Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 114

# Externe komprimierte Graphdarstellungen

Tobias Bagg

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer/in:** | Prof. Dr. Stefan Funke |
| **Betreuer/in:** | Prof. Dr. Stefan Funke |
| **Beginn am:** | 2014-02-06 |
| **Beendet am:** | 2014-05-13 |
| **CR-Nummer:** | G.2.2, E.4 |

# Kurzfassung

In dieser Bachelorarbeit wird untersucht, inwiefern sich Graphdaten für Routenplaner komprimieren lassen und gleichzeitig kürzeste Wege effizient berechnet werden können. Motiviert wird dies insbesondere durch die weiterhin wachsende Größe des Kartenmaterials, was aus dem ständig verbesserten Detailgrad resultiert, als auch durch die Vergrößerung der geographischen Ausdehnung dieser Graphen. Erschwerend kommt hinzu, dass Speicher auf Geräten, welche häufig zur Routenplanung eingesetzt werden, auch heutzutage noch eine eingeschränkte Ressource darstellt. Ebenfalls sind Prozessoren in diesem Einsatzbereich eher auf Energieeffizienz ausgelegt und deshalb leistungsschwächer. Diese Arbeit soll erläutern, wie sich diese Herausforderungen durch die richtige Wahl von Algorithmen, Datenstrukturen und Techniken bewältigen lassen.

Die gewählten Algorithmen und Datenstrukturen wurden in Java implementiert. Die Implementierung wurde mit unterschiedlichen Parametern für Kompression und dem Verhalten des Caching evaluiert, sowie interpretiert und in dieser Arbeit festgehalten.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis



# Verzeichnis der Listings

# Verzeichnis der Algorithmen

# 1 Motivation

Eine Software zur Routenplanung repräsentiert das Straßennetzwerk durch einen gerichteten sowie gewichteten Graphen, mit dessen Hilfe beispielsweise der kürzeste Weg zwischen zwei Punkten berechnet werden kann. Mittlerweile sind diese Graphen detailliert und repräsentieren sehr genau die Realität. Mit der Verbesserung der Graphen ist gleichzeitig der Speicherverbrauch stark angestiegen. Alleine der Straßengraph von Deutschland umfasst mittlerweile 15 Millionen Knoten und 30 Millionen Kanten, wodurch Speicher im Bereich von einem Gigabyte belegt wird. Hier sind allerdings nur Pfade vorhanden, welche von Kraftfahrzeugen befahren werden können. Nimmt man weitere Fuß- und Fahrradwege hinzu und dehnt den Graphen geographisch auf Europa aus, steigt der Speicherverbrauch erneut erheblich an. Routenplaner werden zusätzlich gerne auf mobilen Geräten, wie zum Beispiel Smartphones, benutzt. Hierbei kommt eine weitere Schwierigkeit hinzu: auf mobilen Geräten ist Speicher weiterhin eine kostbare Ressource und die vorhandenen Prozessoren sind nicht so leistungsfähig wie in gewöhnlichen Heimrechnern. Möchte man also den Speicherbedarf verkleinern, so müssen die Graphrepräsentationen komprimiert werden. Kompressionsverfahren sind ein gut erforschtes Thema der Informatik, wodurch sich der Verbrauch an Speicher deutlich senken lässt. Hierbei tritt ein bekannter Kompromiss zwischen Speicherverbrauch und Berechnungszeit auf: Der Speicherverbrauch kann sich auf Kosten einer schlechteren Berechnungszeit verbessern lassen. Umgekehrt gilt auch, dass sich die Berechnungszeit verbessern lässt, wenn mehr Speicherplatz spendiert wird. In dieser Bachelorarbeit soll deshalb untersucht werden, inwiefern sich die vorhanden Graphdaten auf einem Externspeicher komprimieren lassen, kürzeste Wege aber weiterhin effizient und schnell berechnet werden können.

## 1.1 Verwandte Arbeiten

Es existiert eine Vielfalt von Arbeiten, welche sich mit der Beschleunigung der Berechnung kürzester Pfade beschäftigt haben. Dabei gibt es unterschiedliche Herangehensweisen. Die meisten Verfahren erfordern dabei eine Vorverarbeitung des Graphen, wobei zum Beispiel zusätzliche Kanten oder Informationen (zum Beispiel Arc-Flags[KMS05]) hinzugefügt werden. Beispiele für Verfahren sind Core-ALT [BDS+10] oder SHARC [BD09].

Im Bereich kompakter Graphrepräsentationen wurden ebenfalls eine Vielzahl von Arbeiten veröffentlicht. Zum Beispiel erzeugen Blandford et al.[BBK04] im Vorfeld einen sogenannten „separator tree“ durch das Entfernen von Kanten aus dem Graph, um diesen in mehrere Komponenten zu separieren. Die Knoten werden dann an Hand des aufgebauten Baums in der Graphrepräsentation angeordnet, wodurch die Differenz zwischen adjazenten Knoten relativ klein ist. Dies ermöglicht eine effiziente Kodierung der IDs.

Zusätzlich kann noch „Mobile route planning“ von Sanders et al. [SSV08] erwähnt werden. Die Zielsetzung der Autoren war ähnlich wie in dieser Arbeit, wobei eine Implementierung und Evaluierung für mobile Geräte erfolgte die in ~100ms kürzeste Pfade zwischen Knoten berechnet. Der Fokus lag hierbei nicht auf gängigen Kompressionsverfahren, sondern einer sehr kompakten Graphrepräsentation.

# 2 Hintergrund

Innerhalb dieses Kapitels werden die grundsätzlichen Probleme sowie deren Lösungen vorgestellt, welche für die Implementierung auftreten. Dafür werden zuerst gängige Graphdarstellungen in Bezug auf Struktur, Speicherverbrauch und Zugriffszeiten analysiert. Anschließend wird darauf eingegangen auf welche Weise allgemein kürzeste Wege in Graphen mit Hilfe des Dijkstra Algorithmus [Dij59] berechnet werden können und wie sich dies mit Hilfe von Contraction Hierarchies [GSSD08], einer Vorverarbeitung des Graphen, stark beschleunigen lässt. Abschließend wird auf das Thema Datenkompression eingegangen, wobei speziell die in der Implementierung verwendeten Verfahren diskutiert werden.

## 2.1 Probleme

Mobile Geräte wie Smartphones sind im Vergleich zu Desktop-Computern aus verschiedenen Gründen (zum Beispiel Energieverbrauch) deutlich leistungsschwächer. Die stärksten Unterschiede liegen zum Einen bei den Prozessoren und zum Anderen bei der begrenzten Menge an Speicher (Arbeitsspeicher, als auch Kapazität der Speicherkarten). Auf Computern mit großer Speicherkapazität ist es möglich den Graph in unkomprimierter Form zu speichern. Bei genügender Menge an Arbeitsspeicher kann dieser sogar darin zwischengespeichert werden, um langsamen Zugriffen auf die Speicherkarte aus dem Weg zu gehen. Dies ist bei praktisch keinem Mobilgerät der Fall. Eine Implementierung muss deshalb die Anzahl der Zugriffe auf den externen Speicher, als auch das Dekomprimieren von Teilen des Graphs klein halten. Hierzu ist eine Technik zur Berechnung von kürzesten Pfaden nötig, welche möglichst wenige Knoten besuchen muss. Zusätzlich sollten Speicherzugriffe durch das Cachen von Speicherblöcken des Graphen verhindert werden.

## 2.2 Graphrepräsentationen

Graphen lassen sich in Computern auf unterschiedliche Art repräsentieren. Dies führt, je nach Repräsentation, zu unterschiedlichem Speicherverbrauch und Zugriffszeiten auf Kanten. Im Folgenden wird, wie in der Implementierung verwendet, von gerichteten Graphen ausgegangen.

Der in Abbildung 2.1 dargestellte Graph (G(V,E)) soll in den folgenden Abschnitten als Beispiel dienen und zeigen, wie sich ein Graph auf unterschiedliche Art speichern lässt.

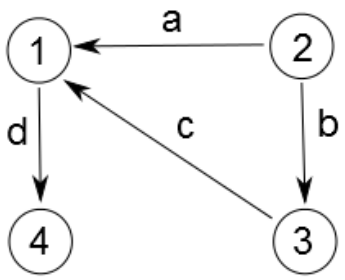



**Abbildung 2.1:** Beispiel eines gerichteten Graph. Jeder Knoten sowie jede Kante ist eindeutig benannt.

### 2.2.1 Adjazenzmatrix

Wie der Name schon andeutet, werden bei dieser Technik die Kanten mit Hilfe einer Matrix verwaltet. Für jeden Knoten entsteht eine Spalte als auch eine Zeile. Dadurch entsteht ein Speicherverbrauch von $O((|V|)^2)$. Steht in der Matrix in Zeile i und Spalte j eine „1“, so ist im Graph eine gerichtete Kante von Knoten i nach Knoten j vorhanden. Steht eine „0“ an dieser Stelle, besteht keine Verbindung in dieser Richtung zwischen diesen beiden Knoten. Ob eine gerichtete Kante zwischen Knoten $v_i$ und $v_j$ vorhanden ist kann somit in konstanter Zeit $O(1)$ bestimmt werden, da lediglich die richtige Stelle der Matrix ausgelesen werden muss. Kompakter beschrieben gilt für eine Adjazenzmatrix M und Graph G(V,E) also:

$$M_{ij} = \begin{cases} 1 \text{ falls } (i,j) \in E, \\ 0 \text{ sonst.} \end{cases}$$

In Tabelle 2.1 ist die Adjazenzmatrix für den Graph in Abbildung 2.1 eingetragen, nach den Regeln die gerade definiert wurden.

| | ***1*** | ***2*** | ***3*** | ***4*** |
|---|---|---|---|---|
| ***1*** | 0 | 0 | 0 | 1 |
| ***2*** | 1 | 0 | 1 | 0 |
| ***3*** | 1 | 0 | 0 | 0 |
| ***4*** | 0 | 0 | 0 | 0 |

**Tabelle 2.1:** Adjazenzmatrix zu Abbildung 2.1

### 2.2.2 Inzidenzmatrix

Eine Inzidenzmatrix speichert die Beziehungen zwischen Knoten und Kanten innerhalb des Graphen. Hierbei entsteht ein Speicherverbrauch von $O(|V| \cdot |E|)$. Die Spalten der Matrix enthalten die Knoten, die Zeilen enthalten die Kanten. Für gerichtete Graphen kann ein Eintrag in der Matrix drei Werte enthalten. Geht von einem Knoten $v_i$ die Kante $e_j$ aus, dann steht in der Matrix in Spalte i und Zeile j eine 1. Würde die Kante nicht von diesem Knoten ausgehen, sondern auf diesen zeigen wäre der Wert eine „-1“. Für eine Inzidenzmatrix M und Graph G(V,E) gilt also:

$$M_{ij} = \begin{cases} 1, & \text{falls } e_j = (v_i, x) \\ 0, & \text{falls } v_i \notin e_j \\ -1, & \text{falls } e_j = (x, v_i) \end{cases}$$

Zum Bestimmen, ob eine gerichtete Kante zwischen zwei Knoten $v_i$ und $v_j$ vorhanden ist, müssen im schlechtesten Fall alle Kanten betrachtet werden. Es entsteht der Aufwand von $O(|E|)$. Erneut ist in Tabelle 2.2 die Struktur für den Graphen in Abbildung 2.1 dargestellt.

| | ***1*** | ***2*** | ***3*** | ***4*** |
|---|---|---|---|---|
| ***a*** | -1 | 1 | 0 | 0 |
| ***b*** | 0 | 1 | -1 | 0 |
| ***c*** | -1 | 0 | 1 | 0 |
| ***d*** | 1 | 0 | 0 | -1 |

**Tabelle 2.2:** Inzidenzmatrix zu Abbildung 2.1.

### 2.2.3 Adjazenzliste

Alle Knoten befinden sich in einer verketteten Liste. Gleichzeitig zeigt jeder Knoten auf eine verkettete Liste seiner adjazenten Knoten. Der benötigte Speicherplatz liegt somit bei $O(|V| + |E|)$.

Ob ein Knoten $v_i$ adjazent zu $v_j$ ist kann nicht in $O(1)$ bestimmt werden. Der Aufwand ist in diesem Fall stark von der Implementierung abhängig. Erfolgt diese in naiver Form, liegt der Aufwand hierfür bei $O(|V| + \#Knoten\ adjazent\ zu\ v_i)$. Es muss zuerst der Knoten $v_i$ in der Liste der Knoten gefunden werden, was im schlechtesten Fall der letzte Knoten der Liste ist. Dann muss der Knoten in der Liste der adjazenten Knoten gefunden werden, was im schlechtesten Fall wieder der letzte Knoten ist.

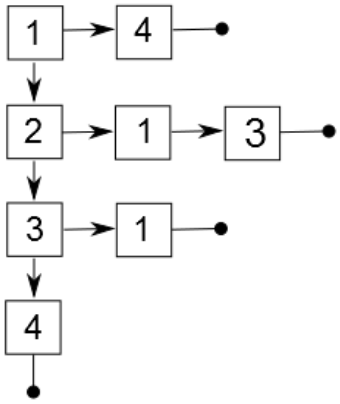



**Abbildung 2.2:** Darstellung des Graph aus Abbildung 2.1 mit Hilfe einer Adjazenzliste

Als Beispiel wird wieder der Graph in Abbildung 2.1 benutzt. In Abbildung 2.2 ist die beschriebene Struktur des Graphen dargestellt. In der vertikalen Liste sind alle Knoten in einer Liste verbunden. Von jedem Knoten geht in der Horizontalen eine Liste seiner adjazenten Knoten aus.

### 2.2.4 Inzidenzliste

Knoten und Kanten werden in getrennten Listen gespeichert. Jede Kante enthält dabei zwei Referenzen auf die Knotenliste: Der Start- und Zielknoten der Kante. Der Speicherbedarf liegt wie bei Adjazenzlisten bei $O(|V| + |E|)$.

Ob eine Kante von $v_i$ nach $v_j$ existiert ist bei dieser Struktur relativ schwierig zu bestimmen. Es müssen im schlimmsten Fall alle Kanten durchsucht werden, somit liegt der Aufwand bei $O(|E|)$.

