Bachelor Thesis

GTFS-Editor

Patrick Bühler

Gutachter: Prof. Dr. Hannah Bast
Betreuer: Patrick Brosi

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Institut für Informatik
Lehrstuhl für Algorithmen und Datenstrukturen

3. April 2017

Bearbeitungszeit

29. 05. 2018 – 29. 08. 2018

Gutachter

Prof. Dr. Hannah Bast

Betreuer

Patrick Brosi

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

Verkehrsbetriebe können die Daten ihrer Fahrpläne in Form von GTFS-Dateien (General Transit Feed Specification) im Internet bereitstellen. Diese sind in der Form von CSV-Textdateien in einer Zip-Datei gespeichert. Diese Dateien zu verändern kann ziemlich mühselig sein, da die Textdateien in den meisten Fällen sehr groß und unübersichtlich sind. Deswegen wäre es vorteilhaft, wenn man einen Editor hätte, mit dem man diese Feeds einfach verändern könnte. Dieser GTFS-Editor soll einfaches und benutzerfreundliches Editieren der Daten ermöglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

List of Algorithms

1 Einleitung

1.1 Motivation

Verkehrsbetriebe können ihre Fahrplandaten in der Form eines GTFS-Feeds im Internet veröffentlichen. Dieser Feed besteht aus einfachen CSV-Textdateien (Comma-Separate-Values). Wie der Name schon sagt, sind die Werte hintereinander geschrieben und mit Kommas abgetrennt. Bei größeren Dateien kann dies unübersichtlich werden. Diese Dateien können je nach Größe des Feeds mehrere Megabyte groß werden und aus mehreren tausend bis zehntausend Zeilen bestehen. Dadurch wird es bei großen Feeds deutlich unbequemer und zeitaufwendiger in diesen Feeds nach Fehlern zu suchen oder bestimmte Stellen zu aktualisieren.

```
shape_id,shape_pt_lat,shape_pt_lon,shape_pt_sequence,shape_dist_traveled
"10-10-I-j17-1.16.H","48.0350703651121","7.86368432524455","1","0"
"10-10-I-j17-1.16.H","48.0355942731014","7.86391418610644","2","60.7289058027559"
"10-10-I-j17-1.16.H","48.0357891760328","7.86361488695268","3","91.841604174964"
"10-10-I-j17-1.16.H","48.0359212403654","7.86241850648988","4","182.26837012399"
"10-10-I-j17-1.16.H","48.035809623121","7.86204548722671","5","212.731462547446"
"10-10-I-j17-1.16.H","48.0355530742359","7.86156838258358","6","258.338479551412"
"10-10-I-j17-1.16.H","48.0348309006098","7.86039083223668","7","377.342681157909"
"10-10-I-j17-1.16.H","48.0344585159883","7.86002357493472","8","426.991447507135"
"10-10-I-j17-1.16.H","48.0342868747857","7.85994690285889","9","446.916306352306"
"10-10-I-j17-1.16.H","48.0338558024676","7.86001004547305","10","495.082684667476"
"10-10-I-j17-1.16.H","48.0336571859799","7.85993396911687","11","517.886193169458"
"10-10-I-j17-1.16.H","48.0335582766702","7.85993615010156","12","528.931554186646"
"10-10-I-j17-1.16.H","48.0333973559012","7.86003356288248","13","548.244762102474"
"10-10-I-j17-1.16.H","48.0329225212662","7.86021835011737","14","602.815816701043"
"10-10-I-j17-1.16.H","48.0328326036916","7.86022033228541","15","612.865692322164"
"10-10-I-j17-1.16.H","48.0328326036916","7.86022033228541","16","612.865692322164"
"10-10-I-j17-1.16.H","48.032571842717","7.86022608051794","17","641.882928579257"
"10-10-I-j17-1.16.H","48.0319773230604","7.8601319151388","18","708.366009133636"
```

Deswegen wäre es gut, wenn man eine Art Editor hätte, mit dem man diese Feeds benutzerfreundlich verändern könnte, ohne in diesen Textdateien nach den jewei-

ligen Stellen zu suchen. Diese Bachelorarbeit beschäftigt sich mit der Erstellung eines solchen Editors, der einfache und intuitive Möglichkeiten bietet den Feed zu verändern.

1.2 Aufbau des Editors

Der Editor besteht aus zwei miteinander kommunizierenden Teilen, dem Backend und dem Frontend. Das Backend besteht aus einem Parser, der Textdateien des Feeds einliest, abspeichert und für die Benutzung bereitstellt. Das Frontend besteht aus einer Website, in der man auf einer Karte die Daten des Feeds editieren kann. Die Daten für das Frontend werden von dem Backend mithilfe eines Servers für das Frontend bereitgestellt.

Wenn im Frontend Elemente verändert werden, schickt das Frontend eine Anfrage zum Server, der anhand dieser Anfrage die Elemente im Parser verändert. Wenn man den Feed dann verändert hat, besteht die Möglichkeit, ihn über die Website zu downloaden.

2 Verwandte Arbeiten

In diesem Kapitel schauen wir uns Arbeiten an, die direkt oder indirekt mit dieser Arbeit zu tun haben, sowie Arbeiten, die sich mit ähnlichen Problemstellungen befassen.

2.1 Google Schedule Viewer

Der Google Schedule Viewer ist ein Tool von Google, das zum Untersuchen von GTFS-Feeds gemacht wurde. Es zeigt alle wichtigen Daten des Feeds wie Stationen, Kanten, Routen und Trips auf einer Karte an. Außerdem bietet das Tool auch die Möglichkeit nach bestimmten Stationen oder Trips zu suchen. Es ist ein Tool, das beim Auffinden von Fehlern helfen soll. Wenn man einen Fehler gefunden hat, muss man ihn jedoch trotzdem manuell in den Textdateien korrigieren, da ein editieren des Feeds mit diesem Tool nicht möglich ist. Ein weiterer Nachteil des Google Schedule Viewer ist die Ladezeit. Bei größeren Feeds kann es lange Dauern, bis der Viewer den Feed ausgelesen hat und ihn darstellen kann.

