Visualisierung sehr großer Graphen

Studienarbeit

am

Institut für Theoretische Informatik
Universität Karlsruhe (TH)

von

Timo Bingmann

Abgegeben am:

20. Juni 2006

Betreut durch:

Prof. Dr. Peter Sanders
Dominik Schultes

Zusammenfassung

In dieser Studienarbeit wird untersucht, mit welchen Methoden sehr große Graphen wie ein Straßennetzwerk von Europa effizient und komfortabel visualisiert werden können. Als Ausarbeitung entsteht ein C++ Rahmenwerk für die Datenhaltung eines Graphen mit Attributen und ein Java Applet, das mit dem Datenhaltungs-Server mittels CORBA kommuniziert. Das Rahmenwerk kann leicht in bestehende Graphanwendungen integriert werden, um deren Algorithmen zu animieren.

Als Basis-Graphstruktur wird ein Adjazenz-Array verwendet und um Strukturen erweitert, die zu jedem Knoten und jeder Kante beliebig viele Attributwerte speichern. Zwei der Knoten-Attribute werden als Zeichenkoordinaten verwendet. Der Grundgraph und die Datenhaltung der Attributwerte wird auf möglichst kompakte Art und Weise gelöst. Graphanwendungen können eine Liste von temporären Änderungen erzeugen, die mit dem großen globalen Graphen zusammengeführt werden können. Um das Vorgehen der Graph-Algorithmen zu visualisieren, werden deren Funktionsaufrufe in einer Änderungsfolge kodiert, welche als Animation zum Java Client übertragen wird.

Um die Geschwindigkeit einer Ausschnittsanfrage zu erhöhen, wird die mehrdimensionale Indexstruktur R-Tree verwendet. Diese ermöglicht Anfragezeiten, die linear zur Anzahl der zurückgelieferten Kanten und unabhängig vom gewählten Ausschnitt sind.

Es können komplexe Filterausdrücke aus Vergleichsbedingungen mit boolschen und arithmetische Operatoren verwendet werden, um die angezeigten Kanten in einem Visualisierungsauschnitt einzuschränken und so komfortabel bestimmte Aspekte der Anwendungs-Algorithmen zu untersuchen oder hervorzuheben.

Als Referenzanwendung wird das Rahmenwerk von der am Institut für Theoretische Informatik in Karlsruhe entwickelten Routenplanungsanwendung zur Visualisiserung mittels Web Applet verwendet.

Abstract

This study thesis investigates and implements methods used to efficiently visualize very large graphs like a street network of Europe. A C++ server framework is designed, which implements a data management library for graphs with attributes. A Java applet communicates with the data server via CORBA and draws sections of the graph. The graph data management library can easily be integrated into existing graph application to visual and animate calculations.

The data management library uses the adjacency array structure for representing the base graph. It is extended by similar data structures to hold an arbitrary number of attributes for each vertex and edge. The data structures are optimized towards highest storage efficiency. To modify the static global graph an application may construct a list of changes, which are efficiently merged into permanent data storage. To visualize the workings of an algorithm its function call sequence can be recorded into a list of graph changes. This change time line can be transfered to the Java client and displayed as an animation.

To accelerate access to arbitrary sections of the graph the spatial index structure R-Tree is used. It enables query times to linearly increase with the number of returned edges and be independent of the section size or position.

Furthermore complex filter expressions including comparisons and arithmetic operators can be applied to limit the displayed edges. This enables the client to explore the graph's details comfortably and highlight interesting aspects of an algorithm.

The graph data management library is used in a route planning application, which is being developed in a research group of the University of Karlsruhe. It will be used to visualize the route using a web applet.

Inhaltsverzeichnis

1 Einleitung

In der Theoretischen Informatik werden viele unterschiedliche Sachverhalte durch Graphen modelliert. Bei der Entwicklung und Präsentation von Graphalgorithmen ist es hilfreich, ein schnelles und flexibles Werkzeug zu haben, welches die untersuchten Ausschnitte zeichnet und bestimmte Aspekte hervorhebt. Graphen können mittels verschiedenen Layoutalgorithmen auf einem zweidimensonalen Koordinatensystem gesetzt werden oder haben bereits durch den repräsentierten Sachverhalt gegebene Knoten-Koordinaten. Zusätzliche Daten werden in einen Graph durch festlegen von Attributwerten auf Knoten und Kanten gespeichert.