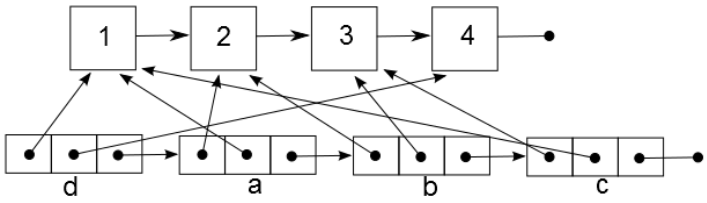



**Abbildung 2.3:** Darstellung des Graph aus Abbildung 2.1 mit Hilfe einer Inzidenzliste

In Abbildung 2.3 ist die Struktur zu Abbildung 2.1 dargestellt. In der oberen Liste sind alle Knoten in einer Liste verkettet. In der Liste darunter befinden sich alle Kanten, welche auf deren Start- und Zielknoten zeigen.

## 2.3 Berechnung kürzester Pfade – Dijkstra

Der klassische Algorithmus zur Bestimmung kürzester Pfade in Graphen mit positiven Kantenkosten ist der Dijkstra Algorithmus [Dij59]. In der allgemeinen Version wird die kürzeste Distanz zu jedem erreichbaren Knoten, ausgehend von einem Startknoten berechnet und die jeweiligen Pfade gespeichert. Bei Anfragen nach zwei konkreten Knoten ist dies nicht gewünscht, sodass die Berechnung abgebrochen werden kann, sobald der Zielknoten erreicht wurde. Der Pseudocode zur Berechnung des kürzesten Pfads zwischen zwei Knoten s und t ist in Algorithmus 2.1 dargestellt. Hierfür wurde lediglich der Dijkstra Algorithmus leicht angepasst.

Zu Beginn werden die drei Arrays „dist“, „previous“ und „settled“ initialisiert, wobei die Anzahl der Indizes der Anzahl der Knoten im Graphen entspricht. Für jeden Knoten $i \in V$ enthält dist[i] den aktuellen Distanzwert, previous[i] den Vorgängerknoten im kürzesten Pfad und settled[i] wird auf „True“ gesetzt, sobald der jeweilige Knoten einmal aus der Warteschlange Q entfernt wird. Während der Berechnung werden die Randknoten mit einer Prioritätswarteschlange (Q, Min-Heap), an Hand der Distanzwerte der Knoten, verwaltet. In jeder Runde wird der Knoten mit dem kleinsten Distanzwert entnommen und dessen Kanten weiter verfolgt. Dijkstras Algorithmus expandiert folglich immer den kürzesten vorhandenen Pfad. Mit einem Min-Heap besitzt dieser Algorithmus eine Worst-Case Laufzeit von $O(|E| + |V| \cdot log(|V|))$.

**Algorithmus 2.1** angepasster Dijkstra-Algorithmus zur Berechnung eines Pfads zwischen zwei Knoten $v_i$ und $v_j$

```
procedure DIJKSTRA(G(V, E), source v_i, target v_j)
    for all v_i ∈ V do
        dist[i] ← ∞, settled[i] ← False, previous[i] ← Null
    end for
    dist[source] ← 0, Q ← source
    while not empty(Q) AND NOT settled[target] do
        u ← getAndRemoveMinElement(Q)
        if not settled[u] then
            for all neighbors v of u do
                alternative ← dist[u] + cost(u, v)
                if alternative < dist[v] then
                    dist[v] ← alternative
                    previous[v] ← u
                    Q ← v
                end if
            end for
        end if
    end while
     return dist, previous
end procedure
```

Eine schematische Darstellung der Ausbreitung des Dijkstra Algorithmus ist in Abbildung 2.4 gegeben. Man erkennt an den gestrichelten grünen Kreisen, dass sich der Algorithmus annähernd kreisförmig um den Startknoten ausbreitet.

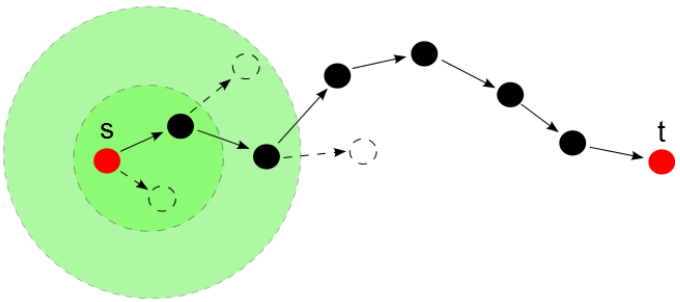



**Abbildung 2.4:** Kreisförmige Ausbreitung des Dijkstra Algorithmus um den Startknoten auf einem Graphen. Start dargestellt durch den Knoten s, das Ziel durch Knoten t.

## 2.4 Contraction Hierarchies

Der klassische Dijkstra Algorithmus [Dij59] eignet sich zum Berechnen kürzester Pfade auf Graphen mit mehreren Millionen Knoten und Kanten nicht, da die Berechnungen zu langsam sind. Deshalb wird eine zusätzliche Technik benötigt. Der Grund wurde bereits im vorherigen Abschnitt in Abbildung 2.4 angedeutet: die kreisförmige Ausbreitung auf großen Graphen liegt im Bereich von mehreren Sekunden. Um dies zu beschleunigen eignen sich Contraction Hierarchies [GSSD08], welche 2008 vorgestellt wurden und eine massive Beschleunigung ermöglichen. Die Grundidee hinter Contraction Hierarchies ist es, sogenannte Shortcuts (zusätzliche Kanten) in den Graphen einzufügen und den Graphen in Levelhierarchien einzuteilen, indem jedem Knoten ein Level zugeordnet wird. Alle Abbildungen zur Erläuterung von Contraction Hierarchies (auch zu „Stall-On-Demand“ im Kapitel „Implementierung“) wurden, meist in abgeänderter Form, aus dem Originalpaper [GSSD08] entnommen.

### 2.4.1 Vorverarbeitung

Bevor schnelle Pfadabfragen auf dem Graph ausgeführt werden können findet eine Vorverarbeitung des Ursprungsgraphen statt. Dabei wird zuerst eine Ordnung auf den Knoten definiert, welche im Folgenden als $Level(u)$ für jeden Knoten u bezeichnet wird. Anschließend werden die Knoten in aufsteigender Reihenfolge aus dem Graph entfernt („Contraction“), wodurch zusätzliche Kanten entstehen. Als Beispiel dient Abbildung 2.5: Wird aus einem Pfad (u,v,w) der Knoten v entfernt, wobei $Level(v) < Level(u)$ und $Level(v) < Level(w)$, wird eine neue Kante (u,w) eingefügt. Dies geschieht allerdings nur, wenn (u,v,w) der kürzeste Pfad zwischen dem Knoten $u$ und $w$ ist. Wichtig ist, dass jede Kante die in diesem Prozess entsteht lediglich bereits vorhandene Pfade im Graph repräsentiert. Allerdings ist es möglich, dass dadurch generierte Shortcuts größere Distanzen überbrücken. Das Level eines Knoten spielt nicht nur bei der „Contraction“ eine Rolle, sondern wird auch später beim Berechnen eines kürzesten Pfads wieder aufgegriffen.

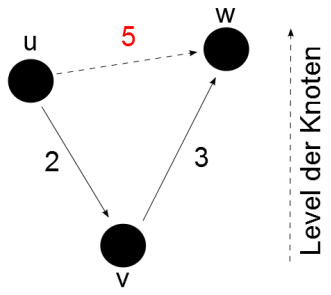



**Abbildung 2.5:** Erstellung einer Shortcut zwischen den Knoten u und w, wenn der Knoten v aus dem Graphen entfernt wird, allerdings nur wenn der kürzeste Pfad über Knoten v verlief.

### 2.4.2 Pfadabfragen

Die Berechnung eines kürzesten Pfads zwischen Knoten s und t kann mit Hilfe eines bidirektionalen Dijkstra Algorithmus berechnet werden. Hierfür wird ein leicht modifizierter Dijkstra Algorithmus zweimal gestartet: Einmal ausgehend vom Startknoten s (Vorwärtssuche) und einmal vom Zielknoten t (Rückwärtssuche). Wichtig hierbei ist, dass von einem Knoten u aus lediglich Kanten betrachtet werden müssen, welche zu einem Knoten v führen für die gilt: $Level(u) < Level(v)$. Dies reduziert den Suchraum enorm und ermöglicht somit eine schnelle Berechnung des Pfads. Nach der Ausführung beider Dijkstrainstanzen müssen schlussendlich alle Knoten betrachtet werden, welche von Vorwärts- und Rückwärtssuche besucht wurden. Gewählt wird der Knoten, dessen addierter Distanzwert aus Vorwärts- und Rückwärtssuche minimal ist.

In Abbildung 2.6 ist die Bestimmung eines Pfads schematisch dargestellt. Darin wird deutlich wie mit jedem weiteren besuchten Knoten auch das Level des Knotens steigen muss. Beide Suchinstanzen, ausgehend vom Start- und Zielknoten, treffen sich an mindestens einem Knoten, wie durch Knoten v in der Abbildung dargestellt.

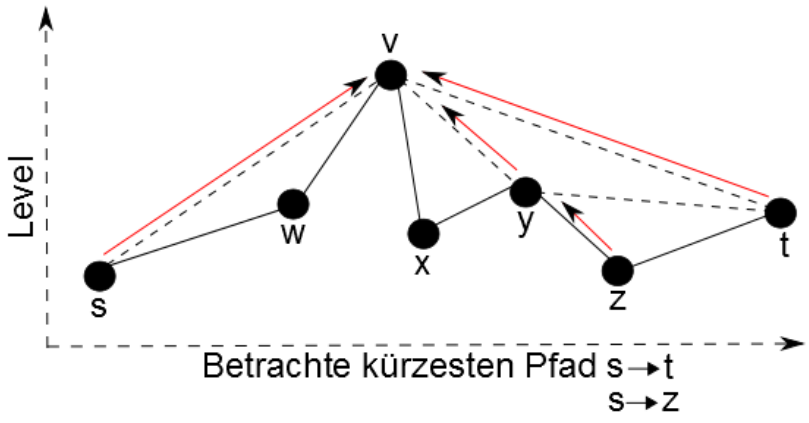



**Abbildung 2.6:** Berechnung des kürzesten Pfads von s nach t sowie s nach z. Gestrichelte Linien stellen Abkürzungen dar, durchgezogene Linien sind Kanten aus dem Graph vor der Vorverarbeitung.

### 2.4.3 Entpacken des CH-Pfads

Da bei der Darstellung des gefundenen Pfads normalerweise keine Shortcuts dargestellt werden sollen (es werden zu große Distanzen überbrückt), sondern die tatsächlichen Kanten, welche durch diese repräsentiert werden, müssen diese aufgelöst werden. Um diesen Schritt einfach durchführen zu können, wird für jeden Shortcut gespeichert, welche zwei Kanten dadurch ersetzt wurden. Der Vorgang kann rekursiv durchgeführt werden bis keine Shortcuts mehr vorhanden sind. Dies wird in Abbildung 2.7 dargestellt. Ein Shortcut mit Distanzwert „9“ wird in zwei Stufen aufgelöst bis lediglich die Originalkanten des Graphen übrig bleiben.

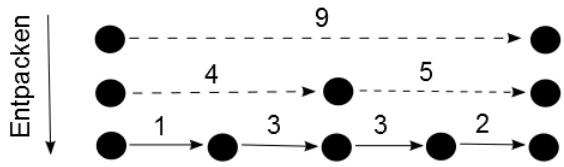



**Abbildung 2.7:** Entpacken eines CH-Pfads. Gestrichelte Pfeile stellen Abkürzungen dar, durchgezogene Pfeile sind vollständig aufgelöste Kanten

## 2.5 Komprimierung von Daten

Durch die Kompression von Daten kann die Übertragungszeit als auch der benötigte Speicherplatz verringert werden. Dies hat den Vorteil, dass Nachrichtenkanäle weniger stark belastet werden oder eventuell teure Speicherhardware vergrößert werden muss. Man unterscheidet verlustfreie und verlustbehaftete Kompressionen. Bei ersterem können die Ursprungsdaten vollständig aus den komprimierten Daten hergestellt werden, wobei dies bei letzterem nicht der Fall ist. Innerhalb dieser Arbeit werden lediglich verlustfreie Verfahren besprochen und verwendet.

### 2.5.1 Lempel-Ziv

Der Lempel-Ziv (LZ77) Algorithmus [ZL06] kann zur verlustfreien Datenkompression genutzt werden. Die Grundidee des Algorithmus ist es, dass Wörter in einem Text mehrfach auftauchen und dadurch mit Hilfe von Referenzen auf das erstmalige Auftauchen ersetzt werden können. Hierfür wird ein Wörterbuch aufgebaut, welches nicht explizit abgespeichert werden muss, sondern beim Einlesen des Datenstroms zur Laufzeit generiert wird.

#### Kompression

Der Pseudocode zur Kompression ist in Algorithmus 2.2 dargestellt. Zuerst wird ein Wörterbuch generiert, welches alle Zeichen der Länge eins in der zu komprimierenden Zeichenfolge enthält. Darauf folgend wird die Zeichenfolge zeichenweise gelesen. Es wird immer versucht möglichst viele Zeichen zu einem Block zusammenzufügen bis der Block nicht mehr im Wörterbuch gefunden werden kann. Zu diesem Zeitpunkt kann eine Referenz auf den Block, ohne das letzte Zeichen, in die Ausgabe geschrieben werden und ein neuer Eintrag im Wörterbuch angelegt werden. Gleichzeitig muss ein neuer Block begonnen werden ab dem Zeichen, welches dazu führte einen neuen Eintrag im Wörterbuch anzulegen.