2.2 Conveyal Datatools-UI

Es hat eine Editierfunktion für GTFS-Feeds und ist das einzige Editor für GTFS-Feeds, den wir bei einer Suche im Internet finden konnten. Allerdings können wir zu

diesem Tool nicht viel sagen, außer dass es umständlich ist, es zum Laufen zu bringen. Für eine schnelle Bearbeitung von Daten kann es daher nicht benutzt werden, wenn es nicht bereits intalliert ist und für Laien dürfte es schwer bis unmöglich sein, es überhaupt zum Laufen zu bringen.

3 Benutzte Algorithmen und Datenstrukturen

In diesem Kapitel geht es darum, welche Algorithmen und Datenstrukturen bei der Implementation des Editors zum Einsatz gekommen sind.

3.1 GTFS

Das GTFS (General Transit Feed Specification) gehört im eigentlichen Sinn weder zu Algorithmen, noch zu Datenstrukturen, ist aber dennoch wichtig zum Verständnis und für die Implementierung des Editors. GTFS ist ein Format, in dem Verkehrsbetriebe ihre Fahrplandaten veröffentlichen können. Es besteht aus CSV-Textdateien, die in einer Zip-Datei gepackt sind. Die Textdateien werden eingeteilt in benötigte Dateien und optionale Dateien und diese Dateien haben benötigte Felder und optionale Felder. Für einen gültigen Feed braucht es alle benötigte Dateien und jede vorhandene Datei - auch optionale Dateien - braucht alle benötigten Felder. Optionale Dateien werden nicht zwingend benötigt um einen gültigen Feed zu erhalten, aber sie enthalten zusätzliche Informationen.

Die benötigten Textdateien sind *agency.txt*, *stops.txt*, *routes.txt*, *trips.txt*, *stop_times.txt* und *calendar.txt*. Die optionalen Textdateien sind *calendar_dates.txt*, *fare_attributes.txt*, *fare_rules.txt*, *shapes.txt*, *frequencies.txt*, *transfers.txt* und *feed_info.txt*.

Manche dieser Textdateien besitzen ein einmaliges Feld, das eine eindeutige Identifikation eines Elements zulässt. Manche Textdateien besitzen einen Fremdschlüssel, das auf ein Element einer anderen Textdatei verweist. Wie diese Relationen untereinander aussehen, kann man diesem Bild entnehmen: **(TODO: Bild einfügen)** Dadurch dass GTFS weiterhin weiterentwickelt wird, können zukünftig Änderungen in diesen hier beschriebenen Eigenschaften auftreten. Ebenfalls ist es möglich, dass der GTFS-Editor nicht mit allen Änderungen umgehen können wird.

3.2 R-Baum

Für den Editor ist es wichtig, auf Stationen und Kanten gezielt - je nach geografischer Position - zugreifen zu können. Das Durchsuchen aller Kanten und Stationen ist aufgrund der großen potenziell großen Anzahl an Stationen und Kanten ineffizient und würde die Laufzeit in diesem Fall stark erhöhen. Deshalb ist es nötig eine Datenstruktur zu verwenden, die einen gezielten Zugriff auf Stationen und Kanten in einem bestimmten geografischen Bereich ermöglicht. Dafür wird in dieser Arbeit der von Antonin Guttman entwickelte R-Baum benutzt. Der n-dimensionale R-Baum ermöglicht die Verarbeitung von n-dimensionalen räumlichen Daten.

Ein Knoten aus dem R-Baum besteht aus 2 Komponenten. Die erste Komponente ist die Bounding Box, die beschreibt, welchen Bereich der Knoten abdeckt. Sie besteht aus einem Intervall pro Dimension, das von dem Knoten abgedeckt wird. Die zweite Komponente ist die Information zu den Kindern, die in dem abgedeckten Bereich liegen. Kinder sind entweder weitere Knoten oder geografische Elemente (Stationen oder Kanten), die in der Bounding Box liegen. Wenn ein Knoten ein Blatt ist, hat er nur Verweise auf geografische Elemente, während ein Knoten, der kein Blatt ist, nur Verweise auf andere Knoten hat. Ein Knoten hat zwischen m und M Verweise, wobei $m \leq \frac{M}{2}$. Einzige Außnahme ist der Wurzelknoten, der weniger als m Elemente haben kann.

Ein R-Baum hat folgende Eigenschaften:

- Jedes Blatt hat immer zwischen m und M Verweise auf Elemente. Einzige Ausnahme kann der Wurzelknoten darstellen.

- Die Bounding Box eines Blattes ist eine minimale Bounding Box, die alle Elemente des jeweiligen Blattes umschließt. Das heißt, dass es keine kleinere Bounding Box gibt, die alle Elemente des Blattes vollständig umschließt.

- Jeder Knoten, der kein Blatt ist, hat zwischen m und M Kindknoten. Einzige Ausnahme kann der Wurzelknoten darstellen.

- Die Bounding Box eines Knoten, der kein Blatt ist, ist eine minimale Bounding Box, die alle Kindknoten vollständig umschließt. Das heißt, dass es keine kleinere Bounding Box gibt, die alle Kindknoten des Knotens vollständig umschließt.

- Der Wurzelknoten hat mindestens 2 Kinder, es sei denn der Wurzelknoten ist ein Blatt.

- Der R-Baum ist ein tiefenbalancierter Baum. Alle Blätter haben dieselbe Tiefe.

Weil jeder Knoten mindestens m Kinder oder Elemente hat (ausgenommen der Wurzel), ist die Tiefe eines Baumes mit n Elementen höchstens $\log_m n$. Da Knoten aber meist mehr als m Kinder haben, was die Tiefe des Baumes reduzieren kann.

3.2.1 Suche

Der Suchalgorithmus hat die Aufgabe, alle Elemente, die ein gegebenes Suchrechteck schneiden, zu finden und zurückzugeben. Der Suchalgorithmus beginnt an der Wurzel. Er betrachtet alle Kinder und wählt die Kinder aus, die den Suchbereich schneiden Auf diese Kinder wird dann der Algortihmus rekursiv angewendet und alle

zurückgegebenen Elemente werden zusammen zurückgegeben. Ist der Knoten ein Blatt, gibt er alle Elemente zurück, die den Suchbereich schneiden.

