort(n) \cdot log(B))$, wenn die Gesamtzahl der Löschungen durch $\Theta(n)$ begrenzt ist, der Graph also dünn bleibt.

Beschränkte Anzahl von Ereignissen mit *hohem j-Ertrag*

Für die Gesamtzahl der BFS-Level-Veränderungen von Knoten $v \in V$ nach Löschung der i-ten Kante wird $\Delta D_i = |D_{i-1} - D_i|$ mit $D_i = \sum_{v \in V \setminus \{w\}} (d_i(v))$ definiert. Wird ein Cluster der Größe $\mu = \frac{\alpha}{4}$ explizit nach H geladen, so gilt für mindestens einen Knoten in diesem Cluster $\Delta d_i(v) > \alpha$. Da jedoch per Definition der Durchmesser des Clusters auf μ begrenzt ist, gilt für alle weiteren Knoten v' des Clusters $\Delta d_i(v') > \alpha - 2 \cdot \mu \geq \frac{\alpha}{2}$. Dies lässt sich daraus ableiten, dass Knoten v seine Level um mehr als α verändert und gleichzeitig alle Knoten v' desselben Clusters vorher wie hinterher maximal μ Kanten von v entfernt sein können. Die Zahl der Knoten, die ihr Level um mehr als $\frac{\alpha}{2}$ ändern, lässt sich aus dem letzten fehlgeschlagenen Versuch $j-1$ einer Löschung i mit erfolgreichem j-ten Versuch ermitteln. Im Versuch $j-1$ wurden bis zur Verdopplung von α bereits $\alpha_{j-1} \cdot \frac{n}{B}$ unterschiedliche Cluster geladen, d.h. $2^{j+4} \cdot \frac{n}{B}$ Cluster.

Unter Annahme der in [4] gezeigten deterministischen levelweisen hierarchischen Clusterung besitzt jedes Cluster $\Theta(\mu) = 2^{j+1}$ Knoten, d.h. im Versuch $j-1$ wurden bereits mindestens $2^{2 \cdot j+5}$ Knoten geladen. Für jeden der via Clusterzugriff geladenen Knoten gilt $\Delta d_i(v) \geq \frac{\alpha_{j-1}}{2} = 2^{j+3}$. Für einen im Versuch j erfolgreichen Löschversuch i gilt $\Delta D_i \geq 2^{3 \cdot j+8} \cdot \frac{n}{B} =: Y_j$, also ein Ereignis mit *hohem j-Ertrag*.

Für eine Reihe von $m' = \Theta(n)$ Löschungen von Kanten in einem dünnen Graphen ist die Zahl der

Ereignisse mit *hohem j-Ertrag* begrenzt. Im Fall der Löschung gilt: $0 < D_0 \leq D_1 \leq \cdots \leq D_{m'} < n^2$. Damit ist die Zahl der Ereignisse mit *hohem j-Ertrag* auf $k = \frac{n^2}{Y_i} = \frac{n \cdot B}{2^{3 \cdot j+8}}$ begrenzt.

Abschluss

Die auftretenden Fälle B unterteilt man nun wiederum in die Fälle B_1, in denen die Breitensuche mit einem Wert $\alpha_j < B^{\frac{1}{3}}$ abgeschlossen wird und die maximal $\Theta(n)$ Mal auftreten können, sowie die Fälle B_2, in denen α_j höher liegt. Für die Fälle B_2 wurde oben gezeigt, dass diese höchstens $O\left(\frac{n \cdot B}{2^{3 \cdot j}}\right)$ Mal auftreten können. Die Zahl der I/O je Update beträgt in beiden Fällen, B_1 und B_2, $O(\alpha_j \cdot \frac{n}{B} + sort(n) \cdot log(B))$.

Die maximal $\Theta(n)$ Fälle B_1 mit $\alpha_j < B^{\frac{1}{3}}$ erzeugen somit höchstens

$$O\left(n\left(\frac{n}{B^{\frac{2}{3}}} + sort(n) \cdot log(B)\right)\right) \text{ I/O.}$$

Die maximal $O\left(\frac{n \cdot B}{2^{3 \cdot j}}\right)$ auftretenden Fälle B_2 mit $\alpha_j = B^{\frac{1}{3}} \cdot 2^j$ erzeugen somit unter Anwendung der Booleschen Ungleichung

$$O\left(\left(\sum_{g \geq 0}\left(\frac{n \cdot B}{(B^{\frac{1}{3}} \cdot 2^g)^3} \cdot \frac{B^{\frac{1}{3}} \cdot 2^g \cdot n}{B}\right)\right) + n \cdot sort(n) \cdot log(B)\right) \text{ I/O} =$$
$$O\left(n\left(\frac{n}{B^{\frac{2}{3}}} + sort(n) \cdot log(B)\right)\right) \text{ I/O.}$$

Hiermit wird die Aussage aus [13] bestätigt, dass Löschungen sich analog zu Einfügungen verhalten. Somit gilt mit hoher Wahrscheinlichkeit die gezeigte I/O-Grenze von $O\left(n\left(\frac{n}{B^{\frac{2}{3}}} + sort(n) \cdot log(B)\right)\right)$ I/O für $\Theta(n)$ dynamische Einfügungen und Löschungen von Kanten in dünnen Graphen.

4.3 Implementierung

Die Implementierung der in Kapitel 4.1 beschriebenen Algorithmen erfolgte basierend auf dem im Rahmen von [4] erzeugten und in Kapitel 3.2 beschriebenen Code. Die Implementierung erfolgt in C++ unter Nutzung der STXXL-Bibliothek zum Management des Externspeichers. Der Sourcecode liegt dem Lehrstuhl für Algorithm Engineering vor.

Neu hinzugefügt wurden im Rahmen dieser Arbeit die Funktionalitäten zur dynamischen Breitensuche nach der Löschung von Kanten. Die inhaltliche Arbeit teilt sich dabei auf in die Bereiche Experimentaldaten zur Durchführung von Experimenten, Implementierung der Algorithmen zur Breitensuche nach Löschung von Kanten und Darstellung der gelöschten Kanten in Graphstrukturen.

4.3.1 Experimentaldaten

Um Experimente durchführen zu können war es notwendig, einige Codeerweiterungen vorzunehmen. Implementiert wurde hier zum einen ein Auswahlalgorithmus zur randomisierten Löschung von Kanten. Dabei wird zufällig ein Knoten des Graphen, der in der Komponente C_w der Wurzel liegt, ausgewählt und die Kante zu einem seiner Nachbarn gelöscht. Zusätzlich wurde im Code für Experimente auch die Möglichkeit geschaffen zufällig zwischen der Löschung und Einfügung einer Kante je Iteration auszuwählen.