**Algorithmus 2.2** Lempel-Ziv Kompression

```
procedure KOMPRIMIERE(Zeichenfolge)
    Wörterbuch ← Generiere Eintrag für jedes Zeichen der Länge eins
    Muster ← (leer)
    for all Zeichen ∈ Zeichenfolge do
        if (Muster + Zeichen) ∈ Wörterbuch then
            Muster ← (Muster + Zeichen)
        else
            Wörterbuch ← Wörterbuch ⋃(Muster + Zeichen)
            Ausgabe ← Ausgabe + Referenz(Muster)
            Muster ← Zeichen
        end if
    end for
end procedure
```

Zum Verständnis trägt folgendes Beispiel mit der Zeichenkette „ababbaabbaabba“, welche komprimiert werden soll, bei (siehe Tabelle 2.3). Das erste Zeichen „a“ wird gelesen, für das bereits ein Eintrag im initialisierten Wörterbuch vorhanden ist. Deshalb wird das nächste Zeichen gelesen und an die bisherige gelesene Zeichenkette angehängt. Für „ab“ gibt es noch keinen Eintrag im Wörterbuch, sodass dieser erstellt wird. Die Referenz auf „a“ wird daraufhin in die Ausgabe geschrieben, „b“ bleibt in einem Puffer gespeichert.

Das nächste Zeichen ist ein „a“, wobei „ba“ ebenfalls noch nicht gelesen wurde und somit ein neuer Eintrag erstellt wird. Die Referenz auf „b“ kann in die Ausgabe geschrieben werden, wobei „a“ gepuffert wird. In der Zeichenkette folgt nun ein „b“. Da die Zeichenkette „ab“ schon gelesen wurde sowie das nächste Zeichen ein b ist und „abb“ noch nicht vorhanden ist, kann eine Referenz auf „ab“ in die Ausgabe geschrieben werden („abb“ wird im Wörterbuch vermerkt) und das nächste Zeichen „b“ kann gepuffert werden.

Im nächsten Schritt kann „ba“ als längste Zeichenkette identifiziert werden („baa“ wird im Wörterbuch vermerkt), welche bereits im Wörterbuch vorhanden ist. Daraufhin wird mit „abb“ zum ersten Mal eine Teilkette der Länge drei gefunden, welche bereits gelesen werden konnte. „abba“ kann außerdem ins Wörterbuch aufgenommen werden. Das folgende „a“ führt zur Aufnahme von „aa“ ins Wörterbuch. Die restlichen Zeichen „abba“ können durch eine einzige Referenz ersetzt werden, da diese schon im Wörterbuch vorhanden sind.

| | Index | Eintrag | |
|---|---|---|---|
| | 0 | a | Initialisierung |
| | 1 | b | |
| | 2 | ab | Zur Laufzeit erzeugt |
| $\underbrace{a}_{0}\ \underbrace{b}_{1}\ \underbrace{ab}_{2}\ \underbrace{ba}_{3}\ \underbrace{abb}_{4}\ \underbrace{a}_{0}\ \underbrace{abba}_{6}$ | 3 | ba | |
| | 4 | abb | |
| | 5 | baa | |
| | 6 | abba | |
| | 7 | aa | |

**Tabelle 2.3:** Beispiel der Kompression einer Zeichenkette. Links die Zeichenkette, Rechts das Wörterbuch.

### Dekomprimierung

Bei der Dekomprimierung wird genau dasselbe Wörterbuch wie bei der vorangehenden Komprimierung aufgebaut. Dies geschieht parallel zum Einlesen des Datenstroms. Der Pseudocode ist in Algorithmus 2.3 dargestellt. Dabei werden nun die Referenzen auf das Wörterbuch wieder in die ursprüngliche unkomprimierte Zeichenfolge aufgelöst. Es findet, genau wie bei der Kompression, eine Vorinitialisierung des Wörterbuchs statt. Dann wird die komprimierte Referenzfolge eingelesen. Befindet sich eine Referenz bereits im Wörterbuch, so wird die dazugehörende Zeichenfolge ausgegeben. Gleichzeitig wird diese gepuffert, da eventuell mit Hilfe der nächsten Referenz ein neuer Eintrag im Wörterbuch angelegt werden kann. Das Prinzip erklärt sich wieder am Besten an Hand eines

**Algorithmus 2.3** Lempel-Ziv Dekomprimierung

**procedure** DEKOMPRIMIERE($Referenzfolge$)
  Wörterbuch $\leftarrow$ $Generiere\ Eintrag\ für\ jedes\ Zeichen\ der$ Länge $eins$
  $letzteZeichenfolge \leftarrow (leer)$
  **for all** $Referenz \in Referenzfolge$ **do**
    **if** $Referenz \in$ Wörterbuch **then**
      Wörterbuch $\leftarrow$ Wörterbuch $\bigcup(letzteZeichenfolge +$ Wörterbuch$[Referenz][0])$
    **else**
      Wörterbuch $\leftarrow$ Wörterbuch $\bigcup(letzteZeichenfolge + letzteZeichenfolge[0])$
    **end if**
    $Ausgabe \leftarrow Wörterbuch[Referenz]$
    $letzteZeichenfolge \leftarrow Wörterbuch[Referenz]$
  **end for**
**end procedure**

Beispiels, wobei die im letzten Abschnitt in Tabelle 2.3 komprimierte Zeichenkette dekomprimiert werden soll. Dies wird in Tabelle 2.4 beschrieben.

Die ersten beiden Referenzzeichen(„0“, „1“) sind bereits aus dem initialen Wörterbuch bekannt. Zusätzlich wird beim Lesen des zweiten Zeichens „1“ ein neuer Eintrag „ab“ angelegt, da dieser noch nicht vorhanden ist. Das nächste eingelesene Zeichen „2“ referenziert genau diesen zuvor angelegten Wörterbucheintrag „ab“. Wieder wird auch hier ein neuer Eintrag angelegt: „b“ aus dem zweiten gelesenen Zeichen und das erste Zeichen aus „ab“. So entsteht „ba“ (Index 3) im Wörterbuch. Diese wird sofort beim nächsten eingelesenen Zeichen benötigt, womit 3 zu „ba“ aufgelöst wird. Im Wörterbuch entsteht damit „abb“ (Index 4). Dieser wird anschließend zum Auflösen des nächsten Zeichens aus dem Datenstrom „4“ benutzt und „abba“ wird als Eintrag angelegt. „0“ und „6“ sind im Wörterbuch bereits vorhanden und ergeben den restlichen Zeichenstring „aabba“.

| | Index | Eintrag | |
|---|---|---|---|
| | 0 | a | Initialisierung |
| | 1 | b | |
| | 2 | ab | Zur Laufzeit erzeugt |
| $\underbrace{0}_{a}\ \underbrace{1}_{b}\ \underbrace{2}_{ab}\ \underbrace{3}_{ba}\ \underbrace{4}_{abb}\ \underbrace{0}_{a}\ \underbrace{6}_{abba}$ | 3 | ba | |
| | 4 | abb | |
| | 5 | baa | |
| | 6 | abba | |
| | 7 | aa | |

**Tabelle 2.4:** Beispiel der Dekompression einer Zeichenkette. Links die komprimierte Zeichenkette, Rechts das Wörterbuch.

### 2.5.2 Huffman-Kodierung

Die Huffman-Kodierung [Huf52] ist eine verlustfreie Entropiekodierung, wodurch Zeichen im zu komprimierenden Text durch Codewörter variabler Länge ersetzt werden. Für jedes Zeichen muss ein binäres Codewort gefunden werden, um dieses zu kodieren. Die Idee ist Wörter, welche häufiger im Text vorkommen, mit kürzeren Codewörtern zu repräsentieren als Wörter, welche seltener vorkommen.

Zur Bestimmung der Codewörter wird zuerst jedes Zeichen und dessen Auftrittshäufigkeit bestimmt (für endliche Zeichenfolgen). Um das Verfahren zu verdeutlichen, soll die Zeichenfolge „xyzzxyzxzzyzx“ als Beispiel genutzt werden. In der folgenden Tabelle sind die vorkommenden Zeichen sowie deren Häufigkeit eingetragen. In den folgenden Schritten wird nun ein binärer Baum aufgebaut, aus welchem die Codewörter abgelesen werden können.

| Zeichen | Häufigkeit |
|---|---|
| x | 4 |
| y | 3 |
| z | 6 |

Zuerst wird für jedes Zeichen ein Knoten, der ebenfalls dessen Häufigkeit enthält, generiert. Dies ist in Abbildung 2.8 links dargestellt. Danach werden schrittweise immer zwei Knoten, mit den jeweils kleinsten Häufigkeiten, zusammengefügt. Dies geschieht indem ein neuer Knoten erzeugt wird, an den diese als Kinder angehängt werden. Der Vaterknoten enthält dann deren addierte Häufigkeiten. Das Zusammenfügen der ersten beiden Knoten ist in Abbildung 2.8 Mitte verbildlicht. Dies geschieht so lange, bis alle einzelnen Knoten in einem Baum zusammengeführt wurden. Das Endergebnis ist in Abbildung 2.8 rechts dargestellt.

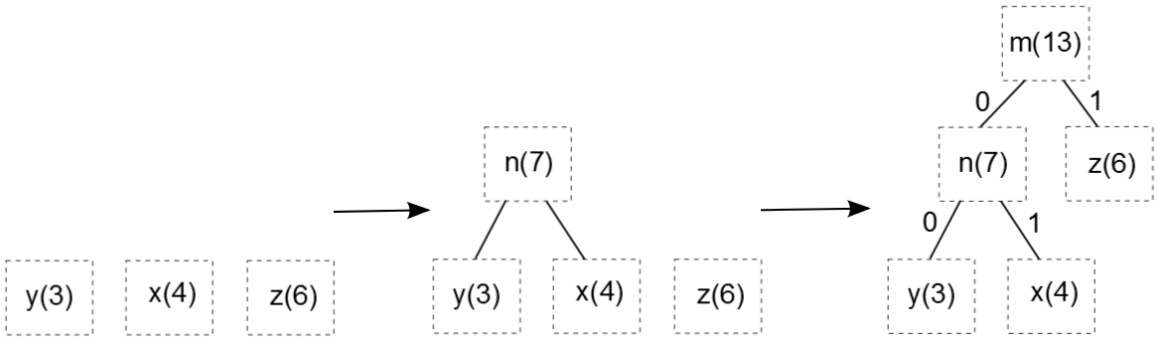



**Abbildung 2.8:** Links: Jedes Zeichen und dessen Häufigkeit als Knoten. Mitte: Zusammenfügen der beiden Knoten mit geringster Häufigkeit. Rechts: Erneutes Zusammenfügen der beiden Knoten mit geringster Häufigkeit.

Nun müssen nur noch die Codewörter abgelesen werden. Vom Wurzelknoten wird zu den Blattknoten abgestiegen und es wird mit einem leeren Codewort begonnen. Steigt man dabei in einen linken Teilbaum ab, wird dem Codewort eine „0“ hinzugefügt ansonsten eine „1“. Dadurch entstehen folgende Codewörter:

| Zeichen | Codewort |
|---|---|
| x | 01 |
| y | 00 |
| z | 1 |

Hierbei zeigt sich, dass die Huffman-Kodierung ein Präfix-Code ist. Das bedeutet, dass kein gültiges Codewort ein Präfix eines anderen Codeworts sein kann.

### 2.5.3 Deflate-Algorithmus

Der Deflate Algorithmus[Deu96] kann zur verlustlosen Datenkompression genutzt werden. Dieser wurde ursprünglich für das bekannte ZIP-Format entwickelt und ist ebenfalls im bekannten Bildformat PNG enthalten. Dieser Algorithmus führt nun die beiden Techniken, welche in den letztem beiden Abschnitten erklärt wurden, zusammen. Hierbei werden Lempel-Ziv und die Huffman-Kodierung kombiniert, um noch bessere Kompressionsraten zu erreichen. Das Format ist dabei unabhängig von eingesetzter Hardware, als auch von Software wie Betriebs- und Dateisystem. Besonders interessant wird der Algorithmus, da er keinem Patent unterliegt. So ist dieser in der populären „zlib“ [GA95], einer Programmbibliothek zur Datenkompression, implementiert, wofür zum Beispiel in Java eine Schnittstelle vorhanden ist.

Der Deflate Algorithmus bietet dabei Freiheitsgrade bei der Kompression. Beispielweise kann die Reihenfolge der Anwendung von Lempel-Ziv und Huffman Kodierung variiert werden, um die Kompression auf die vorhandenen Daten abzustimmen.

# 3 Implementierung

Die Zielsetzung der Implementierung ist auf einem zuvor komprimierten Graphen kürzeste Wege zwischen zwei Punkten effizient zu berechnen und den berechneten Pfad grafisch darzustellen. Hierfür stehen geographische Daten aus OpenStreetMap [Coa04], welche die Vorverarbeitung der Contraction Hierarchies bereits durchlaufen hatten, sowie eine grafische Oberfläche, in welche der Pfad gezeichnet werden konnte, bereit. Diese Zielsetzung wurde schrittweise erreicht, indem aufeinanderfolgende Versionen kontinuierlich verbessert wurden.

Begonnen wurde mit einer simplen Implementierung des Dijkstra Algorithmus [Dij59], welcher auf dem im Arbeitsspeicher gepufferten Kartenmaterial ausgeführt werden konnte. Um die korrekte Umsetzung des Programmcodes überprüfen zu können, erfolgte bereits hier eine Anbindung an die grafische Oberfläche, damit die berechneten Pfade visuell überprüft werden konnten. Anfänglich wurden Knoten und Kanten objektorientiert mit Hilfe von Java Objekten gespeichert. Dies wurde aus Performanzgründen frühzeitig durch Arrays aus primitiven Datentypen ersetzt.

Um die Laufzeit der Berechnung von mehreren Sekunden in den Bereich von einigen Millisekunden zu senken, sollten im nächsten Schritt die in Abschnitt 2.4 vorgestellten Contraction Hierarchies eingebaut werden. Dafür musste lediglich der Dijkstra Algorithmus angepasst werden, wie in Abschnitt 2.4.2 beschrieben.

## 3.1 Senkung des Arbeitsspeicherverbrauchs

Da das Zwischenspeichern des eingelesenen Graphen im Arbeitsspeicher Kapazität im Bereich von mehreren Gigabyte verbraucht (und dies spätestens bei beispielsweise Smartphones problematisch wird), sollte im nächsten Schritt der Graph vollständig auf einem externen Speichergerät verbleiben. Dies führt zu einer Schwierigkeit: Lesen von einem externen Speicher ist im Vergleich zum internen Arbeitsspeicher um circa Faktor 1000 (oder sogar mehr) langsamer. Eine Berechnung von kürzesten Pfaden wird erst dann in benutzbaren Zeitspannen möglich, wenn Zugriffe auf den Externspeicher so oft wie möglich vermieden werden.