Algorithm 1 Search

```
function SEARCH(node, rectangle)
    overlappingElements = []
    if node is leaf then
        for element of node do
            if element overlaps rectangle then
                overlappingElements.add(element)
            end if
        end for
        return overlappingElements
    else
        for childNode of Node do
            SEARCH(childNode, rectangle)
        end for
    end if
end function
```

Die Laufzeit hängt davon ab, wie viele Knoten durchsucht werden müssen. Wie viele Knoten durchsucht werden müssen, hängt im Wesentlichen von 2 Faktoren ab.

Der erste Faktor ist die Größe des Suchrechtecks. Je größer das Rechteck und je mehr Knoten es schneidet, desto länger dauert die Suche. Der zweite Faktor ist, wie gut die Elemente beziehungsweise die Kinder beim Teilen eines Knotens auf dessen Kinder verteilt werden. Je mehr sich die Kinder überschneiden, desto mehr Knoten müssen unter Umständen durchsucht werden.

Im Optimalfall gibt wird immer nur ein Kindknoten pro Knoten ausgewählt. In diesem Fall ist die Laufzeit $O(\log n)$. Im Normalfall bei hinreichend gut verteilten Elementen und nicht zu großem Suchrechteck ist die Laufzeit ähnlich dem optimalen Fall, sodass man von einer Laufzeit von $O(\log n)$ ausgehen kann.

3.2.2 Element einfügen

Der Einfügealgorithmus fügt ein Element in den R-Baum ein.

Zuerst wird ein Blatt ausgewählt, in das das Element eingefügt wird. Falls das Element in dem Blatt noch Platz hat, wird das Element in das Blatt eingefügt. Ansonsten wird der Knoten in 2 neue Knoten geteilt. Diese neuen Knoten sind Kinder des Elternknotens des alten Knoten und haben die Elemente des alten Knotens, sowie das neu eingefügte Element. Wie der Teilalgorithmus aussieht, wird später beschrieben. **(TODO: Verweis auf Kapitel)** Als nächstes wird der Knoten beziehungsweise die 2 neuen Knoten angepasst. Die Veränderungen werden den Baum aufwärts bis zur Wurzel propagiert. Falls der Wurzelknoten dabei geteilt wird, erstelle einen neuen Wurzelknoten als Elternknoten der beiden neuen Knoten.

Algorithm 2 Insert

```
function INSERT(node)
    leaf = CHOOSELEAF(node)
    if leaf.elements < M then
        leaf.elements.add(node)
        ADJUSTTREE(node)
    else
        [L1, L2] = SPLITNODE(node) ADJUSTTREE(L1, L2)
    end if
    if root was split then
        make new rootnode
        old root-splitnodes become children of new root
    end if
end function
```

Die Laufzeit dieses Algorithmus hängt davon ab, wie die Laufzeit der Alogrithmen $chooseLeaf$, $splitNode$ und $adjustTree$ aussieht.

chooseLeaf

Ein Blatt zum Einfügen wird wie folgt gewählt. Man fängt bei der Wurzel an. Wenn man im Algorithmus bei einem Blatt ankommt, gibt man dieses Blatt zurück. Wenn der Knoten kein Blatt ist, wählt man den Kindknoten, dessen Rechteck am wenigsten vergrößert werden muss. Bei 2 Knoten, die gleichviel vergrößert werden müssten, wird der Knoten mit kleinerem Rechteck gewählt. Dieser Vorgang wird wiederholt, bis man an einem Blatt angekommen ist.

Algorithm 3 chooseLeaf

```
function CHOOSELEAF(node)
    n = root
    if n is leaf then
        return n
    end if
    while n is not leaf do
        enlarge= MAX
        ele = null
        for child of n do
            if childEnlarge(node) < enlarge then
                enlarge = childEnlarge(node)
                ele = child
            else if childEnlarge(node) == enlarge and child.Area < ele.Area then
                ele = child
                continue
            end if
        end for
        n = element
    end while
    return element
end function
```

Dieser Algorithmus benötigt pro Knoten $O(M)$ Zeit um alle Kinder zu überprüfen. Von der Wurzel bis zu den Blättern sind $O(\log n)$ Knoten. Somit ist die Laufzeit dieses Algorithmus $O(M \log n)$.

splitNode

Es gibt verschiedene Algorithmen um die Knoten zu splitten. Je nach Algorithmus verändert sich die Laufzeit sowie die Qualität der Aufteilung der Elemente. Diese Analyse beschränkt sich auf den quadratischen Algorithmus, der auch im Editor benutzt wird. Der erste Schritt ist das berechnen der SeedElemente, anhand denen die anderen Elemente den neuen Knoten zugeordnet werden. Als nächstes wird berechnet, welche beiden Elemente bei einem Einfügen in denselben Knoten die ineffizientesten Platznutzung hätten, also die größte ungenutzte Fläche in ihrer Bounding Box. Diese zwei Elemente ergeben jeweils ein Seedelement. Dann werden die übrigen Elemente dem neuen Knoten zugeteilt, bei dem die geringste Vergrößerung der Bounding Box nötig ist. Wenn ein Knoten so wenig Einträge hat, dass die restlichen Elemente diesem Knoten hinzugefügt werden müssen, damit dieser auf m Elemente kommt, füge die restlichen Elemente dem Knoten hinzu.