Zur Durchführung der spezifischen Untersuchung I/O-intensiver Fälle wurde der Code zudem dahingehend erweitert, dass nacheinander in $10\%, 20\%, ..., 90\%.100\%$ Tiefe des initialen BFS-Baumes ausgehend von der Wurzel eine Kante eingefügt wird, der neue BFS-Baum berechnet wird und eben jene Kante in der darauffolgenden Iteration wieder gelöscht wird. Hierdurch wird ein hoher BFS-Level-Änderungsaufwand künstlich erzeugt.
Zur Dokumentation der Ergebnisse wurde der Code abschließend um eine Statistikfunktion erweitert, um bei Experimenten mit vielen Iterationen eine maschinenlesbare Auswertung der Fallzahlen, I/O und Zeit zu erhalten.

4.3.2 Breitensuche nach Löschung von Kanten

Kernstück der Implementierung sind die in Kapitel 4.1 beschrieben Algorithmen. Hierzu wurde zunächst eine Unterscheidung in die dort genannten Fälle implementiert. Für die einfachen Fälle **1 Gelöschte Kante zwischen zwei Knoten desselben BFS-Level**, **3.1 Isolation eines Knotens** und **3.2 Bestehende Kante in vorhergehendes BFS-Level** wird lediglich die betreffende Kante gelöscht und im Falle der Isolation der betroffene Knoten aus dem BFS-Baum entfernt.

Für den Fall **3.3 Bestehende Kante in das gleiche BFS-Level** kann eine ähnliche Vorgehensweise wie im Einfügefall bei einem Delta $< \alpha$ gewählt werden. Hier werden unter alleiniger Nutzung des linearen Hotpools die BFS-Level betroffener Knoten um 1 erhöht, bis kein solcher Knoten mehr gefunden wird.

Der komplexeste Fall **3.4 Bestehende Kante in ein nachfolgendes BFS-Level** wird, wie in Kapitel 4.1 beschrieben, mit dem zweistufigen Vorgehen umgesetzt. Die Voruntersuchung nutzt ebenfalls nur den linearen Hotpool, um ein Verbindungslevel zu finden. Gleichzeitig wird die Abtrennung der Graphkomponente als vorläufiges Ergebnis vorbereitet. Wird keine Verbindung gefunden, so wird das vorläufige zum endgültigen Ergebnis und die Graphkomponente aus dem BFS-Baum entfernt. Falls doch eine Verbindung vorhanden ist, so wird das vorläufige Ergebnis verworfen und ab dem gefundenen Verbindungslevel die Breitensuche zur Änderung der BFS-Level durchgeführt. Dabei wird ab diesem Level der lineare Hotpool H für die in den jeweils nächsthöheren Leveln auftretenden Knoten verwendet. Der Cluster-Hotpool wird für alle dort nicht enthaltenen Knoten benötigt, insbesondere solche, die ein kleineres Level als das Verbindungslevel hatten.
Da in den Leveln, die größer sind als das Verbindungslevel, alle Knoten besucht werden müssen (Erläuterung s. Kapitel 4.1), werden mindestens *alle* Knoten von $|\text{Verbindungslevel} - level_{v_2}|$ Leveln besucht. Im schlechtesten Fall werden allerdings zusätzlich die Knoten von weiteren $|\text{maximales BFS-Level} - level_{v_2}|$ Leveln besucht. Dies kann im schlechtesten Fall dazu führen, dass nahezu alle Knoten des Graphen im Laufe der erneuten Breitensuche besucht werden müssen, d.h. sie enstpricht einer statischen Breitensuche. Dieser Aufwand wurde auch in den Experimenten erkennbar.

Gegenüber den in s. Kapitel 4.1 vorgestellten Algorithmen mussten einige kleinere Anpassungen vorgenommen werden. So ist beispielsweise die im Pseudocode 2 genannte Untersuchung aller Nach-

barn eines Knoten auf deren BFS-Level mit einer hohen Anzahl von unstrukturierten Zugriffen auf die Datenstruktur zur Speicherung des jeweiligen BFS-Levels verbunden, weshalb hier zunächst alle Nachbarn geladen werden und die Prüfung erst in der nächsten Runde, wenn die entsprechenden Adjazenzlisten im Hotpool gefunden sind, durchgeführt wird.

Geplant war zusätzlich eine Option, dass die Voruntersuchung nach einer bestimmten Anzahl besuchter Knoten oder BFS-Level ohne Ergebnis abgebrochen wird und ab dem Abbruch-Level die Breitensuche zur Änderung der BFS-Level durchgeführt wird. Ziel dieses Vorgehens ist es, das doppelte Besuchen der Knoten unterhalb von v_2 aber überhalb des Verbindungslevels (sofern existent) durch Voruntersuchung und anschließende Breitensuche soweit wie möglich zu beschränken. Dieses Vorgehen wurde im Laufe der Implementierung jedoch verworfen, da für den Fall, dass die Graphkomponente C_{v_2} abgetrennt wurde wiederum zusätzlicher Aufwand entstehen würde, um diejenigen Knoten zu identifizieren, die in dieser Graphkomponente liegen. Da keine der vorhandenen Datenstrukturen diese Ermittlung mit geringem I/O-Aufwand ermöglicht, wäre also ohnehin eine Vollendung der Voruntersuchung notwendig. Aus diesem Grund wurde der Ansatz verworfen, die Voruntersuchung muss derzeit immer bis zu einem definitiven Ergebnis fortgeführt werden.

4.3.3 Darstellung gelöschter Kanten in Graphstrukturen

Ein weiterer Schritt zur Durchführung der Experimente bestand darin, dass die Änderungen von jeweiligen Graph G_{i-1} zum Graphen G_i persistiert werden mussten, um eine Folge von Löschungen oder eine Einfügung gefolgt von einer Löschung derselben Kante überhaupt darstellen zu können. Diese Vorverarbeitung für die nächste Iteration wird auch in der theoretischen Betrachtung mit $O(sort(n) \cdot log(B))$ I/O berücksichtigt.
Zunächst wurde die direkte Anpassung der tatsächlich betroffenen Kanten in den sortierten Adjazenzlisten implementiert. Bei Experimenten mit sehr großen Graphen zeigte sich jedoch schnell, dass dies bei einer großen Anzahl von Änderungen zu einem höheren Aufwand führt, als die gesamte Vorverarbeitung von Grund auf neu durchzuführen. Dementsprechend wurde letzterer Ansatz implementiert. Eine neue Clusterung der Knoten wird derzeit nicht durchgeführt.