Der erste Lösungsansatz ist hierbei die Verwendung von Contraction Hierarchies, da hierbei die Anzahl besuchter Knoten im Graph und somit auch teurer I/O-Operationen drastisch gesenkt werden können. Der zweite Ansatz des Caching wird in Abschnitt 3.5 besprochen.

Außerdem verwendet Dijkstra bei der Berechnung des Pfads drei Arrays („settled“, „dist“ und „previous“), welche für jeden Knoten des Graphs einen Index bereitstellen müssen. Für die beiden Integer-Arrays und ein Boolean-Array werden bei circa 15 Millionen Knoten somit $15 \cdot 10^6 \cdot (32 + 32 + 1) = 975000000\ Bit = 121875000\ Byte \approx 122\ Megabyte$ verbraucht. Diese können durch Hashmaps

ersetzt werden, um den Speicherverbrauch auf einen verschwindend geringen Anteil zu senken. Die Zugriffszeiten einer Hashmap sind zwar deutlich langsamer als die eines einfachen Arrays (Array Look-Up versus Berechnung eines Hashs), allerdings wird der Großteil der Berechnung für wesentlich langsamere I/O-Operationen verbraucht, wodurch dies nicht besonders ins Gewicht fällt.

## 3.2 Format der Graphdaten

Die grundlegende Struktur zur Repräsentation des Graphen ist ein Adjazenz-Array mit Offsetwerten. Die Umsetzung erfolgt dabei wie in Abbildung 3.1. Jeder Knoten erhält hierfür einen eindeutigen Index. Eine Kante ist gerichtet und beschreibt einen Start- und Zielknoten sowie ein Gewicht. Die Kanten liegen aufeinanderfolgend sortiert nach deren Startknoten in einer Datei. Um bestimmen zu können ab welcher Kante die erste Kante eines Knoten beginnt wird ein Offset-Array benötigt, welches zusätzlich gespeichert wird. Für jeden Knoten gibt es einen Offsetwert (Kanten-ID), welcher bestimmt ab welcher Kante in der Datei dessen ausgehende Kanten beginnen.

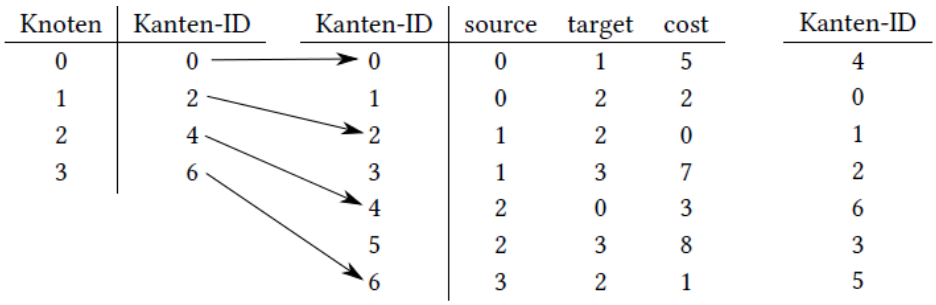



**Abbildung 3.1:** Links: Offset-Tabelle; Mitte : Kanten; Rechts : Sortieren der Kanten nach Zielknoten

Da Dijkstra in Kombination mit Contraction Hierarchies bidirektional ausgeführt wird, werden ebenfalls alle Kanten sortiert nach deren Zielknoten benötigt. Eine einfache Möglichkeit wäre alle Kanten zu duplizieren und sie nach deren Zielknoten zu sortieren. Dies benötigt allerdings den doppelten Speicherplatz der Kanten. Eine weitere Möglichkeit ist, lediglich die Kanten-IDs zu sortieren. Die Funktionsweise ist in Abbildung 3.1 Rechts dargestellt. Hier sind die Kanten-IDs aus Abbildung 3.1 Mitte sortiert nach Zielknoten eingetragen.

### 3.2.1 Anfängliches Format

#### Knotenformat

Die Knoten des Graphen sind in einem einheitlichem Format in einer Datei abgelegt. Für jeden Knoten sind dessen geographische Daten als auch dessen Level nötig. Für die geographischen Daten sind zwei Doublewerte der Größe 8 Byte nötig, also addiert 16 Byte. Für das Level ist lediglich ein Short und somit 2 Byte nötig. Somit wird für ein Knoten 18 Byte Speicherplatz benötigt. In Tabelle 3.1 sind mehrere Knoten beispielhaft dargestellt.

| Latitude | Longitude | Level |
|---|---|---|
| 48.8501024 | 9.1548904 | 0 |
| 49.1362976 | 8.4124400 | 1 |
| 49.0074016 | 8.4269200 | 1 |
| 49.0097984 | 8.4113504 | 2 |
| ... | ... | ... |

**Tabelle 3.1:** Anfängliches Format der Knoten

#### Kantenformat

Genauso wie die Knoten sind auch die Kanten in einem bestimmten Format gespeichert. Eine Kante ist definiert durch eine Quelle, ein Ziel und die Kosten der Kante. Hinzu kommt durch die Verwendung von Contraction Hierarchies und den hierbei eingefügten Abkürzungen in den Graphen die Verwendung von Skip-Werten. Ist eine Kante eine Abkürzung, dann stehen in den Spalten „SkipA“ und „SkipB“ die IDs der beiden Kanten, die durch diese Abkürzung repräsentiert werden. Steht in beiden Spalten „-1“, so ist die Kante keine Abkürzung. Die Skip-Werte werden zum Auflösen der Abkürzungen benötigt, wie in Abschnitt 2.4.3 bereits beschrieben. In Tabelle 3.2 sind mehrere Kanten beispielhaft eingetragen.

| Source | Target | Cost | SkipA | SkipB |
|---|---|---|---|---|
| 0 | 1 | 57 | -1 | -1 |
| 0 | 2 | 57 | -1 | -1 |
| 1 | 3 | 163 | -1 | -1 |
| 1 | 9 | 70 | -1 | -1 |
| 2 | 8 | 94 | -1 | -1 |
| 3 | 11 | 198 | 22 | 34 |
| ... | ... | ... | ... | ... |

**Tabelle 3.2:** Anfängliches Format der Kanten

### 3.2.2 Optimiertes Format

Um ein verbessertes Caching (näheres in Abschnitt 3.5) zu ermöglichen, werden Knoten und Kanten in unterschiedliche Dateien abgelegt. Da die Skip-Werte bei der Berechnung des kürzesten Pfads nicht benötigt werden, sondern erst im Vorgang danach, können die Skip-Werte ebenfalls in eine andere Datei geschrieben werden. Dadurch verändert sich das Format der Kanten.

Eine zweite Veränderung ergibt sich aus folgendem Vorgang: Bei der Relaxierung von Kanten werden die Level der Start- und Zielknoten benötigt. Da dies mehrere Lesezugriffe an unterschiedlichen Stellen in Knoten- und Kantendatei benötigt, sollten die Level der Knoten direkt in die Kanten aufgenommen werden. Dadurch kann mit einem Lesezugriff die gesamte Information gelesen werden. Um redundante Daten zu vermeiden können die Levelwerte in der Knotendatei entfernt werden. Muss das Level eines

Knoten bestimmt werden, muss lediglich zu dessen erster ausgehender Kante gesprungen werden und das Level des Startknoten extrahiert werden.

### Knoten

Durch die eben beschriebenen Verbesserungen ergibt sich für die Knoten des Graphen das Format aus Tabelle 3.3.

| Latitude | Longitude |
|---|---|
| 48.8501024 | 9.1548904 |
| 49.1362976 | 8.4124400 |
| 49.0074016 | 8.4269200 |
| 49.0097984 | 8.4113504 |
| ... | ... |

**Tabelle 3.3:** Verbessertes Format der Knoten

### Kanten

Für die Kanten kommt es durch die Veränderungen zu folgendem Format wie in Tabelle 3.4 gezeigt.

| Source | Target | Cost | Level Source | Level Target |
|---|---|---|---|---|
| 0 | 1 | 57 | 0 | 0 |
| 0 | 2 | 57 | 0 | 1 |
| 1 | 3 | 163 | 0 | 1 |
| 1 | 9 | 70 | 0 | 2 |
| 2 | 8 | 94 | 1 | 2 |
| 3 | 11 | 198 | 1 | 3 |
| ... | ... | ... | ... | ... |

**Tabelle 3.4:** Verbessertes Format der Kanten

## 3.3 Optimierungen zu Contraction Hierarchies

Die bereits relativ geringe Anzahl an Knoten, welche mit Hilfe von Contraction Hierarchies besucht werden müssen, lässt sich weiter verringern. Dies ist im Bezug auf die Implementierung besonders wichtig, um die Anzahl langsamer I/O-Operationen weiter zu verkleinern.

### 3.3.1 Stall-On-Demand

Diese Optimierung wird in der Veröffentlichung [GSSD08] zu Contraction Hierarchies beschrieben. Am Besten erklärt sich diese Technik an Hand des Beispiels in Abbildung 3.2. Bei der Berechnung des kürzesten Pfads werden nicht alle Kanten eines Knotens relaxiert wodurch es möglich ist, dass ein Knoten nur über einen suboptimalen Pfad erreicht wird. Da dies nicht gewünscht ist, kann die Suche an solch einem Knoten abgebrochen werden. In Abbildung 3.2 gilt $Level(u) < Level(t)$, dadurch wird der Knoten u mit suboptimaler Distanz 4 über den Pfad zwischen s und u festgelegt. Alle folgenden Knoten (zum Beispiel v), welche von u aus besucht werden sind somit auch suboptimal. Somit kann die Suche bei Knoten u abgebrochen werden.

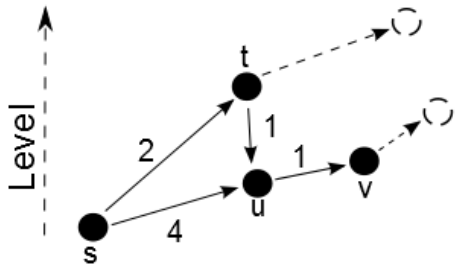



**Abbildung 3.2:** Beispiel von Stall-On-Demand an Hand des Knoten u

Die Umsetzung in Programmcode kann in Listing 3.1 betrachtet werden. Dieser Codeausschnitt wird für jeden Knoten ausgeführt, der aus der Prioritätswarteschlange genommen wird. Dabei wird über alle einfallenden Kanten des aktuellen Knoten iteriert. Sobald eine einfallende Kante auf diesen Knoten gefunden wird, deren Distanz des Startknotens addiert mit den Kosten der Kante einen geringeren Distanzwert ergibt als die des aktuellen Knotens, wird die Suche an diesem Knoten abgebrochen.

**Listing 3.1** Stall-On-Demand

```
int element = Q.poll();
stalled = false;
for(int i=start;i>=end;i--){
      if(dist[element] > (dist[graph.source[i]] + graph.cost[i])){
            stalled = true;
            break;
      }
}
if(stalled){continue;}
```

### 3.3.2 Sortierung der Kanten

Durch die Verwendung von Contraction Hierarchies wurde eine weitere Eigenschaft für Knoten eingeführt: Level. Wie bereits in Abschnitt 2.4 beschrieben, müssen von einem Knoten ausgehend lediglich Kanten relaxiert werden, welche zu einem Knoten mit höherem Level führen. Diese Tatsache kann zur Optimierung genutzt werden, um nicht über alle Kanten eines Knoten iterieren zu müssen. Werden alle Kanten eines Knotens ab- oder aufsteigend nach deren Level sortiert, so kann die Iteration über die Kanten abgebrochen werden, sobald eine Kante relaxiert wird die zu einem Knoten mit kleinerem Level führt, da alle folgenden Kanten lediglich zu Knoten führen können mit kleinerem Level.

Einen Auszug der Implementierung kann in Listing 3.2 betrachtet werden. Die Kanten eines Knoten sind hier absteigend nach deren Level sortiert. Dadurch wird bei der letzten Kante begonnen. In Zeile 2 der for-Schleife wird die Iteration abgebrochen, sobald das Ziel der Kante ein niedrigeres Level besitzt.

**Listing 3.2** Abbruch der Iteration über Kanten eines Knotens sobald ein Level gefunden wurde, welches kleiner ist als das des aktuellen Knotens.

```
int element = Q.poll();
for(int i=start; i>=end;i--){
      int edgeTarget = graph.target[i];
      if(graph.level[element] > graph.level[edgeTarget]){break;}

      if(!settled[edgeTarget]){
            int alternative = dist[element] + graph.cost[i];
            if(alternative < dist[edgeTarget]){
                  dist[edgeTarget] = alternative;
                  previous[edgeTarget] = element;
                  Q.add(edgeTarget);
            }
      }
}
```

## 3.4 Komprimierung

Zu diesem Zeitpunkt ist es mit Hilfe der Implementierung möglich kürzeste Wege zu berechnen, wobei der Graph vollständig auf einem externen Speichergerät verbleiben kann. Nun kann der Graph komprimiert und der Programmcode angepasst werden, um den Graph bei Bedarf blockweise zu dekomprimieren.

Zur Kompression von Daten bietet Java das Modul Deflater [def], auf welches innerhalb dieser Implementierung zurückgegriffen wird.

Der grundsätzliche Ansatz ist eine bestimmte Anzahl von Elementen, zum Beispiel Knoten, Kanten oder Skip-Werte zu einem Block zusammenzufassen und mit Hilfe des Deflaters zu komprimieren. Muss innerhalb der Berechnung des Pfads zwischen zwei Knoten beispielsweise eine bestimmte Kante geladen werden, muss vorher der gesamte Block mit Kanten dekomprimiert werden.

Der Vorgang der Kompression läuft in Java auf Byte-Ebene ab. In Listing 3.3 ist hierzu der Code dargestellt. Die Variable „blockSize“ repräsentiert dabei die zuvor erwähnte Größe eines Blocks. Dieses Beispiel ist aus der Kompression der Kanten genommen, deshalb enthält die Variable „edgeWidthInBytes“ die Breite einer einzelnen Kante in Byte. Die Daten werden dann blockweise aus der Datei gelesen mit „inputFILE.read(block)“, der Komprimierer wird initialisiert und die komprimierten Daten in „output“ gespeichert, welche dann in eine extra Datei geschrieben werden.