Algorithm 4 splitNode

```
function splitNode(node)
    rectangleSize = 0
    combination = null
    for combination of two distinct children c1, c2 of node do
        if rectangleSize < rectangleSize(c1, c2) then
            rectangeSize = rectangleSize(c1, c2)
            combination = c1, c2
        end if
    end for
    seeds = combination
    make new nodes n1, n2 from seeds
    for child of remaining children do
        if n1 or n2 need child for getting m children then
            Insert child in that node
        else
            Insert child in new node with less needed enlargement
        end if
    end for
    return n1, n2
end function
```

In diesem Algorithmus benötigt das Auswählen der Seeds eine Laufzeit $O(M^2)$, da M Elemente miteinander verglichen werden. Das Einfügen von den restlichen Kindern läuft in $O(M)$. Somit ist der Aglorithmus $O(M^2 + M) = O(M^2)$.

adjustTree

Bei diesem Algorithmus wird vom Blatt aus bis zum Wurzelknoten die Bounding Box der Knoten angepasst und falls nötig der Knoten geteilt. Man setzt eine Variable N auf den Knoten oder falls geteilt wurde auf die beiden neuen Knoten. Falls N nicht die Wurzel ist, korrigiere falls nötig die Bounding Box des Elternknotens. Falls zuvor geteilt wurde, schaue zusätzlich, ob der Elternknoten noch genug Platz für den zusätzlichen Knoten hat. Falls er nicht genug Platz hat, teile den Elternknoten. Weise N dem Elternknoten zu und wiederhole den Vorgang.

Algorithm 5 adjustTree

```
function ADJUSTTREE(leaf (, leaf2))
    N = leaf
    NN = leaf2 if split occured
    while true do
        if N is root then
            return
        end if
        adjustParentRectangle()
        if split occured then
            replace splitted node with N
            add NN to parent
            split parent in p1 and p2 if necessary
            if parent was split then
                N = p1
                NN = p2
                continue
            end if
        end if
        N = parent
        NN = null
    end while
end function
```

Die Laufzeit dieses Algorithmus hängt davon ab, wie oft Knoten geteilt werden. Wenn keine Knoten geteilt werden, muss für jeden Knoten von Blatt bis Wurzelknoten nur die Bounding Box angepasst werden. In diesem Fall beträgt die Laufzeit $O(\log n)$. Im schlimmsten Fall müssen jedoch alle Knoten gesplitted werden. Dann müssten $O(\log n)$ mal die splitNode Funktion aufgerufen werden. Das würde die Laufzeit auf $O(M^2 \log n)$ erhöhen.

Laufzeitanalyse Einfügen

Die Laufzeit des Einfügens hängt mit der Frage zusammen, ob und wie viele Knoten geteilt werden müssen. Wenn kein Knoten geteilt wird, hängt die Laufzeit nur von chooseLeaf() ab. In diesem Fall wäre die Laufzeit $O(M \log n)$. Wenn jedoch im schlimmsten Fall alle Knoten von Blatt bis Wurzel geteilt werden müssen, wäre die Laufzeit $O(M \log n + M^2 \log n)$, also $O(M^2 \log n)$ Wenn man eine Worst Case Analyse betreibt, kommt man also auf eine Laufzeit von $O(M^2 \log n)$. Demzufolge würde daas einfügen von n Elementen in einer Laufzeit von $O(M^2 n \log n)$ stattfinden.

Da jedoch nicht jede Einfügeoperation eine Teilung eines Knotens zur Folge hat, ist diese Einschätzung übertrieben. Für diesen Fall bietet sich eine amortisierte Laufzeitanalyse an. So verteilt man die Kosten einer teuren Einfügeoperation auf billige Einfügeoperationen um eine durchschnittliche Laufzeit einer Operation zu bestimmen. Die folgenden Abschnitt geht von einem Blattknoten aus, funktioniert jedoch anaolg bei einem inneren Knoten mit Kindern statt Elementen.

Wenn ein Knoten M Elemente besitzt und das $(M+1)$te Element eingefügt wird, spaltet er sich in 2 Knoten mit zusammen Platz für $2M$ Elemente. Von diesen $2M$ Elementen sind $(M+1)$ Elemente bereits belegt. $(M-1)$ Elemente können maximal eingefügt werden, bis ein weiteres Teilen unvermeidlich ist. Somit sind nach einer teuren Einfügeoperation $(M-1)$ billige Einfügeoperationen möglich. Also kann die

durchschnittliche Laufzeit von der teuren Operation und den blligen Operationen berechnet werden.

Die Laufzeit der teuren Einfügeoperation ist $O(M^2 \log n)$, also $\leq cM^2 \log n$ für eine Konstante c. Die Laufzeit der billigen Einfügeoperation ist $O(M \log n)$, also $\leq dM \log n$ für eine Konstante d. Der Durchschnitt von einer teuren und $(M-1)$ billigen Operationen beträgt also:

$$\frac{cM^2 \log n + (M-1)dM \log n}{M}$$

$$= \frac{cM^2 \log n + dM^2 \log n - dM \log n}{M}$$

$$= \frac{cdM^2 \log n - dM \log n}{M}$$

$$= cdM \log n - d \log n$$

$$= O(cdM \log n - d \log n)$$

$$= O(M \log n)$$

Diese durchschnittliche Laufzeit beträgt nun nicht mehr $O(M^2 \log n)$, sondern nur noch $O(M \log n)$.

3.2.3 Löschen

In diesem Abschnitt wird der Algorithmus zum Löschen eines Elements beschrieben.

Als erstes wird das Blatt gesucht, in dem sich das Element befindet. Dort wird das Element dann entfernt. Falls dieses Blatt dann weniger als m Elemente hat, wird es entfernt und dessen Einträge neu verteilt. Falls nötig wrd das entfernen der Knoten nach oben weitergeführt. Falls der Wurzelknoten dann nur noch ein Kind hat, wird dieses Kind die neue Wurzel.

Algorithm 6 delete

```
function DELETE(element)
    leaf = FINDLEAF(element)
    if leaf not found then
        return
    end if
    leaf.remove(element)
    CONDENSETREE(leaf)
    if root has 1 child then
        root.child = root
    end if
end function
```

findLeaf

Dieser Abschnitt beschäftigt sich mit dem Finden eines Blattes mit einem bestimmten Element. Man beginnt bei dem Wurzelknoten. Wenn dieser ein Blatt ist, wird kontrolliert, ob das gesuchte Element darin ist. Falls ja, wird dieser Knoten zurückgegeben und falls nicht wird null zurückgegeben. Wenn der untersuchte Knoten Knoten kein Blatt war, wird geschaut, welche Kindknoten mit dem Element überlappen. Für jedes dieser Knoten wird der Algorithmus mit dem jeweiligen Knoten statt dem Wurzelknoten erneut angewand.