Kapitel 5

Experimentelle Untersuchung der Implementierung

Um den entstandenen Code auf seine Leistungsfähigkeit hin zu untersuchen, wurden Experimente mit unterschiedlichen Datensätzen durchgeführt. Ziel der Untersuchung ist einerseits der Vergleich der Laufzeit mit der bestehenden Implementierung des Einfügens von Kanten aus [4] und der Laufzeit des statischen MM_BFS. Andererseits wurden Experimente auf realen Graphen durchgeführt, um bei zufälliger Kantenlöschung auszuwerten, welche der in Kapitel 4 dargestellten Fälle von Kantenlöschung wie oft auftreten und um eine realistische Aussage zur praktischen amortisierten Laufzeit zu erhalten.

5.1 Experimentalumgebung

Der Code wird mit dem Compiler GCC in der Version 5.3.1 im Modus C++11 unter Nutzung des Optimierungslevels 3 kompiliert. Dabei eingebunden wird die STXXL Bibliothek (s. [7]) in der Version 1.4.0. Ausgeführt werden die Tests auf zwei Umgebungen, die die in Tabelle 5.1 dargestellten Eigenschaften besitzen.

Eigenschaft	Umgebung 1	Umgebung 2
CPU	AMD A10-6800K Quad Core	AMD FX-4170 Quad Core
Takt	2 GHz	1,4 GHz
RAM	32 GB	16 GB
Festplatten	1 Festplatte für OS und logs 4 Festplatten mit je 500 GB für STXXL	1 Festplatte für OS und logs 4 Festplatten mit je 7 TB für STXXL
OS	Debian GNU/Linux amd64 ‘stretch’	Debian GNU/Linux amd64 ‘stretch’

Tabelle 5.1: Umgebungskonfiguration für Experimente

Als Vergleichszeiten für die statische Breitensuche im Externspeicher werden die Ergebnisse aus [4] herangezogen. Die Vergleichbarkeit ist gegeben, da die in Experimente auf einer ähnlichen Umgebung stattfanden.

5.2 Experimente

Als Experimentaldaten stehen die Originalgraphen aus [4] zur Verfügung. Dabei handelt es sich insgesamt um vier verschiedene Graphklassen. Zwei der Graphenklassen sind synthetisch erzeugt und besitzen jeweils einen Durchmesser von $\Theta(n)$ bzw. $\Theta(\sqrt{n})$. Somit haben sie auch eine entsprechende Anzahl von BFS-Leveln im Ausgangszustand G_0. Die Zahl der Knoten beträgt in beiden Graphen $n = 2^{28}$, die Zahl der Kanten liegt bei $m = 0,9 \cdot 10^9$ im $n-Level$ Graph und $m = 1,1 \cdot 10^9$ im $\sqrt{n}-Level$ Graph. Die dritte Graphklasse wird durch den Graphen *sk2005*, der reale Daten aus einem Web-Crawl darstellt, repräsentiert. Dieser Graph wurde unter anderem in [6] verwendet und wird dort näher beschrieben. Er besteht aus ca. $5 \cdot 10^7$ Knoten und $1,8 \cdot 10^9$ Kanten und hat einen Durchmesser von 40. In der vierten Klasse befindet sich der Graph *cl_n2_29* mit 2^{29} Knoten und einer besonderen listenartigen Baumstruktur, deren Beschreibung [4] entnommen werden kann.

Weiterhin wurden Tests auf kleinen Graphen, die realen Anwendungen entstammen durchgeführt. Hierbei handelt es sich bei den Graphen *ca-AstroPh* und *dblp* um zwei Graphen aus [11], die ein Interaktionsnetzwerk beschreiben und damit sehr ähnlich zu den beiden Graphen *Facebook NY* und *Facebook Santa Barbara (SB)* sind, die das Facebooknetzwerk aus dem Jahr 2008 in der jeweiligen Region beschreiben und [19] entstammen. Der Graph *(p2p-)Gnutella31* bildet ein Peer-to-Peer-Netzwerk ab, *web-BerkStan* ist ein Web-Graph und *hyperGrid* bildet einen synthetischen Hypergraphen. Nähere Beschreibungen der Graphen finden sich in [2]. Alle kleinen realen Graphen haben maximal 10^6 Knoten und Durchmesser zwischen 5 und 23.

5.2.1 Experimente auf großen Graphen

Die großen Graphen wurden bereits in [4] verwendet, um das Verhalten der dynamischen BFS im Externspeicher beim Einfügen von Kanten zu untersuchen. Diese Graphen werden im Rahmen dieser Arbeit wieder verwendet. Es wird das Verhalten der dynamischen Breitensuche beim Löschen von Kanten in I/O-intensiven Fällen untersucht. Der I/O-intensive Fall der Kantenlöschung ist gemäß Kapitel 4.1 Fall **3.4 Bestehende Kante in ein nachfolgendes BFS-Level**. Je tiefer das Verbindungslevel liegt, desto aufwendiger zudem die Änderungen im BFS-Baum. Um solche Situationen künstlich zu erzeugen wurde die in Kapitel 4.3 beschriebene Funktion genutzt, mit der zunächst eine Kante zu von der Wurzel unterschiedlich tief im BFS-Baum sitzenden Knoten eingefügt und anschließend wieder gelöscht wird. Somit ist in jeder zweiten Iteration der Orignalgraph und Original-BFS-Baum wiederhergestellt.
Die Untersuchungen wurden in der Umgebung 1 auf den oben beschriebenen Originalgraphen aus [4] durchgeführt. Aus zeitlichen Gründen nicht untersucht werden konnte der Graph *cl_n2_29*. Für den $\sqrt{n} - level$ Graph wurde das Experiment nur für die erste Iteration ausgeführt. Für letzteren Graphen ließen sich jedoch die Ergebnisse auf Basis der Einfügeergebnisse aus [4] extrapolieren. Die Ergebnisse für die jeweilige Laufzeit beim Einfügen bzw. Löschen der jeweiligen Kante, sowie im Vergleich

die Zeit für die statische BFS des Graphen, lassen sich der Abbildung 5.1 entnehmen. Die detaillierten Messergebnisse finden sich im Anhang A.