**Listing 3.3** Kompression einer Datei in Java mit Hilfe des Deflater. Zusätzlich wird eine Mapping-Datei erstellt, welche speichert welcher unkomprimierte Block in der komprimierten Datei an welcher Byte Position beginnt, da die komprimierten Blöcke keine feste Länge haben.

```
byte[] block = new byte[blockSize*edgeWidthInBytes];
byte[] output = new byte[blockSize*edgeWidthInBytes];
int numberOfBytes = inputFILE.read(block);

while(numberOfBytes != -1){ // -1 => Dateiende
      blockStartFILE.writeInt(position);
      compresser.setInput(block, 0, numberOfBytes);
      compresser.finish();
      int compressedBytes = compresser.deflate(output);
      outputFILE.write(output, 0, compressedBytes);
      position += compressedBytes;
      numberOfBytes = inputFILE.read(block);
      compresser.reset();
}

compresser.end();
```

Durch die Komprimierung entsteht das Problem, dass man nicht mehr bestimmen kann wo ein unkomprimierter Block in der komprimierten Datei beginnt. Dieses Problem entsteht dadurch, dass ein unkomprimierter Block immer die selbe Länge besitzt (zum Beispiel Blockgröße · Kantenbreite Bytes). Werden unterschiedliche Blöcke komprimiert, kann ein Block größer sein als ein anderer. Dieser Umstand ist in Abbildung 3.3 dargestellt. Um später bestimmen zu können, welcher Block in der komprimierten Datei an welcher Byteposition beginnt, muss beim Kompressionsvorgang diese Abbildung gespeichert werden. Dies geschieht in der Zeile „blockStartFILE.writeInt(position)“.

Welchen Block man für eine bestimmte Kante laden muss, lässt sich somit einfach durch die Division der Kanten-ID mit der Blockgröße berechnen sowie dem anschließenden Auslesen der Byte-Position dieses Blocks aus der Mapping-Datei. Somit gilt: Blocknummer$= \lfloor \frac{\text{KantenID}}{\text{Blockgröße}} \rfloor$

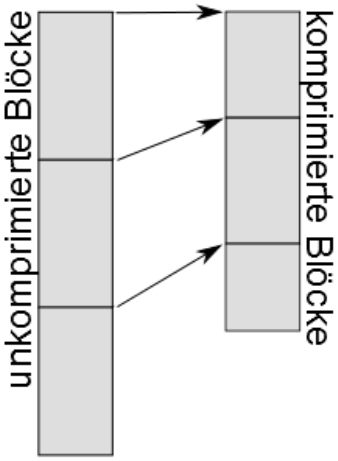



**Abbildung 3.3:** Abbildung der Blocknummer auf die erste Byteposition eines komprimierten Blocks.

## 3.5 Caching von Speicherblöcken

Die Verwendung von Contraction Hierarchies und dessen Optimierungen aus Abschnitt 3.3 verringern bereits durch das Besuchen weniger Knoten die Anzahl langsamer I/O-Operationen. Dies lässt sich jedoch weiter optimieren, indem Teile des Graphen im deutlich schnelleren Arbeitsspeicher gepuffert werden.

Das Prinzip des Caching wird in Computern an vielen Stellen eingesetzt, um teure Lese- oder Schreibzugriffe auf langsame Speichergeräte zu verhindern. Dies geschieht oft in mehreren Stufen, was in Abbildung 3.4 verdeutlicht wird.

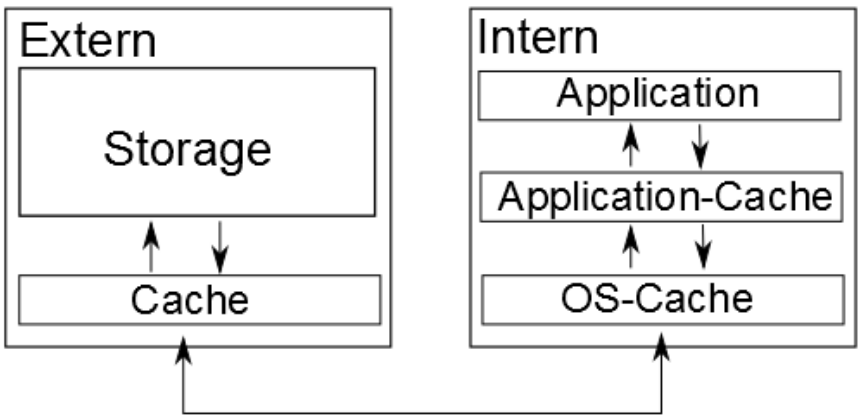



**Abbildung 3.4:** Beispielhafte Darstellung der Cache Mechanismen innerhalb eines Computersystems. Zugriffe auf Speicher erfolgen nie direkt auf das Medium, sondern über eine oder mehrere Cacheschichten.

Jede Anwendung implementiert, falls nötig, sein eigenes Caching (Abbildung 3.4 rechts) und behält Daten, welche einmal von einem langsamen Speicher gelesen wurden im Arbeitsspeicher zwischengespeichert, um bei wiederholtem Lesen dieser Daten diese deutlich schneller zu laden. Zusätzlich stellt das Betriebssystem einen zentralen Cache bereit, auf den mehrere Anwendungen gleichzeitig zugreifen können (Abbildung 3.4, OS-Cache). Der Vorteil liegt hier darin, dass eventuell mehrere Anwendungen auf die selben Dateien zugreifen und somit nicht jede Anwendung einen eigenen Cache besitzt, wodurch Speicherplatz gespart wird. Dadurch wird eventuell redundantes Caching von Datenblöcken verhindert. Ein Nachteil dabei kann allerdings sein, dass sich mehrere Anwendungen hierbei beeinflussen können. Ein Cache hat immer eine maximale Größe, wodurch irgendwann Blöcke verworfen werden müssen wenn der Cache voll ist, um Platz für neue Blöcke zu machen. Dadurch kann durch Lesezugriffe der einen Anwendung ein Block verworfen werden, welchen eine andere Anwendung in den Cache geladen hat und zu einem späteren Zeitpunkt noch benötigen könnte.

Die letzte Cacheschicht befindet sich in Abbildung 3.4 direkt im externen Gerät in Form eines Hardware Cache. Dieser ist deutlich kleiner als das Speichermedium des Geräts, allerdings können daraus Daten mit hoher Geschwindigkeit gelesen werden. Wie man erkennen kann läuft ein Lesezugriff einer Anwendung in folgenden Stufen ab: Es wird zuerst im, falls vorhandenen, eigenen Cache der Anwendung nach den gewünschten Daten gesucht. Sind diese dort nicht vorhanden wird beim Betriebssystem angefragt, ob dieses die Daten noch zwischengespeichert hat. Falls dies auch nicht der Fall ist, wird beim internen Cache des Zielgeräts angefragt. Erst wenn die Daten auch hier nicht vorhanden sind, wird ein langsamer Zugriff auf den Speicher durchgeführt.

Die Implementierung setzt auf zwei Cacheebenen: In der ersten Schicht werden komprimierte Blöcke aus den Dateien gepuffert. In der zweiten Schicht werden bereits dekomprimierte Blöcke zwischengespeichert. Für die erste Schicht wird mit Hilfe von „MappedByteBuffer“ [mbb] der Cache des Betriebssystem genutzt (Abbildung 3.4, OS-Cache). Die Initialisierung erfolgt wie in Listing 3.4. Danach kann die Datei an beliebigen Positionen gelesen werden und das Betriebssystem übernimmt transparent im Hintergrund das Puffern von Blöcken der Datei.

**Listing 3.4** Initialisierung eines MappedByteBuffer

```
FileChannel channel = new RandomAccessFile(path, "r" ).getChannel();
long size = channel.size();
MappedByteBuffer memoryMapping = channel.map(FileChannel.MapMode.READ_ONLY, 0, size);
```

Für die zweite Stufe des Caching muss eine Struktur zum Speichern von dekomprimierten Blöcken entworfen werden. Die Umsetzung eines Cache zum Zwischenspeichern von Blöcken lässt sich in Java mit Hilfe einer einfachen Struktur umsetzen. Der minimale Code wird in Listing 3.5 gezeigt. Benötigt wird hierzu lediglich eine Hashmap [has], welche als Schlüssel die jeweilige Blocknummer entgegen nimmt und damit den Block als ByteBuffer zurückgibt.

Ist die Kapazität eines Cache ausgelastet müssen Datenblöcke gelöscht werden, um Platz für neue Blöcke zu machen, was in Listing 3.5 bisher noch nicht gelöst wird. Für die Bestimmung des Blocks, welcher gelöscht werden soll, existieren unterschiedliche Strategien. Zwei möglich Strategien, die auch in der Implementierung eingesetzt werden, sollen hier kurz beschrieben werden.

**Listing 3.5** Klasse zum Cachen von Blöcken

```
public class BlockCache {
      private HashMap<Integer, ByteBuffer> cachedBlocks;

      public ByteBuffer getBlock(int blockNumber){
            ByteBuffer block = cachedBlocks.get(blockNumber);
            if(block == null){
                  // Block nicht vorhanden => laden und cachen
                  // Variable "‘block"’ auf den geladenen Block setzen
            }
            return block;
      }
}
```

### Random Replacement

Eine einfache Methode besteht darin einen Block völlig zufällig auszuwählen falls Speicherplatz benötigt wird und diesen dann zu entfernen. Der Vorteil ist hierbei, dass keine zusätzliche Information über Blöcke gespeichert werden muss, zum Beispiel wie oft ein Block gelesen wurde oder wie lange sich der Block bereits im Cache befindet. Ein Ausschnitt der Umsetzung in Java kann in Listing 3.6 betrachtet werden. Alle Blocknummern, welche aktuell mit Hilfe der Hashmap auf den tatsächlichen Block abgebildet werden, sind zusätzlich im Array „blockNumbersOfCachedBlocks“ gespeichert. Ist der Cache voll wird aus diesem Array eine Blocknummer gewählt, der Block gelöscht und durch den neuen Block ersetzt.

**Listing 3.6** Zufälliges ersetzen von Blöcken in einem Cache

```
if(numberOfCachedBlocks == MAX_SIZE){
      int idx = (int)(Math.random() * MAX_SIZE);
      cachedBlocks.remove(blockNumbersOfCachedBlocks[idx]);
      blockNumbersOfCachedBlocks[idx] = blockNumber;
      cachedBlocks.put(blockNumber, block);
}
```

### Least Recently Used Replacement (LRU)

Eine etwas aufwändigere Methode ist den Block zu entfernen, der am längsten nicht gelesen wurde. Hier müssen zusätzliche Informationen gespeichert werden, um den richtigen Block bestimmen zu können. Java bietet hierzu eine Erweiterung der einfachen Hashmap durch die „LinkedHashMap“ [lin]. Dabei werden die Elemente der Hashmap zusätzlich in einer verketteten Liste verwaltet, nämlich geordnet nach deren letztem Zugriff.

In Listing 3.7 wird die Verwendung der „LinkedHashmap“ [lin] gezeigt. Einfacherweise wird eine neue Klasse erstellt, welche von LinkedHashMap erbt. Der Konstruktor erlaubt die maximale Größe des Cache festzulegen. Zusätzlich muss lediglich die Methode „removeEldestEntry()“ überschrieben

werden. Diese wird nach jedem Aufruf von „put()“ der Hashmap aufgerufen. Gibt diese Methode „true“ zurück wird automatisch das Element gelöscht, welches am Längsten nicht gelesen wurde.

**Listing 3.7** Löschen des Block, der am Längsten nicht gelesen wurde

```
public static class LRU<K,V> extends LinkedHashMap<K,V> {
            private int CACHE_SIZE;
            private static float LOAD_FACTOR = 0.75f;
            private static boolean ACCESS_ORDER = true;

            public LRU(int size) {
                  super(size, LOAD_FACTOR, ACCESS_ORDER);
                  this.CACHE_SIZE = size;
            }

            protected boolean removeEldestEntry(Map.Entry eldest) {
                  return size() > CACHE_SIZE;
            }
}
```

# 4 Evaluierung

Wie in Kapitel 3 deutlich wurde gibt es Parameter für die Kompression und das Verhalten des Caching, welche den Speicherbedarf als auch die Berechnungszeit von kürzesten Pfaden beeinflussen. Für die Kompression ist dies hauptsächlich die Blockgröße und leichte Modifikationen des Kompressionsverfahrens. Beim Caching kann die Anzahl der Blöcke, welche im Arbeitsspeicher gehalten werden, die Berechnungszeit stark beeinflussen. Ebenfalls die Strategie zum Löschen von Blöcken eines vollen Cache hat darauf Einfluss.

Dass die Variation dieser Parameter Einfluss auf die Performanz des Programms hat ist intuitiv. Wie stark diese Auswirkungen allerdings tatsächlich sind, soll innerhalb dieses Kapitels mit Hilfe von Experimenten evaluiert werden. Um die ermittelten Werte besser einschätzen zu können, soll zusätzlich auf die Performanz der Berechnungen mit unkomprimiertem Graphen eingegangen werden. Hierfür wurden im Vorfeld zufällig 100 Paare von Knoten bestimmt zwischen denen später jeweils der kürzeste Pfad berechnet werden soll. Die Tests wurden auf einem Intel i3-2310M (2.1GHz) mit 8GB RAM ausgeführt. Als Externspeicher wurde einmal eine SSD (Samsung 840 Evo 256GB) als auch eine SDHC-Speicherkarte (Transcend Extreme-Speed Class 10 16GB) eingesetzt. Die Speicherkarte wird verwendet, um die Speicherressource von vielen mobilen Geräten nachzustellen. Diese erreicht wesentlich geringere Lesegeschwindigkeiten und besitzt keinen eingebauten Hardwarecache.

### Problem mit der bisherigen Implementierung

Die in Listing 3.4 vorgestellte Möglichkeit zum Cachen von komprimierten Blöcken bereitet beim Durchführen der Experimente einige Probleme: Obwohl sich die Klasse „MappedByteBuffer“ generell zum Einsatz innerhalb der Implementierung eignet, bereitet diese beim Ausführen der Experimente durch die schlechte Konfigurierbarkeit Schwierigkeiten. Erstens lässt sich nicht einstellen wieviele Blöcke maximal gespeichert oder wieviel Speicherplatz maximal belegt werden soll. Dadurch lässt sich diese Variable schlecht untersuchen. Zweitens werden die gespeicherten Blöcke vom Betriebssystem verwaltet, wofür auch nach längerer Recherche keine zuverlässige Methode gefunden werden konnte, diese bei Bedarf wieder zu löschen.