Algorithm 7 findLeaf

```
function FINDLEAF(element, node)
    if node is leaf then
        if node has Element then
            return node
        else
            return null
        end if
    else
        for child of node do
            if child overlaps element then
                FINDLEAF(element, child)
            end if
        end for
    end if
end function
```

Da die Vorgehensweise bis auf das Suchrechteck gleich zu der Suche ist, ist auch die Laufzeit gleich zu der Suche, also im Normalfall $O(\log n)$.

CondenseTree

Mit diesem Algortihmus wird sichergestellt, dass alle Knoten mindestens m Elemente beziehungsweise Kinder haben. Falls ein Knoten diese Bedingung ncht erfüllt, wird er gelöscht und seine Elemente oder Kinder anderweitig eingefügt.

Algorithm 8 condenseTree

```
function CONDENSETREE(leaf)
    node = leaf
    eliminatedNodes = []
    while node not root do
        parent = node.parent
        if node less than m entries then
            parent.removeEntry(node)
            eliminatedNodes.add(node.entries)
        else
            node.adjustCoveringRectangle()
        end if
        node = parent
    end while
    for node in eliminatedNodes do
        INSERT(node)
    end for
end function
```

Die Laufzeit dieser Funktion hängt maßgeblich von der Laufzeit von Insert ab. Ohne Entfernen eines Knotens und damit ohne Insert wäre die Laufzeit $O(\log n)$, da der Baum von unten nach oben einmal durchgegangen wird. Wenn aber ein Knoten entfernt wird, müssen dessen Verweise erneut eingefügt werden. Es können maximal $\log(n) - 1$ Knoten entfernt werden. Jeder dieser Knoten hat $m - 1$ Verweise. Also würde Insert dann $(\log(n) - 1)(m - 1)$ mal aufgerufen werden. Mit der amortisierten Durchschnittslaufzeit wäre das dann eine Gesamtlaufzeit von $O(mM \log^2 n)$.

Analyse Laufzeit Delete

Die Laufzeit von Delete hängt zum großen Teil davon ab, wie viele Knoten gelöscht werden und damit wie viele Elemente neu eingefügt werden. Somit kann man folgern, dass die Laufzeit $O(mM \log^2 n)$ ist.

3.3 Douglas-Peucker

Der Douglas-Peucker Algorithmus dient dazu Kanten zu vereinfachen. Der Algorithmus betrachtet eine Liste von Punkten, die eine Kante darstellt. Als nächstes werden alle Punkte zwischen dem ersten Punkt P_1 und dem letzten Punkt P_n durchgegangen. Es wird der Punkt P_k herausgesucht, der den größten Abstand zur Strecke $\overline{P_1 P_n}$ hat. Falls dieser Abstand unter einem Grenzwert ist, werden alle Punkte bis auf P_1 und P_n entfernt. Falls er über dem Grenzwert liegt, wird Douglas Peucker rekursiv auf die Strecken $\overline{P_1 P_k}$ und $\overline{P_k P_n}$ angewandt.

Algorithm 9 douglasPeucker

```
function douglasPeucker(ListOfPoints, epsilon)
    line = LineListOfPoints[first], ListOfPoints[last]
    maxDist = -1
    mDInd = null
    for point in ListOfPoint except first and last Point do
        if Distance between point and line ≥ maxDist then
            maxDist = Distance between point and line
            mDPoint = point
        end if
    end for
    if maxDist ≥ epsilon then
        leftPoints = douglasPeucker(ListOfPoints[first...mDPoint], epsilon)
        rightPoints = douglasPeucker(ListOfPoints[mdPoint...last], epsilon) re-
turn [leftPoints, rightPoints without first element]
    elsereturn [ListOfPoints[first], ListOfPoints[last]]
    end if
end function
```

Dieser Algorithmus ist ein Divide-and-Conquer Algorithmus. Das Problem wird in 2 Subprobleme aufgeteilt und rekursiv bearbeitet. Die Kosten für den Algorithmus setzen sich zusammen aus den Kosten zur Findung des Maximum Distance Points und den Kosten für die 2 rekursiven Funktionsaufrufe, sowie den Kosten für das Zusammenführen. Wir gehen davon aus, dass das Zusammenführen konstant und unabhängig der Größe der beiden Teillisten ist. Außerdem ist die Laufzeit des Findens des Maximum Distance Points $O(n)$. Des weiteren gehen wir davon aus, dass die Funktion bei Eingabe einer Liste in konstanter Zeit durchgeführt wird. Dann hängt die Laufzeit davon ab, wie die beiden Listen geteilt werden.

Sei $T(n)$ die Anzahl der Rechenoperationen, die für den Algorithmus mit Liste der Größe n gebraucht wird. Wenn die Liste bei jedem Funktionsaufruf in der Mitte geteilt würde, sähe es wie folgt aus:

$$T(n) \leq 2T(\frac{n}{2}) + cn$$

$$T(n) \leq 2(2T(\frac{n}{4})\frac{n}{2})$$

$$T(n) \leq 4T(\frac{n}{4}) + 2cn$$

$$\vdots$$

$$T(n) \leq nT(1) + cn \log n$$

$$T(n) = O(n \log n)$$

Also wäre die Laufzeit $O(n \log n)$.

Wenn jedoch die Liste in eine Liste mit 2 Elementen und eine Liste mit $n - 1$ Elementen geteilt würde, sehe es wie folgt aus:

$$T(n) \leq T(1) + T(n-1) + cn$$

$$T(n) \leq 2T(1) + T(n-2) + cn + c(n-1)$$

$$T(n) \leq 3T(1) + T(n-3) + cn + c(n-1) + c(n-2)$$

$$\vdots$$

$$T(n) \leq nT(1) + c\sum_{k=2}^{k}$$

$$T(n) \leq nt(1) + c(\frac{n^2+n}{2} - 1))$$

$$T(n) = O(n^2)$$

4 GTFS-Editor

Dieses Kapitel beschäftigt sich mit dem eigentlichen Editor und wie er implementiert ist. Er ist in 2 Teile unterteilt, die miteinander kommunizieren, das Backend und die Website.