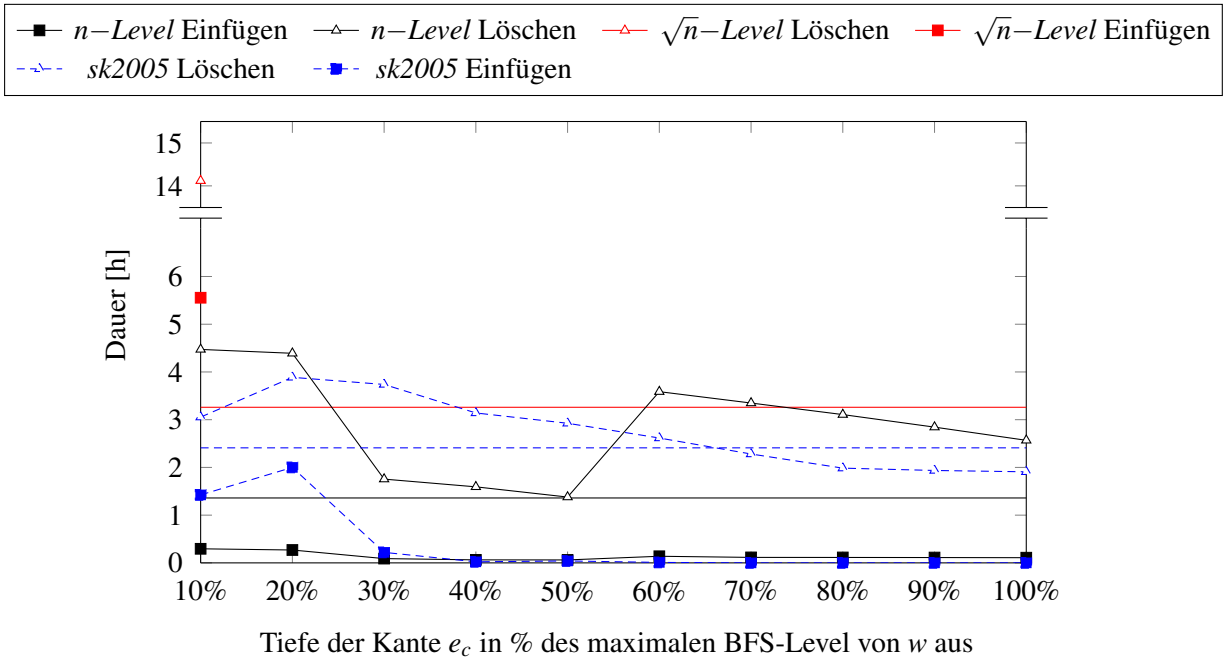

Abbildung 5.1: Laufzeitvergleich von Einfügen und Löschen der Kanten bei dynamischer Breitensuche

Betrachtet man die Ergebnisse, so stellt man fest, dass die Löschung der Kanten mehr Zeit beansprucht als das Einfügen der Kanten. Die Laufzeit ist um einen Faktor $1,5-30$ höher, als für das Einfügen der jeweiligen Kante. Für die beiden Graphen $n-level\ Graph$ und $\sqrt{n}-level\ Graph$ zeigt sich, dass die Zeiten der dynamischen Breitensuche über den Zeiten der statischen Breitensuche für diese Graphen liegen. Für den realen Graphen *sk2005* liegen die Zeiten teilweise über, teilweise unter der Laufzeit der statischen Breitensuche. Im Ergebnis wurde außerdem eine um Faktor > 200 höher liegende Zahl von I/O für den Löschfall erkannt.

Die höhere Laufzeit und Zahl von I/O lässt sich dadurch begründen, dass alle Knoten, die im Rahmen der Voruntersuchung bis zum Erreichen des Abbruchkriteriums besucht werden, ein zweites Mal in der daran anschließenden BFS-Phase besucht werden müssen. Da aus Abbildung 5.2 ersichtlich wird, dass die Änderungen teilweise eine sehr hohe Zahl an Knoten betreffen, wird klar, dass ein Durchlauf des Algorithmus im ungünstigsten Fall aufgrund des mehrfachen Besuchs von Knoten letztendlich mehr als n Knoten besucht. Dies resultiert in einer schlechteren Laufzeit als statisches MM_BFS.
Weiterhin gilt die in Kapitel 4.3 beschriebene Problematik, dass in der eigentlichen Breitensuche zur Leveländerung in den linear durchlaufenen Leveln bei den derzeit gewählten Algorithmen *alle* Knoten betrachtet werden müssen. Dies führt zu einer großen Zahl (teilweise unnötig) besuchter Knoten im Vergleich mit der Kanteneinfügung. Allerdings erklären beide Sachverhalte zusammen nicht die insgesamt sehr hohe Abweichung gegenüber der Laufzeit von statischer Breitensuche. Es ist anzu-

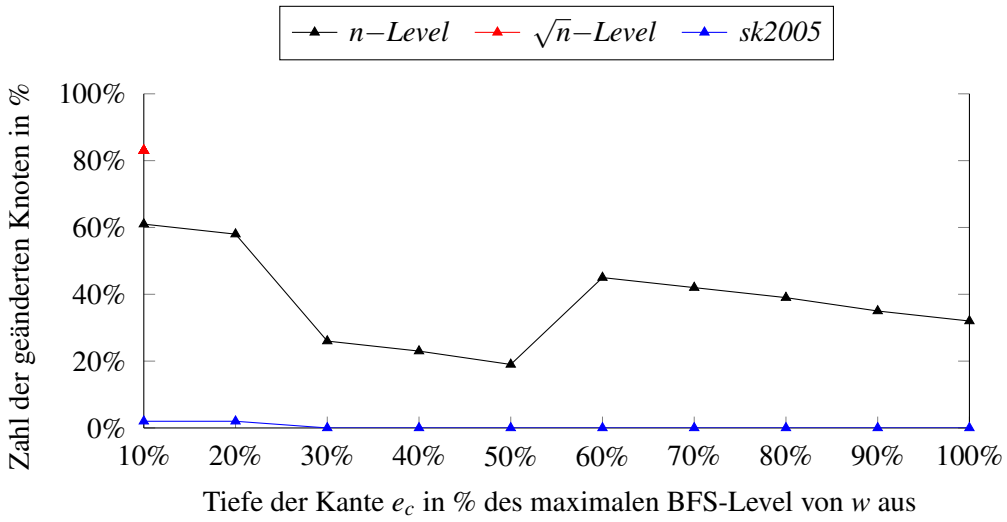

Abbildung 5.2: Zahl der geänderten Knoten

nehmen, dass in der aktuellen Implementierung derzeit noch Codebestandteile existieren, die zuviel Lesezugriffe bei der dynamischen Breitensuche nach dem Löschen von Kanten verursachen.