Um diese Schwachpunkte zu beheben, wurde die erste Cacheschicht, welche durch den „MappedByteBuffer“ gebildet wurde, durch eine Eigenimplementierung ersetzt, welche strukturell fast identisch zu Listing 3.5 ist.

## 4.1 Performanz mit unkomprimiertem Graph

Bevor näher auf die Kompression und das Caching eingegangen wird, soll dargestellt werden wie schnell sich eine Implementierung des Standard Dijkstra Algorithmus auf einem unkomprimierten Graphen verhält. Diese soll mit der optimierten Version, welche mit Contraction Hierarchies erweitert wurde, verglichen werden. Um den Performanzunterschied zwischen dem Dijkstra Algorithmus und der Erweiterung mit Contraction Hierarchies zu verdeutlichen, wurden in Tabelle 4.1 die Berechnungszeit sowie die Anzahl an Knoten, welche aus der Queue entnommen wurden, gegenübergestellt. Zusätzlich wurde unterschieden, ob der Graph bei der Berechnung vollständig im Arbeitsspeicher gehalten wurde oder lediglich auf einem Externspeicher (SSD) verblieb.

Nach kurzer Betrachtung der Werte in Tabelle 4.1 wird die massive Performanzsteigerung deutlich. Bei Dijkstra ohne Contraction Hierarchies sollte bedacht werden, dass dies Durchschnittswerte sind. Bei langen Pfaden steigt die Berechnungszeit schnell in Bereiche von ~20 Sekunden, sodass aus der Queue mehrere Millionen Knoten entnommen werden. Bei der Erweiterung mit Contraction Hierarchies ist dies nicht der Fall, sodass diese teilweise um Faktor ~1000 schneller ist. Ohne die Optimierung „Stall-On-Demand“ werden lediglich ~13700 Knoten aus der Queue entnommen. Wird diese Optimierung angewandt, kann diese Anzahl auf ~3500 gesenkt werden. Die Betrachtung der Versionen, welche von der SSD agieren, macht optimistisch, dass auch nach der Kompression schnelle Berechnungszeiten erreicht werden können.

| | **#Knoten aus der Queue entnommen** | **Berechnungszeit** | **Graph gepuffert in** |
|---|---|---|---|
| Dijkstra ohne CH | 139059 | 5.5s | RAM |
| Dijkstra mit CH | 13671 | 0.0179s | RAM |
| Dijkstra mit CH | 13671 | 0.073s | SSD (mit Memory-Mapping) |
| Dijkstra mit CH mit Stall-On-Demand | 3488 | 0.0078s | RAM |
| Dijkstra mit CH mit Stall-On-Demand | 3488 | 0.04s | SSD (mit Memory-Mapping) |
| **#Knoten im Graph** | 15172432 | | |
| **#Kanten im Graph** | 59473027 | | |

**Tabelle 4.1:** Vergleich von Dijkstra mit dessen optimierten Versionen in Bezug auf die Berechnungszeit und der Anzahl entnommener Knoten aus der Queue

## 4.2 Komprimierung

Nun soll der Einfluss der Variablen auf die Kompression evaluiert werden. Hierfür wurden drei unterschiedliche Kompressionsmodi aus dem Modul Deflater [def] gewählt, um die vorhandenen Daten zu komprimieren. Innerhalb dieser verschiedenen Verfahren konnte zusätzlich die Größe der Blöcke variiert werden, welche zu komprimieren sind. Die drei verwendeten Kompressionsmodi „BEST_COMPRESSION“, „HUFFMAN_ONLY“, „SPEED_COMPRESSION“ werden mit BEST, HUFFMAN und SPEED abgekürzt.

### Kompressionsmodi und Blöckgrößen

Zur Kompression wurde einmal der Modus BEST, welcher eine starke Kompressionsrate allerdings langsame Kompressions- und Dekompressionsgeschwindigkeit bietet, gewählt. Zusätzlich der SPEED-Modus, welcher ein etwas schwächeres Kompressionslevel erreicht, dafür aber schnelle Komprimierung und Dekomprimierung der Blöcke zulässt. Als letzten Modus HUFFMAN, welcher als drittes Verfahren keine Lempel-Ziv Kompression enthält.

Es wurden für jede Datei zehn verschiedene Blöckgrößen gewählt, wobei die Größen in einem Bereich gewählt wurden, in welchem der Einfluss der Größe auf die Kompression möglichst deutlich wird. In Abbildung 4.1 sind die Diagramme der erhobenen Daten dargestellt. Dabei wurden alle vorhandenen Graphdateien mit den eben besprochenen Variablen komprimiert. Zusätzlich sind zum Vergleich die Dateigrößen im unkomprimierten Zustand eingetragen („NO_COMPRESS“).

### Skipwerte

In Abbildung 4.1a ist die Kompression der Skipwerte eingetragen. Die Blöckgröße variiert hierbei von 500 bis 5000 Skipwerte pro Block. Wie erwartet komprimiert der Modus BEST am Stärksten, wobei der Unterschied zwischen SPEED und HUFFMAN nicht allzu groß ist. Zusätzlich kann abgelesen werden, dass sich die Größe der komprimierten Daten bei variierten Blockgrößen kaum verändert. Die Skipwerte konnten somit von ~475MB auf ~147MB mit Hilfe des BEST-Modus herabgesetzt werden.

### Knoten

Der Einfluss der Blockgröße kann bei den Knotendaten in Abbildung4.1b besser verdeutlicht werden. Diese variiert von 1000 bis 10000 Knoten pro Block. Zu Beginn der drei Kurven ist eine deutliche Verringerung des Speicherverbrauchs durch eine Vergrößerung der Blöcke erkennbar. Der Speicherverbrauch beginnt allerdings schnell im Bereich von ~158MB zu konvergieren. Der SPEED-Modus unterscheidet sich nur marginal von HUFFMAN, allerdings wird der Abstand zu BEST schon deutlicher. Der Speicherverbauch konnte von ~242MB im unkomprimierten Zustand auf ~158MB mit BEST gesenkt werden.

### Kanten

Die Kurven in Abbildung 4.1c verlaufen relativ flach, was allerdings an der Skalierung der y-Achse liegt, obwohl die Blockgrößen einen ähnlichen Effekt wie bei den Knotendaten haben. Die Blockgröße ist von 1000 bis 10000 Elemente pro Block gewählt. Der Abstand zwischen BEST zu SPEED sowie HUFFMAN verstärkt sich weiter und liegt bei ~50MB, da die Dateigröße deutlich größer ist. Der Speicherbrauch der Kanten konnte von ~951MB auf ~397MB mit BEST gesenkt werden.

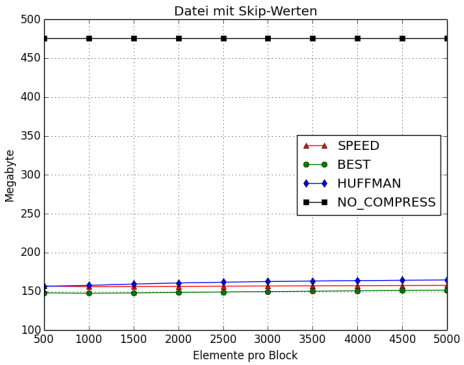
### Offsetwerte

Einen interessanten Fall bieten die beiden Dateien mit Offsetwerten in Abbildung 4.1d und Abbildung 4.1e, sowie die Datei mit den nach Zielknoten sortierten Kanten-IDs in Abbildung 4.1f. Die Blockgröße liegt zwischen 100 und 1000 Elementen pro Block. Hier erreicht nicht BEST, sondern SPEED die stärkste Kompression. Für manche Blöckgrößen ist BEST sogar schwächer als HUFFMAN. Daran wird deutlich, dass ein Kompressionsverfahren immer abhängig von den zu komprimierenden Daten ist. Im allgemeinen Fall liefert BEST die besten Kompressionsresultate. Erfüllen die Daten allerdings spezielle Eigenschaften, so kann zum Beispiel die Verwendung einer Huffman-Kodierung bessere Resultate erreichen. Eine Huffman-Kodierung funktioniert besonders gut, wenn eine Datei aus einem Universum von möglichen Werten nur eine geringe Anzahl dieser Werte beinhaltet. Dann können besonders kurze Codewörter generiert werden. Genau dies ist bei Offsetwerten der Fall. Jeder Offsetwert wird mit einem Integer (32 Bit) dargestellt, es gibt also $2^{32}$ mögliche Werte. Für jeden Knoten innerhalb des Graphen gibt es genau einen Offsetwert. Bei ~15 Millionen Knoten werden lediglich $\frac{15 \cdot 10^6}{2^{32}} = 0.0035\%$ der möglichen Werte benutzt. Die beiden Offset-Dateien können somit von jeweils ~60MB auf ~26MB mit Hilfe des SPEED-Modus gesenkt werden. Die Datei mit sortierten Kanten-IDs wird von ~237MB auf ~165MB mit SPEED verringert.

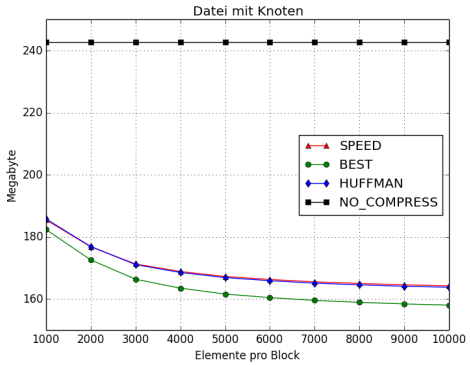
Durch die Wahl des jeweils besten Kompressionsmodus (im Bezug auf die Kompressionsstärke) für jede Datei ergibt sich folgende Gesamtgröße des komprimierten Graphen:

| Datei | unkomprimiert | komprimiert | Modus | Blockgröße |
|---|---|---|---|---|
| Kanten | 951MB | 397MB | BEST_COMPRESS | 10000 |
| Knoten | 242MB | 158MB | BEST_COMPRESS | 10000 |
| Skip | 475MB | 147MB | BEST_COMPRESS | 1000 |
| Offset (Source) | 60MB | 26MB | SPEED_COMPRESS | 1000 |
| Offset (Target) | 60MB | 26MB | SPEED_COMPRESS | 1000 |
| EdgesByTarget | 237MB | 165MB | SPEED_COMPRESS | 1000 |
| SUMME | **2025MB** | **919MB** | | |

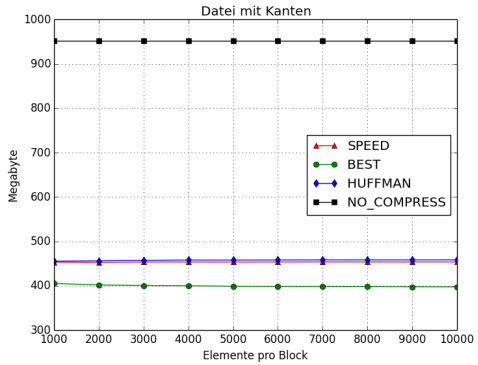
**Tabelle 4.2:** Dateigrößen in komprimiertem und unkomprimierten Zustand



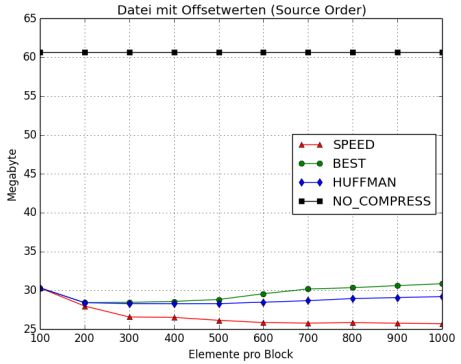

**(a)** Datei mit Skipwerten



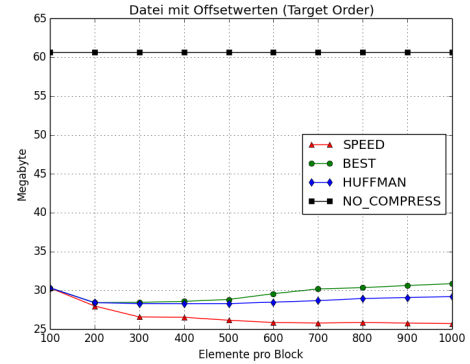

**(b)** Datei mit Knoten



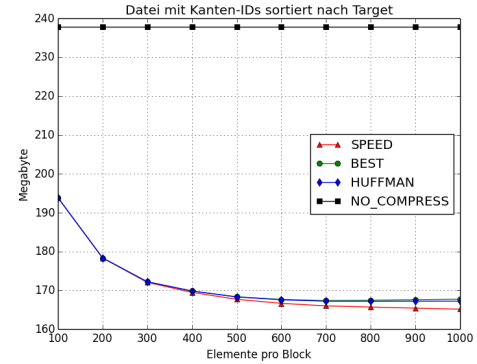

**(c)** Datei mit Kanten




**(d)** Datei mit Offsetwerten (Source)




**(e)** Datei mit Offsetwerten (Target)




**(f)** Datei mit geordneten Kanten-IDs

**Abbildung 4.1:** Auswertung unterschiedlicher Kompressionsmodis sowie Blöckgrößen auf die Dateigröße

## 4.3 Caching

In diesem Abschnitt sollen die Auswirkungen des Caching auf die Berechnungszeit ausgewertet werden. Durch die Pufferung von Teilen des Graphen im schnellen Arbeitsspeicher kann die Berechnungszeit massiv beschleunigt werden.

Im ersten Teil soll dabei für jeden Kompressionstyp gezeigt werden, wie sich die Berechnungszeit mit einer Vergrößerung der Cacheschichten verändert. Dabei wurde für jeden Kompressionstyp die Blockgröße pro Datei gewählt, welche die stärkste Kompression ermöglicht. Ein direkter Vergleich zwischen den Kompressionstypen ist hier noch nicht möglich, da die Blockgrößen unterschiedlich gewählt wurden und jede Cacheschicht durch eine maximale Anzahl an Blöcken und nicht Speicherkapazität limitiert ist.