4.1 Backend

Das Backend ist in Java geschrieben. Das Backend lässt sich sich weiterhin in 2 Teile einteilen, den Parser und den Server.

4.1.1 Parser

Der Parser ist dafür zuständig, den Feed einzulesen, und in Form eines Graphs abzuspeichern und den gezielten Zugriff zu ermöglichen.

Zum Starten des Parser reicht das Ausführen des Makefiles. Zuvor müssen jedoch in dem Makefile die Source und der Port korrekt eingestellt werden. Als Source kann ein Verzeichnis mit den jeweiligen Textdateien dienen, sowie das Zipfile des Feeds.

Für jede Textdatei eines Feeds gibt eine eigene Art von Objekten. Anhand der ersten Linie der Textdatei wird bestimmt, welche Attribute vorhanden sind, welche Attribute nicht vorhanden sind, sowie die Reihenfolge in der die Attribute vorkommen.

Benötigte Attribute müssen zur korrekten Verarbeitung vorhanden sein. Optionale Attribute können fehlen.

Wenn die erste Linie verarbeitet wurde, werden die restlichen Zeilen anhand der Informationen der ersten Linie nacheinander eingelesen. Grundsätzlich wird eine Zeile in einem Objekt abgespeichert. Einzige Ausnahmen sind StopTimes.txt und Shape.txt. Bei den StopTimes werden alle StopTimes eines Trip in einem Objekt abgespeichert. Bei den Shapes werden alle Punkte einer Shape in einem Objekt abgespeichert. Das Einlesen der Textdateien erfolgt in einer Reihenfolge, die auf den Beziehungen der Objekte untereinander beruht. So wird zum Beispiel Routes.txt vor Trips.txt eigelesen, da ein Trip einer Route zugeordnet werden muss.

Um schnellen Zugriff auf die jeweiligen Feedobjekte zu gewährleisten, sind die meisten Feedobjekte im Parser in HashMaps gespeichert. Als Schlüssel zu einem Feedobjekt werden die Attribute des Feedobjekts genommen, unter denen man auf dieses Objekt zugreift. So werden zum Beispiel für StopTimes der dazugehörige Trip genommen. Für die meisten Feedobjekte werden einfach ihre IDs genommen. Die Feedobjekte, bei denen ein schneller Zugang nicht so wichtig ist, werden ArrayLists verwendet. Zu diesen Feedobjekten gehören zum Beispiel FareAttributes oder Frequencies.

Bei dem Einlesen der Stops aus Stops.txt wird zusätzlich noch ein neuer R-Baum erstellt und die Stops eingefügt. Auf die Implementierung des R-Baums gehen wir später ein.

Zusätzlich zu den Feedobjekten, die durch Auslesen aus den Textdateien erstellt wurden, gibt es noch den Objekttyp Edge. Diese Objekte stellen die Kanten zwischen 2 Stops dar. Zusätzlich speichern Edges welche Trips auf ihnen verkehren. Dafür werden bei der Erstellung einer Edge die 2 Stops abgespeichert. Außerdem werden die 2 Punkte der dazugehörigen Shape genommen, die am nächsten an den Stops sind, als Start- und Endpunkt der Edge genommen.

Für die Erstellung aller Edges werden alle Trips durchgegangen. Danach wird für diesen Trip die dazugehörige Shape herausgesucht. Als nächstes werden anhand der StopTimes des Trips geschaut, in welcher Reihenfolge die Stops angefahren werden. Falls dann zwischen 2 in diesem Trip aufeinanderfolgenden Stops noch keine Edge dieser Shape existiert, wird eine Edge erstellt. Falls jedoch schon ein anderer Trip eine Edge anhand dieser Shape zwischen diesen Stops erstellt hat, wird nicht noch einmal eine Edge erstellt, sondern der neue Trip in die bereits vorhandenen Edge eingetragen. Die Edges werden in einer verschachtelten Hashmap (shapeEdges) gespeichert. Die Schlüssel sind die dazugehörige Shape und die beiden Stops.

Nachdem auf diese Weise alle Edges erstellt wurden, werden die Edges noch aggregiert. Man geht davon aus, dass wenn 2 Fahrten zwischen den gleichen 2 Stationen stattfinden und das Verkehrsmittel das gleiche ist, dass diese Verkehrsmittel dann immer den selben Weg fahren. Somit kann mehrere Edges mit gleichem Verkehrsmittel (route_type) zwischen den gleichen Stationen zu einer Edge reduzieren. Es werden also alle Edges durchgegangen und in einer anderen verschachtelten HashMap (vehicleEdges) gespeichert. Die Schlüssel sind der route_type und die beiden Stops. Wenn also in vehicleEdges noch keine Edge mit den jeweiligen Schlüsseln vorhanden ist, wird die Edge aus shapeEdges übernommen. Ist jedoch bereits eine Edge mit den selben Schlüsseln in vehicleEdges vorhanden, werden die beiden Edges miteinander verglichen. Die Edge mit einer höheren Dichte an Punkten auf der Kante wird als genauer angesehen und wird in vehicleEdges eingefügt beziehungsweise bleibt in vehicleEdges. Von der anderen Edge werden die Trips in die Edge in der HashMap übertragen. So werden alle Edges von shapeEdges auf die vehicleEdges übertragen. Sobald alle Edges aggregiert wurden, werden die Edges ebenfalls in den R-Baum gespeichert.

Für den R-Baum gibt es 2 Klassen und ein Interface.

Das Interface ist RTreeElement, das von Stop, Edge und RTreeNode implementiert wird. Es stellt die nötigen Werte für die Bounding Box des jeweiligen Objekts sicher,

sodass das Objekt in den Baum eingefügt werden kann.