Bei den durchgeführten Experimenten fällt außerdem auf, dass viel Zeit für die Anpassung und Sortierung der vorverarbeiteten Adjazenzlisten an das neue Ergebnis benötigt wird. Wie bereits in Kapitel 4.3 beschrieben, wurde zunächst die direkte Anpassung der einzelnen Einträge implementiert, dies später jedoch durch einen völligen Neuaufbau der vorsortierten Adjazenzlisten ersetzt. Dieses Vorgehen hat zur Folge, dass bei jeder gravierenden Änderung in BFS-Leveln der Neuaufbau erfolgt. Die dafür benötigte Zeit lag bereits in den Experimenten aus [4] zwischen $0,5$ und 2 Stunden, was sich in den hier durchgeführten Experimenten bestätigte. Die hierfür benötigten I/O gehen in der Theorie mit $O(sort(n) \cdot log(B))$ ein und müssen in der Praxis mit entsprechendem Zeitbedarf auch geleistet werden.

5.2.2 Experimente auf realen Graphen

Auf den realen Graphen wurden in der oben beschriebenen Umgebung 2 Experimente durchgeführt, um ein Indiz dafür zu erhalten, wie häufig die Einzelnen in Kapitel 4.1 definierten Fälle auftreten. Insbesondere interessant ist, wie häufig ein I/O-intensiver Fall, wie er in den im vorangegegangenen Abschnitt beschriebenen Experimenten bewusst provoziert wurde, auftritt. Zu diesem Zweck wurden in den vorliegenden realen Graphen jeweils 1000 Kanten nacheinander gelöscht und der jeweilige auftretende Fall gemäß protokolliert.

Um diesen Sachverhalt zu untersuchen war es nicht zwingend notwendig möglichst große Graphen zu wählen, sondern möglichst auf realen Daten basierende Graphen. Daher wurden neben dem großen realen Graphen *sk2005* vor allem die oben beschriebenen kleinen realen Graphen untersucht. Die Ergebnisse der Untersuchung werden in Tabelle 5.2 dargestellt.

Graph	Delta = 0	Knoten-isolation	Nachbar in vorher-gehendem Level	Nachbar in selbem Level	Nachbar in nach-folgendem Level
ca-AstroPh	54,5%	0%	39,1%	6,3%	0,1%
dblp	42,7%	11,4%	30,7%	14,8%	0,4%
Facebook NY	44,6%	12,6%	37,1%	4,9%	0,8%
Facebook SB	41,3%	12,9%	37,2%	7,5%	1,1%
Gnutella31	17,9%	0,4%	69,5%	5,5%	6,7%
hyperGrid	12,0%	43,8%	36,9%	7,3%	0%
sk2005	55,6%	8,4%	32,0%	3,4%	0,6%
web-BerkStan	51,7%	0%	34,7%	9,2%	4,4%

Tabelle 5.2: Anzahl der auftretenden Fälle beim zufälligen Löschen von 1000 Kanten je Graph

Die Ergebnisse zeigen, dass die Zahl der I/O-intensiven Fälle deutlich hinter den einfachen Änderungen zurückbleibt. Bei den meisten Graphen fallen lediglich $0-1,1\%$ aller Löschungen in diese Kategorie. Bei wenigen Graphen liegt der Anteil der I/O-intensiven Fälle höher, bleibt aber immer weit unter 10% aller Fälle. Die Zeit, die für eine Kantenlöschung im einfachen Fall benötigt wird, liegt deutlich unter der Zeit für die I/O-intensiven Fälle. Für den großen Graphen *sk2005* wurden hier Zeiten von ca. $0,03$ Stunden gemessen.

Kritisch betrachten muss man, dass hier jeweils nur 1000 Kanten gelöscht wurden, was in allen Graphen einem Anteil von maximal 1% aller Kanten entspricht. Je mehr Kanten gelöscht werden, desto dünner wird der Graph und umso höher ist die Wahrscheinlichkeit, dass mit einer Kantenlöschung ein I/O-intensiver Fall betrachtet werden muss, da keine Kanten zu Knoten in vorangehenden BFS-Leveln mehr existieren.

5.3 Gesamtinterpretation

Die praktischen Experimente mit dem im Rahmen dieser Arbeit erstellten Code haben gezeigt, dass dieser Code in den I/O-intensiven Fällen der Kantenlöschung noch nicht immer die notwendigen Ergebnisse liefert um einen Vorteil gegenüber statischem MM_BFS aufzuweisen. Insbesondere auf den synthetischen Graphen liegen die Laufzeitmessungen deutlich über der Zeit für statische Breitensuche und verfehlen damit das eigentliche Ziel.
Wichtig ist die gemeinsame Erkenntnis aus den beiden durchgeführten Experimenten, dass die I/O-intensiven Fälle für Kantenlöschung bei zufälliger Kantenwahl im Verhältnis zu den einfachen Fällen eher selten auftreten. Dabei ist es wichtig das Experiment aufgrund der geringen Zahl gelöschter Kanten hinsichtlich seiner Signifikanz zu hinterfragen. Betrachtet man allerdings beispielsweise Kantenlöschungen in einem sozialen Netzwerk, so erscheinen die Ergebnisse plausibel. Das Äquivalent zu einem I/O-intensiven Fall der Kantenlöschung wäre dort beispielsweise die Aufhebung einer Freundschaftsbeziehung zwischen zwei Mitgliedern, deren Freunde und Freunde dieser Freunde in einer langen Liste keinerlei Beziehung zueinander haben. Dies könnte das Löschen einer Freundschaftsbeziehung zu einer Urlaubsbekanntschaft sein, deren Freundeskreis nie zu den eigenen Freunden gehört

hat. Hierzu konnte keine existierende Untersuchung gefunden werden, bzw. auf Basis der vorhandenen Graphen selbst durchgeführt werden, aber die Annahme, dass dieser Fall im Gegensatz zu normalen Freundschaftsdynamiken, sprich das Bilden eines neuen eng vermaschten Netzwerkes, seltener auftritt scheint zunächst legitim.