Im zweiten Teil sollen die Kompressionstypen direkt in ihrer Berechnungszeit verglichen werden. Dies ist nur möglich, wenn jeweils die selbe Blockgröße als auch maximale Anzahl an Blöcken im Cache gewählt wird.

Im dritten Teil wird die Speicherressource von einer SSD auf eine SDHC-Speicherkarte gewechselt. Hierbei soll sich der Graph aus Dateien unterschiedlicher Kompressionstypen und Blockgrößen zusammensetzen. Diese wurden für jede Datei optimal gewählt, sodass der Graph möglichst wenig Speicherplatz verbraucht. Dies wurde bereits in Tabelle 4.2 gezeigt.

Im Folgenden soll die erste Cacheschicht, welche dekomprimierte Blöcke puffert, als „Stufe1“ bezeichnet werden. Die zweite Cacheschicht, welche komprimierte Blöcke speichert, wird „Stufe2“ genannt.

### 4.3.1 Berechnungszeit für unterschiedliche Kompressionstypen

Wie bereits erwähnt sollen zuerst alle drei Kompressionstypen mit der jeweils besten Blockgröße, in Bezug auf die Kompressionstärke, auf deren Berechnungszeit untersucht werden. Dabei werden folgende Fälle verwendet:

1. Stufe1 variiert, Stufe2 ist ausgeschalten, Replace Strategie = Random
2. Stufe1 variiert, Stufe2 ist ausgeschalten, Replace Strategie = LRU
3. Stufe1 ausgeschalten, Stufe2 variiert, Replace Strategie = Random
4. Stufe1 ausgeschalten, Stufe2 variiert, Replace Strategie = LRU
5. Stufe1 variiert, Stufe2 variiert, Replace Strategie = Random
6. Stufe1 variiert, Stufe2 variiert, Replace Strategie = LRU

Der jeweils variierende Cache soll zwischen 0 und 1000 gepufferten Blöcken ausgewertet werden. Die ersten vier Fälle zeigen jeweils die Berechnungszeit, wenn man die beiden Cacheschichten isoliert. Erst in den letzten beiden Fällen sind beide Cacheschichten aktiv. Dabei sollen zusätzlich die beiden folgenden Berechnungstypen unterschieden werden.

**„Kaltstart“:** Nach jeder Berechnung eines Pfads werden die Caches geleert. Somit werden die Berechnungszeiten bestimmt, die das Programm für die erste Berechnung nach dem Start benötigt. Dadurch beeinflussen sich die aufeinanderfolgenden Berechnungen nicht.

**„Warmstart“:** Die Caches werden nicht geleert. Dies stellt die „normale“ Benutzung des Programms dar. Dadurch beeinflussen sich die einzelnen Pfadberechnungen, da nach dem ersten Durchlauf einer Berechnung der Cache voll ist und somit Teile des Graphs im schnellen Arbeitsspeicher gespeichert bleiben.

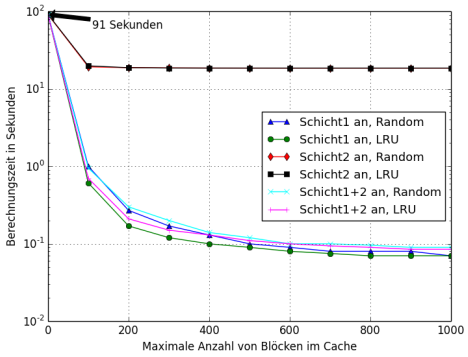
### Ergebnisse

In Abbildung 4.2 sind alle Ergebnisse in Diagrammen vorhanden. In Abbildung 4.2d-4.2f sind für jeden Kompressionsmodi die Berechnungszeiten im „Kaltstart“ bei unterschiedlichen Cachegrößen eingetragen. Es wird deutlich, dass bereits für kleine Cachegrößen die Laufzeit massiv verbessert werden kann. Bereits mit einer maximalen Größe von 100 Blöcken kann im BEST-Modus die Zeit von 91 auf unter eine Sekunde gesenkt werden. Dies kann im „Warmstart“ noch verbessert werden, beginnt aber bei allen Kompressionytypen spätestens bei einer maximalen Größe von 400 Blöcken keine signifikant besseren Ergebnisse zu liefern.

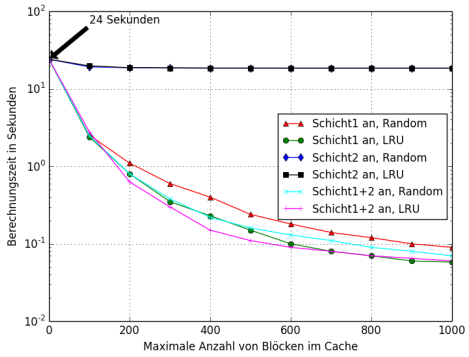
In den Abbildungen 4.2a-4.2c sind alle Kompressionstypen im „Warmstart“ aufgeführt. Im Vergleich zum „Kaltstart“ kann die Berechnungszeit durch die bereits gefüllten Caches um Faktor ~2 verbessert werden. Erkennbar flachen die Kurven langsamer ab, was darin resultiert, dass die maximale Cachegröße länger vergrößert werden kann, sodass sich die Berechnungszeit ebenfalls verbessert.

Unabhängig ob ein Cache nach jeder Berechnung geleert wird oder nicht können einige Punkte festgestellt werden: Sind beide Cacheschichten abgestellt, werden die unterschiedlichen Kompressionsgeschwindigkeiten der drei Modi deutlich. Eine Ausführung ohne Caching ist beim BEST-Modus um Faktor ~7 langsamer als im HUFFMAN-Modus und um Faktor ~4 langsamer als SPEED. Wie intuitiv vermutet wirkt sich eine verbesserte Kompressionsstärke im allgemeinen Fall negativ auf die Berechnungszeit aus.

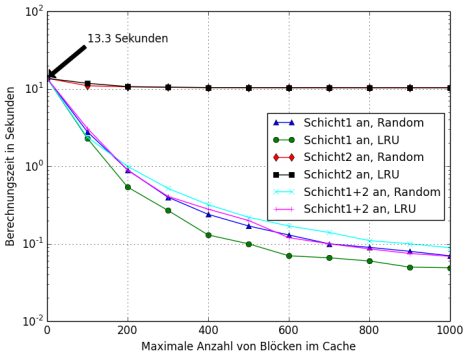
Ein weiterer wichtiger Punkt ist die Beurteilung des Verhaltens der unterschiedlichen Cacheschichten. Die zweite Cacheschicht bringt keinen signifikant verbesserten Performanzunterschied. Die Unterschiede zwischen der ersten und zweiten Cachestufe werden bei der Betrachtung der isolierten Ausführung beider Schichten deutlich. Die zweite Schicht bringt nur relativ kleine Verbesserungen in der Berechnungszeit. Obwohl hierbei Speicherzugriffe auf ein externes Gerät abgefangen werden können geht der Großteil der Zeit beim Dekomprimieren von Blöcken verloren. Deutlich besser funktioniert hier die erste Schicht. Diese verhindert ebenfalls Zugriffe auf den externen Speicher, macht es aber zusätzlich möglich Blöcke nicht mehrfach Dekomprimieren zu müssen. Teilweise ist die Ausführung mit beiden aktiven Schichten sogar langsamer als lediglich die erste Schicht aktiv zu haben (obwohl die Kapazität von zwei Schichten größer ist). Dies könnte vor allem in Verbindung mit dem LRU-Verfahren aus dem erhöhten Verwaltungsaufwand resultieren. Aus diesen Ergebnissen kann gefolgert werden, dass es sinnvoller ist die Pufferung von komprimierten Blöcken abzuschalten und diese Kapazität in die Speicherung von dekomprimierten Blöcken zu verlagern.



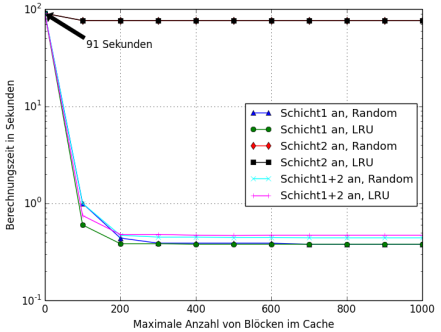

**(a)** BEST, „Warmstart“



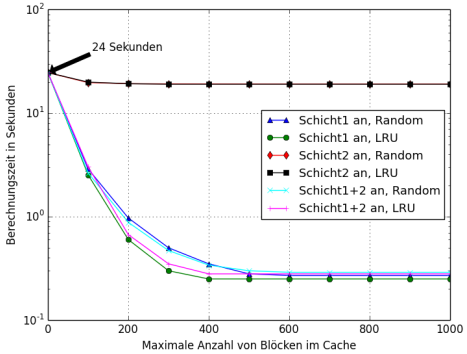

**(b)** SPEED, „Warmstart“



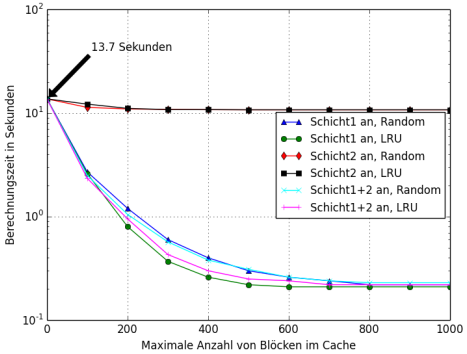

**(c)** HUFFMAN, „Warmstart“




**(d)** BEST, „Kaltstart“




**(e)** SPEED, „Kaltstart“




**(f)** HUFFMAN, „Kaltstart“

**Abbildung 4.2:** Auswertung von BEST, SPEED und HUFFMAN mit LRU- sowie Random Replacement jeweils mit „Kalt-“ sowie „Warmstart“

### 4.3.2 Auswirkung der Blöckgröße auf das Caching

Die Blockgröße, welche bei der Kompression des Graphen gewählt wurde, könnte Auswirkungen auf das Caching haben und soll deshalb in diesem Abschnitt untersucht werden. Damit dies möglich ist muss eine feste Cachekapazität in MByte gewählt werden, in diesem Fall ~37 MByte. Somit variiert die Anzahl der maximal gepufferten Blöcke, da diese je nach Blockgröße unterschiedlich groß sind. Je größer ein Block ist, desto weniger Blöcke werden gecached. Ist die Größe eines Blocks sehr klein, so können dafür sehr viele kleine Blöcke gespeichert werden.

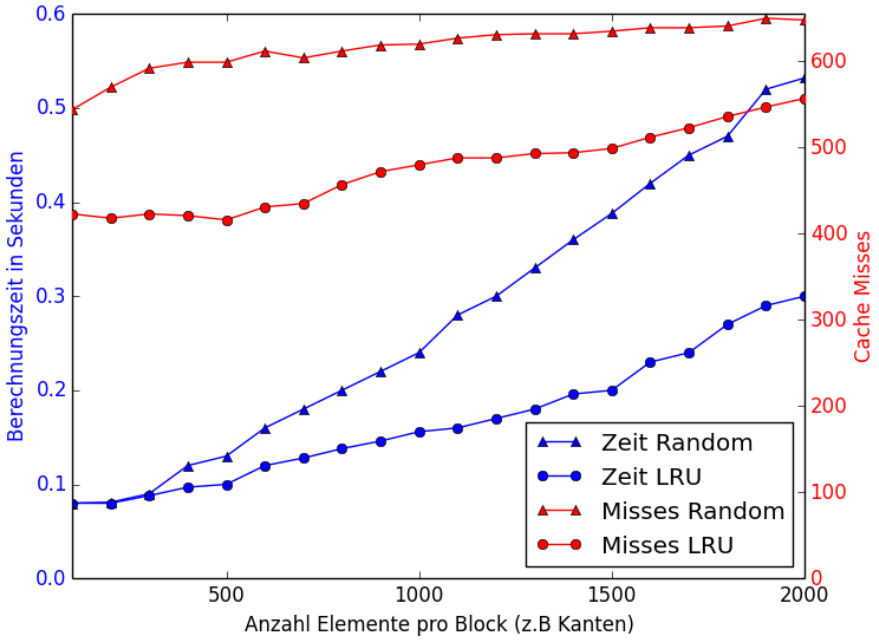



**Abbildung 4.3:** Einfluss der Blockgröße auf die Berechnungszeit und Anzahl der Cache Misses

#### Ergebnisse

Die Auswertung ist in Abbildung 4.3 dargestellt. Dabei variiert die Blockgröße für Offsetdateien zwischen 100 und 2000 Werten pro Block. Bei der Kantendatei wurde die Blockgröße um Faktor 10 größer gewählt, sodass diese von 1000 bis 20000 variiert. Man kann gut erkennen, dass mit steigender Anzahl von Elementen in einem Block die Berechnungszeit steigt und die Anzahl an Cache Misses leicht zunimmt. Dabei verläuft die Kurve der Berechnungszeit mit Random Replacement deutlich steiler als bei LRU. Für die Zunahme der Berechnungszeit mit wachsender Blockgröße gibt es zwei Erklärungsansätze: Einmal steigt die Dekompressionszeit für einen Block, da die Größe des Blocks zunimmt. Zweitens wird bei größer werdenden Blöcken immer mehr Redundanz, welche eventuell gar nicht gebraucht wird, in den Speicher geladen. Bis zu einem gewissen Grad wirkt sich das Laden größerer Blöcke positiv aus, um später weitere Zugriffe auf langsamen Speicher zu vermeiden. Übersteigen die Blöcke allerdings einen Schwellwert, werden Daten in den Arbeitsspeicher

| | | Cache Misses | |
|---|---|---|---|
| Blockgröße | #Anfragen | Random | LRU |
| 100 | 117829 | 544 | 423 |
| 300 | 100707 | 592 | 423 |
| 500 | 100792 | 599 | 416 |
| 700 | 101141 | 604 | 435 |
| 900 | 101490 | 619 | 472 |
| 1100 | 101660 | 627 | 488 |
| 1300 | 101571 | 632 | 493 |
| 1500 | 101508 | 635 | 499 |
| 1700 | 101595 | 639 | 523 |
| 1900 | 101565 | 650 | 547 |

**Tabelle 4.3:** Verhältnis zwischen Anfragen und Cache Misses

geladen die auch später nicht gebraucht werden und lediglich Platz für günstigere Blöcke belegen. In Tabelle 4.3 sind zusätzlich die Verhältnisse zwischen Anfragen an einen Cache und Misses für mehrere Blockgrößen eingetragen. Da sich das Verhältnis zwischen Anfragen und Misses nicht signifikant verschiebt, sollte dies keine großen Auswirkungen auf die Berechnungszeit haben. Es ist eher die Kombination aus leicht steigender Anzahl an Cache Misses und vergrößerter Zeit zur Dekompression von einzelnen Blöcke, das die Berechnungszeit steigen lässt.