Die eine Klasse ist RTreeNode. RTreeNode stellt einen Knoten des R-Baums dar. Er hat eine minimale Bounding Box, die die in ihm abgespeicherten Elemente umgibt. In ihm werden die RTreeElements abgespeichert. Jeder RTreeNode hat einen Verweis zu seinem Elternknoten. Sollte der RTreeNode der Wurzelknoten sein, zeigt dieser Verweis auf sich selbst. Wenn es sich bei dem RTreeNode um ein Blatt handelt, sind die abgespeicherten RTreeElements ausschließlich Stops und Edges. Ist der RTreeNode kein Blatt, sind in ihm ausschließlich andere RTreeNodes gespeichert. Ein RTreeNode implementiert die Funktionalitäten zum finden aller Elemente in einem Suchrechteck und zum Finden des Blattes, das ein bestimmtes Element enthält. Außerdem enthält es Funktionen zum Teilen des jeweiligen Blattes.

Die zweite Klasse ist das Grundgerüst, der RTree. Sie ist der eigentliche R-Baum. Er speichert den Wurzelknoten. Zusätzlich ist in ihm M gespeichert. m ist in diesem R-Baum hart auf 2 gesetzt. RTree ist die Verbindung zwischen den Knoten und dem Parser. Die Funktionen Search, Insert, und Delete werden über RTree ausgeführt. RTree ruft die nötigen Funktionen des Wurzelknoten aus oder operiert auf den Knoten des R-Baums.

4.1.2 Server

Der Server dient zur Kommunikation mit der Website. Für die Verbindung mit der Webiste wird Javas ServerSocket und Socket benutzt. Ein Serverobjekt wird direkt bei in der Mainmethode erstellt. Bei der Erstellung eines Serverobjekts wird zuerst ein Parserobjekt erstellt und ihm das Inputfile übergeben. Nachdem der Server erstellt wurde, wird in einer Endlosschleife die getAndWorkRequest Methode des Servers aufgerufen. Die getAndWorkRequest Methode wartet auf eine Anfrage eines Clients und akzeptiert diese. Von dieser Verbindung zum Client bekommt der Server

dann eine Anfrage und gibt sie an die workRequest Methode weiter. Der Server beantwortet nur GET-Requests.

Es folgen if-else Überprüfungen, um herauszufinden, welcher Art die Anfrage an den Server ist. Bei der init-Anfrage schickt der Server Parameter für die Initialisierung der Website.

Bei der elements-Anfrage fragt der Client den Server nach allen Stops und Kanten in einem bestimmten Gebiet. In der Anfrage steht die Bounding Box, sowie das Zoom-Level der Karte der Website. Die Anfrage wird über den R-Baum abgewickelt. Der R-Baum gibt alle Kanten und Stops zurück, die in der Bounding Box liegen. Bei den Kanten wird zusätzlich noch Douglas-Peucker durchgeführt. Das Epsilon ist abhängig von dem Zoom-Level. Diese Elemente werden dann als JSON der Website übermittelt.

Bei der stop-Anfrage wird nach den Informationen zu dem Stop mit einer bestimmten ID gefragt. Dieser Stop wird dann über die HashMap der Stops gefunden und anschließend als JSON übermittelt.

Bei der trips-Anfrage wird nach allen Trips einer bestimmten Route einer bestimmten Kante gefragt. Die Kante wird über die HashMap vehicleEdges des Parsers gefunden und anschließend von der Kante die Trips der entsprechenden Route ausgelesen. Die Trips werden als JSON an die Website geschickt.

Es gibt 2 Arten von info-Anfragen. Die erste Art von Anfrage fragt nach allen Trips einer bestimmten Route bei einem bestimmten Stop. Die zweite Art an Info Anfrage gibt zu einem bestimmten Trip nähere Infos wie z.B. alle die Haltestellen und die Haltezeiten.

Die editStop-Anfrage modifiziert vorhandene Stops. Es gibt die Möglichkeit die Position zu verändern oder die sonstigen Attribute des Stops zu verändern. Falls die Position verändert wird, wird zuerst der Stop aus dem R-Baum genommen, dann

wird der Stop verändert und am Schluss wieder in den R-Baum eingefügt. Falls die sonstigen Attribut verändert werden, werden die Attribute verändert, die in der Anfrage stehen. Der Rest der Attribute wird nicht verändert. Die ID wird nur verändert, wenn die neue ID noch nicht vergeben ist. Ansonsten wird die ID nicht verändert und ein Hinweis in die Rückmeldung für die Website geschrieben. Bei dem Rest der Attribute wird keine Fehlerüberprüfung durchgeführt. Es wird angenommen, dass unzulässige Eingaben durch die Website verhindert werden.

Die editRoute-Anfrage modifiziert vorhandene Routen. Es ist möglich alle Attribute zu verändern. Es können auch mehrere Attribute mit einer Anfrage verändert werden. Die ID der Route wird nur verändert, wenn die neue ID noch nicht vergeben ist. Ansonsten wird die Route ID nicht verändert und ein Hinweis in die Rückmeldung für die Website geschrieben. Das verändern der Agency ID ist nur möglich, wenn es zu der neuen Agency ID auch eine korrespondierendes Agency Objekt gibt. Ansonsten wird die Agency ID nicht verändert und ein Hinweis in die Rückmeldung für die Website geschrieben. Bei den restlichen Attributen wird davon ausgegangen, dass die Website keine unzulässigen Eingaben zulässt.

Die editEdge-Anfrage verändert die Shape einer Edge. Die Anfrage enthält neue Punkte. Die Edge wird aus dem R-Baum entfernt. Die alten Punkte der Edge zwischen den 2 Stationen der Edge werden durch die neuen Punkte ersetzt. Dann wird die Edge mit den neue Punkten wieder in den R-Baum eingefügt.

Durch eine zip-Anfrage werden die momentan gespeicherten Elemente des Feeds wieder in GTFS-Format gespeichert. Die resultierenden Textdateien werden dann gezipt und als .zip der Website gesendet.

Bei einer routesOverview-Anfrage wird ein Überblick über alle Routen mit jeweiligen ID, shortName und longName an den Server geschickt. Bei einer route-Anfrage werden dann ausführliche Informationen zu einer bestimmten Route an den Server geschickt.