Ebenso legitim erscheint die Annahme, dass bei der Betrachtung realer Graphen deutlich weniger Kantenlöschungen als Kanteneinfügungen auftreten. In sozialen Netzwerken beispielsweise werden deutlich seltener Beziehungen zwischen Personen oder zu anderen Inhalten aufgelöst, als dass diese neu geschaffen werden, wie es z.B. dem Vergleich der Facebook-Freundschaften der Jahre 2013 [15] und 2014 [17] entnommen werden kann. Dies gilt im Regelfall auch für andere Netzwerke wie Peer-to-Peer Netzwerke oder Webgraphen. Die Zahl der Inhalte oder beteiligter Nutzer steigt im Regelfall deutlich an, s. z.B. [18]. Eine nennenswerte Ausnahme ist das deutsche soziale Netzwerk studiVZ, s. [16]. Auch bei wissenschaftlichen Netzwerken zur Untersuchung biologischer, physikalischer oder chemischer Prozesse, die den Zerfall einer Struktur zum Ziel haben, ist der Sachverhalt genau umgekehrt. Allerdings gilt auch für diese die in der Theorie nachgewiesene Beschänkung der Zahl I/O-intensiver Graphänderungen: nicht jede eingefügte/gelöschte Kante kann zu einer gravierenden Graphänderung, die viele I/O benötigt, führen.

Betrachtet man die Ergebnisse gesamthaft, so lässt sich der vorliegende Code nutzen, um bei Kantenlöschungen die einfachen Fälle zu identifizieren und von den I/O-intensiven Fällen zu unterscheiden. Für die I/O-intensiven Fälle scheint es je nach Graph sinnvoll eher eine erneute statische Breitensuche durchzuführen, insofern der vorliegende Code nicht weiter optimiert werden kann (s. nächstes Kapitel). Unabhängig von einer Anpassung an dieser Stelle lässt sich jedoch aufgrund der geringen Laufzeit für einfache Updates (ca. $0,03$ Stunden, s.o.) schlussfolgern, dass auf realen Graphen für eine Folge von m' Kantenlöschungen die Gesamtzahl der I/O und damit der Laufzeit bei Anwendung der hier vorgestellten Algorithmen tatsächlich unter der m'-maligen Anwendung des statischen MM_BFS liegt. Die Aufwände, die für einen Neuaufbau der vorsortierten Adjazenzlisten bzw. die Korrektur der Inhalte auftreten, sind jedoch bei Nutzung des dynamischen BFS-Codes nicht zu unterschätzen.

5.4 Verbesserungspotentiale

Verbesserungen an der aktuellen Implementierung sind möglich und durchaus notwendig, um das Ziel Laufzeit gegenüber dem statischen MM_BFS einzusparen in allen Fällen zu erreichen. Dabei ist es sinnvoll – neben einer kritischen Untersuchung der aktuellen Implementierung auf Schwachstellen im Code – für die folgenden identifizierten Problemstellungen Lösungen zu entwickeln:

Doppeltes Besuchen von Knoten

Um zu vermeiden, dass durch die dynamische Breitensuche mehr als n Knoten besucht werden, gilt es, ein geeignetes Kriterium zu finden, um in eine statische Neuberechnung der BFS-Level umzusteigen. Ein geeignetes Abbruchkriterium könnte im Rahmen der Voruntersuchung nach einem Verbindungslevel implementiert werden. Ein mögliches Maß wäre die Zahl der bereits besuchten Knoten

im Vergleich zur Gesamtzahl aller Knoten. Sobald ein Anteil $\beta < 50\%$ aller Knoten des Graphen besucht wurde, wird der Vorgang abgebrochen und in die statische Neuberechnung gewechselt. Es kann allerdings auch dann nicht garantiert werden, dass maximal n Knoten besucht werden, da bereits $\beta \cdot n$ Knoten besucht wurden. Um diesem Problem entgegenzuwirken, kann die im Kapitel 4.1 vorgestellte Idee zum Abbruch genutzt werden: die statische Breitensuche muss erst ab dem dann bereits erreichten Voruntersuchungslevel – unter Ausschluss aller bereits besuchten Knoten dieses Levels – durchgeführt werden. Wie dort bereits beschrieben muss allerdings eine Lösung für den Fall der Abtrennung einer Graphkomponente gefunden werden.

Ein weiterer Ansatz ist die Nutzung der Voruntersuchung um die jeweilige BFS-Level-Abhängigkeit zwischen zwei Nachbarn zu speichern und bei Auffinden eines Verbindungslevels die resultierenden Level auf schnelle Art und Weise auf die bereits besuchten Knoten linear fortzupropagieren. Dieses Verfahren kann Zeit einsparen, erfordert allerdings eine vertiefende Betrachtung, da beispielsweise im in Abbildung 4.3 dargestellten Fall eine einfache lineare Propagation über das erste gefundene Verbindungslevel zum falschen Ergebnis führen würde.

Besuchen aller Knoten eines Levels

Wie bereits erwähnt, ist es bei den verwendeten Algorithmen in der Breitensuche zur Veränderung der BFS-Level notwendig, immer alle Knoten eines Levels und deren Nachbarn zu besuchen – auch wenn sich an deren BFS-Level nichts ändert. Es werden also im schlechtesten Fall sehr viele Knoten besucht, ohne dass der Besuch einen Mehrwert für die momentane Breitensuche bringt. An dieser Stelle kann möglicherweise eine große Zahl von I/O gespart werden, indem der Algorithmus zur Voruntersuchung so definiert wird, dass zunächst *alle* Verbindungslevel zu Knoten mit kürzestem Weg zu w, der nicht über v_2 führt, identifiziert werden und die Breitensuche nur an diesen Nachbarknoten im jeweiligen Level gestartet wird. Je nach Graph und konkreter Situation kann der Aufwand *alle* Verbindungslevel zu identifizieren sehr hoch werden. Ob sich dieses Vorgehen insgesamt amortisiert und wie sich die Leistungsfähigkeit im Vergleich zwischen realen Graphen und synthetischen Graphen darstellt, wäre experimentell zu untersuchen.

Anpassung der vorsortierten Adjazenzlisten

Die Experimente haben gezeigt, dass die Anpassung der vorsortierten Adjazenzlisten für die schnelle Befüllung der Hotpools in den nächsten Iterationen erhebliche Zeit in Anspruch nehmen. Insbesondere beim Einfügen von Kanten liegt die hierfür benötigte Zeit teilweise um mehr als das tausendfache über der eigentlichen dynamischen Breitensuche. Ist mehr als ein Knoten betroffen, so erfolgt derzeit immer ein grundsätzlicher Neuaufbau der vorsortierten Adjazenzlisten. Für eine kleine Anzahl von Knoten ist dieses Vorgehen nicht effizient. Insbesondere in den einfachen Fällen müssen im Regelfall nur wenige Änderungen erfolgen, z.B. das Löschen einer Kante und das Entfernen eines Knotens aus dem BFS-Baum bei der Isolation eines Knotens. Auch hier sollte ein Schwellwert gefunden werden, unterhalb dessen nur einzelne Reparaturen durchgeführt werden, wohingegen bei einer großen Zahl von Änderungen der grundsätzliche Neuaufbau der vorsortierten Adjazenzlisten bestehen bleibt.