### 4.3.3 Wechsel des Speichergeräts

Um eine realistischere Umgebung herzustellen soll in diesem Abschnitt die SSD durch eine einfache SDHC-Speicherkarte ersetzt werden. Hierbei soll die Zusammenstellung des Graphen aus Tabelle 4.2 verwendet werden, welche den wenigsten Speicherplatz durch die richtige Auswahl von Blockgröße und Kompressionsmodus verbraucht. Da innerhalb der Evaluierung deutlich wurde, dass die Cacheschicht zur Pufferung von komprimierten Blöcken die Performanz nicht positiv beeinflusst, soll diese hier abgeschalten bleiben. Diese Kapazität kann dafür in der ersten Schicht verwendet werden, um bereits dekomprimierte Blöcke zwischenzuspeichern.

| | | Speicherverbrauch für jeweilige Anzahl von Blöcken | | | | |
|---|---|---|---|---|---|---|
| Cache | Breite einer Zeile | 1 | 100 | 200 | 300 | 400 |
| Kanten | 16 Byte | 160kByte | 16MByte | 32MByte | 48MByte | 64MByte |
| Offsets | 4 Byte | 4kByte | 400kByte | 800kByte | 1.2MByte | 1.6MByte |
| EdgesByTarget | 4 Byte | 4kByte | 400kByte | 800kByte | 1.2MByte | 1.6MByte |

**Tabelle 4.4:** Speicherverbrauch für jeden Dateicache für unterschiedliche Anzahl von Blöcken

Zusätzlich soll auch der Verbrauch von Arbeitsspeicher bedacht werden, da dieser bei mobilen Geräten ebenfalls eingeschränkt ist. Zur Berechnung eines kürzesten Pfads werden lediglich die beiden Offsetdateien, die Datei mit den Kanten und die Datei mit den sortierten Kanten-IDs nach

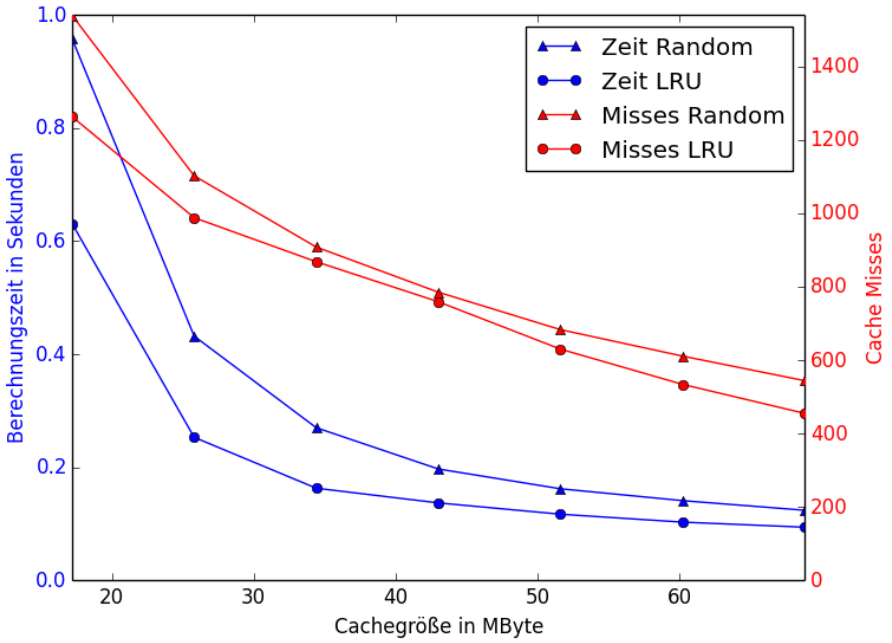



**Abbildung 4.4:** Berechnungszeit von einer SDHC

Zielknoten benötigt. Um den Speicherverbrauch der verwendeten Caches zu Bestimmen wird die Breite einer „Zeile“ der Dateien benötigt.

In Tabelle 4.4 kann abgelesen werden wieviel Arbeitsspeicher bei welcher Anzahl von Blöcken verbraucht wird. Werden für alle vier Dateien je 400 Blöcke als maximale Kapazität festgelegt, so wird $64 + 3 \cdot 1.6 = 68.8$MByte an Speicher verbraucht. Dies liegt für mobile Geräte absolut im Rahmen.

### Ergebnisse

Die Auswertung (im „Warmstart“) kann in Abbildung 4.4 betrachtet werden. Hierbei wurde einmal die Berechnungszeit berücksichtigt, als auch die Anzahl an Caches Misses für die jeweilige Größe des Cache. Die Anzahl an Anfragen an den Cache nach Blöcken lag insgesamt bei **101211**. Die beiden blauen Kurven stellen die Berechnungszeit, einmal mit Random- das andere mal mit LRU-Replacement, dar. Es zeigt sich, dass sich der erhöhte Verwaltungsaufwand bei LRU in einer schnelleren Berechnungszeit auszahlt. Beide Verfahren ermöglichen es schon mit sehr wenig Cachekapazität von <20MByte eine Berechnungszeit des kürzesten Pfads von unter einer Sekunde zu ermöglichen. Erhöhte man die Kapazität des Cache in einen Bereich von ~50MByte senkt sich die Zeit auf ~0.25 Sekunden und sinkt bis auf ~0.09 Sekunden mit Hilfe von LRU bei einem Verbrauch von ~70MByte.

Interessant sind ebenfalls die beiden roten Kurven, welche die Anzahl der Cache Misses für jedes Replacement Verfahren darstellen. Wie bereits erwähnt lag die Anzahl an Blockanfragen bei 101211. Vergleicht man nun die gesamte Anzahl an Blockanfragen mit der jeweiligen Anzahl an Cache

Misses, also die Situation wenn ein Block nicht im Arbeitsspeicher vorhanden ist, wird die Stärke der Cacheschicht deutlich. Bereits mit einer Cachegröße von <20MByte müssen von 101211 Anfragen weniger als 1500 Anfragen auf den externen Speicher propagiert werden. Die restlichen Anfragen können mit Hilfe von bereits vorhandenen Daten im Arbeitsspeicher bearbeitet werden. Dies kann weiter gesenkt werden, indem mehr Kapazität für den Cache bereitgestellt wird. So ist es mit Hilfe von LRU möglich die Anzahl an Misses auf gerade einmal ~450 zu senken.

## 4.4 Fazit der Evaluierung

Die, innerhalb der Evaluierung, erhobenen Daten ermöglichen folgende Aussagen zu treffen.

1. Cachschicht 2 kann abgeschalten und deren Kapazität in Schicht 1 verwendet werden.
2. Die gewählte Blockgröße hat Einfluss auf die Kompressionstärke.
3. Die Stärke von Kompressionsverfahren ist abhängig von den zu komprimierenden Daten.
4. Die gewählte Blockgröße bei der Kompression hat zusätzlich Einfluss auf das Caching und sollte nicht zu groß gewählt werden. Hier muss ein Kompromiss aus Berechnungszeit und Kompressionsstärke getroffen werden.
5. Schnelle Berechnungszeiten sind durch das Caching auch von Speicherkarten möglich.
6. Im Vergleich zur Berechnung mit unkomprimiertem Graph ist die Berechnungszeit lediglich um Faktor ~2-3 gefallen.

# 5 Zusammenfassung

In dieser Bachelorarbeit wurde eine Lösung erarbeitet, um kürzeste Wege auf komprimierten Graphen effizient zu berechnen. Dafür wurde der Standard Algorithmus zur Berechnung kürzester Pfade, der Dijkstra Algorithmus, mit Hilfe von Contraction Hierarchies erweitert, um die Laufzeit massiv zu beschleunigen. Diese Technik ermöglicht es, die Anzahl besuchter Knoten drastisch zu verringern. Zusätzlich wurde die Berechnung so angepasst, dass der Graph vollständig auf einem externen Speichergerät verbleibt ohne vollständig in den Arbeitsspeicher geladen werden zu müssen. Auf Grund des großen Speicherverbrauchs auf dem Externspeicher sollten die Graphdaten komprimiert werden. Um bei einer Berechnung eines Pfads nicht den gesamten Graphen dekomprimieren zu müssen, werden kleinere Blöcke des Graphen komprimiert und bei Bedarf dekomprimiert. Da Zugriffe auf einen Externspeicher langsam sind und das wiederholte Dekomprimieren von Blöcken möglichst vermieden werden soll, wurde eine mehrschichtige Cachelösung entwickelt. Innerhalb der Evaluierung wurde herausgefunden, inwiefern sich verschiedene Parameter wie Kompressionsverfahren oder Block- und Cachegröße manipulieren lassen, um Speicherverbrauch oder Berechnungszeit zu verbessern. Durch die ermittelten Ergebnisse wird geschlussfolgert, dass sich die Implementierung auch für mobile Geräte wie Smartphones eignet, um kürzeste Pfade in Zeitspannen zu berechnen, welche für einen Benutzer nicht zu langen Wartepausen führen.

## 5.1 Ausblick

Die aktuelle Implementierung bietet weiterhin Möglichkeiten zur Optimierung, welche aus Zeitgründen nicht vollständig umgesetzt werden konnten.

#### Einsparen von Speicherplatz

In Abschnitt 3.2.2 wurde eine verbesserte Variante des anfänglichen Graphformats eingeführt. Zu diesem Zeitpunkt wurde das Format geändert, da das Caching noch nicht vollständig entwickelt war und I/O-Operationen besonders langsam waren. Die Kanten enthalten dadurch stark redundante Daten, da in jeder Kante das Level des Start- und Zielknoten gespeichert ist. Für 60 Millionen Kanten sind dies insgesamt $60 \cdot 10^6 \cdot 4Byte = 240$ Megabyte. Würde man die Levelinformationen, wie ursprünglich, lediglich für jeden Knoten einmalig speichern, würden nur $15 \cdot 10^6 \cdot 2Byte = 30$ Megabyte benötigt. Dadurch ließen sich 210 Megabyte einsparen. Mit einer erhöhten Größe für den Cache, welcher die Datei mit Knoteninformationen verwaltet, könnten sich die zusätzlichen Leseoperationen kompensieren lassen.

### Verzahnen des Bidirektionalen Dijkstra

Anstatt beide Dijkstrainstanzen nacheinander von Start- und Zielknoten aus durchzuführen, kann diese Berechnung verzahnt werden. Beide Instanzen laufen hierbei parallel ab, wodurch die Suche schneller beendet werden kann. Dies verhilft, noch weniger Knoten besuchen zu müssen.

### Portierung auf eine mobile Plattform

Die bisherige Implementierung wurde nicht auf der tatsächlichen Zielplattform (mobile Geräte mit eingeschränkten Ressourcen) getestet. Es wurden allerdings Evaluierungen auf leistungsstärkerer Hardware durchgeführt, welche die Interpretation zulassen, dass der Einsatz auf mobilen Geräten möglich ist. Da die Implementierung in Java erfolgte, bietet es sich an, diese ohne allzu großen Aufwand auf Android zu portieren.

# Literaturverzeichnis


[BBK04] D. K. Blandford, G. E. Blelloch, I. A. Kash. An experimental analysis of a compact graph representation. 2004. (Zitiert auf Seite 9)

[BD09] R. Bauer, D. Delling. SHARC: Fast and robust unidirectional routing. *Journal of Experimental Algorithmics (JEA)*, 14:4, 2009. (Zitiert auf Seite 9)

[BDS+10] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, D. Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 15:2–3, 2010. (Zitiert auf Seite 9)

[Coa04] S. Coast. OpenStreetMap: Collaborative project to create a free editable map of the world, 2004. URL http://www.openstreetmap.org/. (Zitiert auf Seite 25)

[def] Java Deflater. URL http://docs.oracle.com/javase/7/docs/api/java/util/zip/Deflater.html. (Zitiert auf den Seiten 30 und 39)

[Deu96] P. Deutsch. DEFLATE Compressed Data Format Specification Version 1.3, 1996. (Zitiert auf Seite 23)

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. URL http://dx.doi.org/10.1007/BF01386390. 10.1007/BF01386390. (Zitiert auf den Seiten 11, 15, 16 und 25)

[GA95] J. Gailly, M. Adler. zlib: Software library for data compression, 1995. URL http://www.zlib.net/. (Zitiert auf Seite 23)

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, S. 319–333. Springer-Verlag, Berlin, Heidelberg, 2008. URL http://dl.acm.org/citation.cfm?id=1788888.1788912. (Zitiert auf den Seiten 11, 16 und 29)

[has] HashMap. URL http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html. (Zitiert auf Seite 33)

[Huf52] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952. (Zitiert auf Seite 22)

[KMS05] E. Köhler, R. H. Möhring, H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *Experimental and Efficient Algorithms*, S. 126–138. Springer, 2005. (Zitiert auf Seite 9)

[lin] LinkedHashMap. URL http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html. (Zitiert auf Seite 34)

[mbb] MappedByteBuffer. URL http://docs.oracle.com/javase/7/docs/api/java/nio/MappedByteBuffer.html. (Zitiert auf Seite 33)

[SSV08] P. Sanders, D. Schultes, C. Vetter. Mobile route planning. In *Algorithms-ESA 2008*, S. 732–743. Springer, 2008. (Zitiert auf Seite 10)

[ZL06] J. Ziv, A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, 2006. doi:10.1109/TIT.1977.1055714. URL http://dx.doi.org/10.1109/TIT.1977.1055714. (Zitiert auf Seite 19)

Alle URLs wurden zuletzt am 12. 05. 2014 geprüft.

**Erklärung**

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

---

Ort, Datum, Unterschrift