Der Server kann auch Dateien an den Browser schicken. Dies wird nur zum übermitteln der Website benötigt.

4.2 Website

Die Website besteht aus .html, .css und mehreren .js Dateien. Die eigentliche Funktionalität wird durch die Javascript Dateien erreicht. Die Javascript Dateien sind in 3 Teile aufgeteilt, dem Backend.js, dem GUI.js und dem site.js. Die site.js ist die Main Datei, die das Backend.js und das GUI.js initialisiert. Backend.js und GUI.js beinhalten die eigentlichen Funktionalitäten. Die GUI.js beinhaltet die Funktionalitäten der Benutzeroberfläche und alle Funktionen zum Interagieren mit der Karte. Die Backend.js beinhaltet die Funktionen zum Kommunizieren mit dem Server. Die Backend.js schickt GET-Requests ab und erhält die Antworten des Servers.

Die GUI besteht zum Großteil aus einer Karte. Für die Karte wird das Framework Leaflet benutzt. Die Tiles für die Karte kommen von Mapbox. Über der Karte gibt es eine Buttonleiste mit dem Button zum Aktivieren und Deaktivieren des Editiermodus und dem Button zum anzeigen der vorhandenen Routen.

Bei der Initialisierung wird zunächst bei Server nach der Initialposition gefragt. Das ist der Mittelwert aus allen Stationskoordinaten. Hinterher schickt die Website eine Anfrage für alle anzeigbaren Elemente, also den Stationen (Stops) und den Kanten (Edges). Diese werden dann abgespeichert. Für jede Station wird dann auf der Karte ein Marker gesetzt. Jede Kante wird durch eine Polyline angezeigt. Jedes Mal, wenn die Karte bewegt wird oder gezoomt wird, werden alle Elemente neu geladen. Popups werden in diesem Fall geschlossen.

Durch Klicken auf die jeweiligen Kartenelemente wird ein Popup mit näheren Informationen zum jeweiligen Element geöffnet. Klickt man auf einen Marker, werden alle Routen, die an dieser Station halten, angezeigt. Für jede Route gibt es einen

Button. Klickt man auf diesen Button, werden alle Haltezeiten dieser Route an der jeweiligen Station angezeigt. Für jede dieser Haltezeiten gibt es einen Button. Klickt man auf diesen Button, werden alle Informationen zu dem jeweiligen Trip, zu dem diese Haltezeit gehört, angezeigt. Für jede Haltestelle in diesem Trip gibt es einen Button. Klickt man auf diesen Button positioniert sich die Karte an der Position dieser Station und es öffnet sich das Popup des korrespondierenden Markers. Es gibt zudem einen Button um zu den vorherigen Informationen zurückzugehen.

Beim Klicken auf eine Polyline öffnet sich ein Popup, das die beiden Stationen anzeigt, die zu dieser Kante gehören. Zusätzlich werden noch die Trips, die zu dieser Kante gehören angezeigt. Es besteht die Möglichkeit entweder zu den beiden Stationen zu gehen oder Informationen zu den Trips anzeigen zu lassen. Lässt man sich die Trips anzeigen, bekommt man grundsätzliche Informationen zu jedem dieser Trips. Außerdem bekommt man die Möglichkeit sich die zu dem Trip dazugehörigen Stops anzeigen zu lassen. Per Buttonklick kann man dann zu den jeweiligen Stops gehen. Außerdem gibt es auch hier einen Button um zu den vorherigen Informationen zurückzugehen.

Wenn man in der Buttonleiste den Editiermodus aktiviert, gibt es verschiedene Möglichkeiten Daten zu verändern. Man kann die Marker per Drag and Drop verschieben. Hat man einen Marker verschoben, öffnet sich ein Popup, das fragt, ob man die neue Position speichern will. Man kann auf *speichern* drücken, um dem Server die neuen Informationen zu übermitteln. Man kann auf *cancel* drücken, um den Bearbeitungsvorgang zu beenden und den Marker auf der alten Position zu lassen. Das Schließen des Popups wird als *cancel* gewertet.

Wenn man im Editiermodus auf einen Marker drückt, sieht das Popup anders aus. Es werden alle Attribute außer den Koordinaten des Stops angezeigt und man hat die Möglichkeit jedes Attribut zu verändern. Man kann per Klick auf *save* die neuen Attribute speichern oder per Klick auf *cancel* die Bearbeitung abbrechen. Das Schließen des Popups wird ebenfalls als *cancel* gewertet.

Wenn man im Editiermodus auf deine Polyline drückt, so wird das Editieren der Line ermöglicht. Man kann Punkte der Polyline verschieben, sowie Punkte hinzufügen. Diese Funktionalität basiert auf der Leaflet-Erweiterung *Leaflet.Editable* Nach jedem Verschieben eines Punktes wird ein Popup geöffnet mit den Möglichkeiten *save* und *undo*. Bei einem Klick auf *save* werden die neuen Koordinaten der Edge an den Server geschickt. Bei einem Klick auf *cancel* werden alle Änderungen an der Kante verworfen und die gespeicherte Kante neu geladen. Wenn man das Popup schließt, bleiben die Änderungen an der Polyline vorhanden, werden aber noch nicht gespeichert. So kann man mehrere Koordinaten einer Kante mit einem Speichern verändern.

Der andere Button in der Buttonleiste oben öffnet eine *FancyBox* mit allen Routen. Bei jeder dieser Routen kann man sich per Button die Details anzeigen lassen. Wenn man sich die Details anzeigen lässt, ist es auch möglich, diese zu verändern. Man kann diese Veränderungen speichern oder die Bearbeitung abbrechen. Das Schließen des Fensters führt ebenfalls zum Abbruch des Bearbeitens. Der Button *show Trips* besitzt aus Zeitgründen noch keine Funktionalität.

In der Buttonleiste ist außerdem noch ein Hyperlink, durch den man den aktuellen GTFS-Feed downloaden kann. Falls Veränderungen gespeichert wurden, sind diese in dem downloadbaren Feed vorhanden.

5 Zukünftige Arbeit