Kapitel 6

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wird in Kapitel 4 gezeigt, dass die theoretischen Ergebnisse zur dynamischen Breitensuche aus [13] – wie dort bereits erwartet – auch für das Löschen von Kanten gelten. Die Zahl der I/O ist mit hoher Wahrscheinlichkeit auf $O\left(n\left(\frac{n}{B^{\frac{2}{3}}} + sort(n) \cdot log(B)\right)\right)$ I/O bei $\Theta(n)$ dynamischen Einfügungen und Löschungen von Kanten in dünnen Graphen beschränkt.
Zur praktischen Überprüfung wurde die Implementierung, die im Rahmen der Arbeit an [4] erfolgt ist, um das Löschen von Kanten erweitert. In Kapitel 5 wurde die experimentelle Untersuchung dieser Implementierung mit einigen synthetischen und realen Graphen dargestellt. Dabei hat sich gezeigt, dass die aktuelle Implementierung in der Lage ist die I/O-intensiven Fälle der Kantenlöschung von den einfachen Fällen zu unterscheiden. Für die I/O-intensiven Fälle ist die Implementierung noch nicht optimal, da bei bestimmten Graphen und Konstellationen Laufzeiten auftreten, die diejenige der statischen Breitensuche übersteigen. Die einfachen Fälle weisen hingegen eine äußerst geringe Laufzeit auf und treten bei Zufallsexperimenten auf realen Graphen deutlich häufiger auf, als die I/O-intensiven Fälle. Betrachtet man eine Folge von $m' = \Theta(n)$ Einfügungen, so amortisieren sich die Laufzeiten gegenüber der m'-maligen Anwendung der statischen Breitensuche.

Der vorliegende Code sollte weiter optimiert werden, da Laufzeiten der dynamischen Breitensuche, die über der Durchführung einer statischen Breitensuche liegen, nicht zielführend und notwendig sind. Hierzu wurden bereits im Kapitel 5.4 einige Vorschläge gemacht, die zu einer geringeren Zahl betrachteter Knoten und damit einer Laufzeitreduktion führen können.

In [13], [4] und dieser Arbeit wurde gezeigt und experimentell nachgewiesen, dass es auch für große Graphen, die nicht vollständig im internen Speicher gehalten werden können, eine Möglichkeit gibt, dynamische Graphänderungen in kurzer Zeit in einen angepassten BFS-Baum zu überführen.
Eine direkte Überführung der Implementierung zu realer Anwendung, wie sie in Kapitel 1 benannt wird, ist allerdings noch nicht möglich. Auf großen realen dynamischen Graphen ist es nicht praktikabel jede Veränderung des Graphen sequentiell zu berechnen. Solche Graphen verändern sich teilweise in Sekundenbruchteilen, insbesondere werden auch gleichzeitig Kanten eingefügt und gelöscht. Somit ist eine Betrachtung einer gleichzeitigen Änderung von mehr als einer Kante erforderlich. Hierfür müssten die existierenden Algorithmen angepasst werden. In diesem Fall ist insbesondere eine er-

neute Betrachtung notwendig, ob sich in der Praxis weiterhin ein zeitlicher Vorteil gegenüber der wiederholten Durchführung der statischen Breitensuche ergibt.
Da viele der in Kapitel 1 genannten Anwendungen auf sich in Echtzeit verändernde Daten zurückgreifen, wird als zusätzliche Herausforderung die Reduktion der Laufzeit auf echtzeittaugliche Werte bestehen. Wie die Ergebnisse in Kapitel 5 zeigen, ist man davon für eine einzelne Änderung noch weit entfernt. Die Werte für reale Graphen sind allerdings bereits deutlich besser, als für synthetische Graphen. Um dynamische Breitensuche in Echtzeit zu erreichen sind neben der Algorithmenverbesserung weitere Optimierungsmöglichkeiten, wie beispielsweise die Parallelisierung zu betrachten.

Die Breitensuche ist nur einer von vielen Graphalgorithmen, die auf sehr großen Graphen eine spezielle Betrachtung benötigen, um eine praxistaugliche Laufzeit zu erhalten. Insofern bleibt nicht nur für die Breitensuche, sondern auch für andere grundlegende Graphalgorithmen noch viel Raum zur theoretischen und experimentellen wissenschaftlichen Untersuchung. Die Notwendigkeit für solche Untersuchungen ist aufgrund der rasanten Datenzunahme und dem ebenso stetig wachsenden Interesse an der Untersuchung strukturierter wie unstrukturierter Daten gegeben.

Anhang A

Experimentalergebnisse

Für die Untersuchung der großen Graphen sind die detaillierten Ergebnisse im Folgenden aufgelistet.

A.1 $n - level$ Graph

A.1.1 Einfügen von Kanten

Beim Einfügen von Kanten in der entsprechenden Tiefe wurden folgende Laufzeitergebnisse erzielt:

Tiefe der Kante	Dauer Voruntersuchung [s]	Dauer BFS-Phase [s]	Zahl geänderter Knoten
10%	0	1062	164287145
20%	0	971	155640453
30%	0	323	69561539
40%	0	227	60914847
50%	0	222	52268155
60%	0	499	121053685
70%	0	413	112406993
80%	0	409	103760301
90%	0	402	95113609
100%	0	391	86466918

Tabelle A.1: Laufzeit beim Einfügen einer Kante in entsprechender Tiefe ausgehend von Wurzel w im $n - level$ Graph

A.1.2 Löschen von Kanten

Beim Löschen von Kanten in der entsprechenden Tiefe wurden folgende Laufzeitergebnisse erzielt:

Tiefe der Kante	Dauer Voruntersuchung [s]	Dauer BFS-Phase [s]	Zahl geänderter Knoten
10%	208	15891	164287145
20%	419	15390	155640453
30%	627	5692	69561539
40%	748	4986	60914847
50%	688	4277	52268155
60%	1260	11653	121053685
70%	1364	10694	112406993
80%	1306	9878	103760301
90%	1256	8981	95113609
100%	1163	8083	86466918

Tabelle A.2: Laufzeit beim Löschen einer Kante nach dem Einfügen in entsprechender Tiefe ausgehend von Wurzel *w*

A.2 *sk2005* Graph

A.2.1 Einfügen von Kanten

Beim Einfügen von Kanten in der entsprechenden Tiefe wurden folgende Laufzeitergebnisse erzielt:

Tiefe der Kante	Dauer Voruntersuchung [s]	Dauer BFS-Phase [s]	Zahl geänderter Knoten
10%	0	5126	873776
20%	0	7200	873767
30%	0	792	28777
40%	0	81	4712
50%	0	145	744
60%	0	35	8510
70%	0	0	12
80%	0	0	9
90%	0	1	101
100%	0	1	155

Tabelle A.3: Laufzeit beim Einfügen einer Kante in entsprechender Tiefe ausgehend von Wurzel *w* im *sk2005* Graph

A.2.2 Löschen von Kanten

Beim Löschen von Kanten in der entsprechenden Tiefe wurden folgende Laufzeitergebnisse erzielt:

Tiefe der Kante	Dauer Voruntersuchung [s]	Dauer BFS-Phase [s]	Zahl geänderter Knoten
10%	0	10993	873776
20%	1504	12489	873767
30%	2977	10478	28777
40%	4460	6848	4712
50%	6339	4190	744
60%	6635	2783	8510
70%	6880	1330	12
80%	6654	493	9
90%	6855	113	101
100%	6843	24	155

Tabelle A.4: Laufzeit beim Löschen einer Kante nach dem Einfügen in entsprechender Tiefe ausgehend von Wurzel *w* im *sk2005* Graph

Literaturverzeichnis

[1] AGGARWAL, Alok ; VITTER, S. Jeffrey: The Input/Output Complexity of Sorting and Related Problems. In: *Commun. ACM* 31 (1988), sep, Nr. 9, 1116–1127. http://dx.doi.org/10.1145/48529.48535. – DOI 10.1145/48529.48535. – ISSN 0001–0782

[2] AJWANI, Deepak ; KENNEDY, W. S. ; SALA, Alessandra ; SANIEE, Iraj: A Geometric Distance Oracle for Large Real-World Graphs. In: *CoRR* abs/1404.5002 (2014). http://arxiv.org/abs/1404.5002

[3] ARGE, Lars ; BRODAL, Gerth S. ; TOMA, Laura: On external-memory MST, {SSSP} and multi-way planar graph separation. In: *Journal of Algorithms* 53 (2004), Nr. 2, 186 - 206. http://dx.doi.org/http://dx.doi.org/10.1016/j.jalgor.2004.04.001. – DOI http://dx.doi.org/10.1016/j.jalgor.2004.04.001. – ISSN 0196–6774

[4] BECKMANN, Andreas ; MEYER, Ulrich ; VEITH, David: An Implementation of I/O-Efficient Dynamic Breadth-First Search Using Level-Aligned Hierarchical Clustering. In: *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, 2013, 121–132

[5] CORMEN, T.H. ; LEISERSON, C.E. ; RIVEST, R.L. ; STEIN, C.: *Introduction To Algorithms*. MIT Press, 2001 https://books.google.at/books?id=NLngYyWFl_YC. – ISBN 9780262032933

[6] *Kapitel* Finding the Diameter in Real-World Graphs. In: CRESCENZI, Pierluigi ; GROSSI, Roberto ; IMBRENDA, Claudio ; LANZI, Leonardo ; MARINO, Andrea: *Algorithms – ESA 2010: 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. – ISBN 978–3–642–15775–2, 302–313

[7] DEMENTIEV, R. ; KETTNER, L. ; ; SANDERS, P.: STXXL: standard template library for XXL data sets. In: *Software: Practice and Experience* 38 (2008), S. 589–637. http://dx.doi.org/10.1002/spe.844. – DOI 10.1002/spe.844

[8] DIETZFELBINGER, Martin ; MEHLHORN, Kurt ; SANDERS, Peter: *Algorithmen und Datenstrukturen: Die Grundwerkzeuge*. Springer Berlin Heidelberg, 2014 (eXamen.press). https://books.google.at/books?id=CdwfBAAAQBAJ. – ISBN 9783642054723

[9] EMC CORPORATION: *EMC Newsroom - Press Release - Study Projects Nearly 45-Fold Annual Data Growth by 2020*. http://www.france.rsa.com/about/news/press/2010/20100504-01.htm, 2010. – Accessed on 13.03.2016

[10] FACEBOOK IRELAND LIMITED: *Facebook Newsroom - Company Info - Stats*. http://newsroom.fb.com/company-info/, 2015. – Accessed on 13.03.2016

[11] Accessed on 01.04.2016

[12] MEHLHORN, K. ; MEYER, U.: External-Memory Breadth-First Search with Sublinear I/O. In: *Proceedings of the 10th annual European Symposium on Algorithms (ESA)* Bd. 2461, Springer, 2002 (LNCS), S. 723–735

[13] MEYER, Ulrich: On Dynamic Breadth-First Search in External-Memory. In: *CoRR* abs/0802.2847 (2008). http://arxiv.org/abs/0802.2847

[14] MUNAGALA, K. ; RANADE, A.: I/O-complexity of graph algorithms. In: *Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA)*, ACM-SIAM, 1999, S. 687–694

[15] PEW RESEARCH CENTER: *6 new facts about Facebook*. http://www.pewresearch.org/fact-tank/2014/02/03/6-new-facts-about-facebook/, 2013. – Accessed on 02.04.2016

[16] SEMPHATIC ONLINE MARKETING: *Wann stirbt StudiVZ.net?* https://wannstirbtstudivz.net, 2016. – Accessed on 02.04.2016

[17] STATISTA GMBH: *Durchschnittliche Anzahl von Facebook-Freunden bei US-amerikanischen Nutzern nach Altersgruppe im Jahr 2014*. http://de.statista.com/statistik/daten/studie/325772/umfrage/durchschnittliche-anzahl-von-facebook-freunden-in-den-usa-nach-altersgruppe/, 2014. – Accessed on 02.04.2016

[18] STATISTA GMBH: *Anzahl der Nutzer sozialer Netzwerke weltweit in den Jahren 2010 bis 2013 sowie eine Prognose bis 2018 (in Milliarden)*. http://de.statista.com/statistik/daten/studie/219903/umfrage/prognose-zur-anzahl-der-weltweiten-nutzer-sozialer-netzwerke/, 2016. – Accessed on 02.04.2016

[19] WILSON, Christo ; BOE, Bryce ; SALA, Alessandra ; PUTTASWAMY, Krishna P. ; ZHAO, Ben Y.: User Interactions in Social Networks and Their Implications. In: *Proceedings of the 4th ACM European Conference on Computer Systems*. New York, NY, USA : ACM, 2009 (EuroSys '09). – ISBN 978–1–60558–482–9, 205–218