Die effiziente Darstellung und Verwaltung von sehr großen Graphen ist eine interessante Herausforderung mit hohen praktischem Nutzwert. Die Speicherung von Attributwerten in Graphen der Größenordnung mehrerer Millionen Knoten und Kanten muss sorgfältig geplant und durchdacht sein. Darüber hinaus bedarf es einer geeigneten Indexstruktur, um verschiedene Ausschnitte des Graphen schnell auszuwählen und zu zeichnen. Im Rahmen dieser Studienarbeit entsteht ein Rahmenwerk, mit dem Graphalgorithmen bequem und schnell visualisiert werden können.

Als Referenzanwendung dient eine am Lehrstuhl für Theoretische Informatik an der Universität Karlsruhe entwickelte Routenplanung, welche auf einem Straßennetzwerk von Europa operiert. Das Straßennetzwerk besteht aus über 18 Millionen Knoten (Straßenkreuzungen) und 22 Millionen ungerichteten Kanten (Straßenabschnitten) und dient als Beispielgraph für Rechnungen.

Insgesamt jedoch ist das Graphrahmenwerk allgemein gehalten und lässt sich direkt auf andere Graphen mit festgelegten Knotenkoordinaten anwenden, wie zum Beispiel Web-Hyperlink-Graphen, die mit speziellen Algorithmen angeordnet wurden. Die entwickelten Module lassen sich leicht in bestehende Graphanwendungen integrieren, um dessen Ergebnisse und Berechnungen zu veranschaulichen.

Als Testanwendung entstand ein Java Programm sowie ein Web Applet, welches komfortable Navigation innerhalb des Straßennetzwerks von Europa erlaubt. Dabei werden Ausschnittsdaten von einem Datenhaltungs-Server mittels CORBA übertragen, der auf einem entfernten Rechner laufen kann. Die angezeigten Kanten können durch einen komplexen Filterausdruck mit boolschen Operatoren und Vergleichsbedingungen eingeschränkt werden. Durch Animation des Graphs kann der Rountenplanungsalgorithmus visualisiert werden. Es ist geplant, dass das Web Applet vom Institut für Präsentationen benutzt und über die Webseite des Instituts allgemein zugänglich gemacht wird.

1.1 Graphen mit Attributen

Das erarbeitete Rahmenwerk erlaubt es, beliebige gerichtete Graphen zu speichern. In dem Graphen kann jedem Knoten und jeder Kante beliebige Attributwerte zugewiesen werden. Von den Knotenattributen werden zwei als Koordinaten zum Zeichnen verwendet.

Der Graph hat für jeweils Knoten und Kanten eine Attribut-Definitionsliste, welche die möglichen Attribute durch einen eindeutigen String-Bezeichner und einen Attributtyp festlegt. Hierbei werden zum Beispiel die Typen `bool`, `integer`, `double` und `string` unterstützt. Weiter kann jede Attributdefinition einen Default-Wert enthalten, welcher verwendet wird, falls das Attribut nicht gesetzt ist.

1.2 Client/Server Trennung

Innerhalb des Graphrahmenwerks werden die beiden Aufgabenbereiche *Zeichnen des Graphen* und die *Datenhaltung* nach dem Client/Server Muster aufgeteilt. Der Datenhaltungs-Server ist immer mit der Anwendung als Bibliothek zusammen gelinkt und bietet dem Client einen Dienst, welcher die benötigten Ausschnitte des Graphen liefert. Die meiste Funktionalität wird im Server implementiert, daher spricht man von einem *Thin Client*.

Die Client/Server Komponenten können durch eine netzwerkfähige Middleware auf verschiedenen Rechnern liegen. Doch ebenfalls ist es möglich, Client und Server in eine ausführbare Binary zu linken.

Der Hintergrund dieser Trennung liegt in den großen Datenmengen, welche der Server zu verwalten hat. Diese kann mehrere Gigabyte betragen. Es wird jedoch immer nur ein kleiner Ausschnitt dieser Datenmenge zur Zeichnung benötigt.

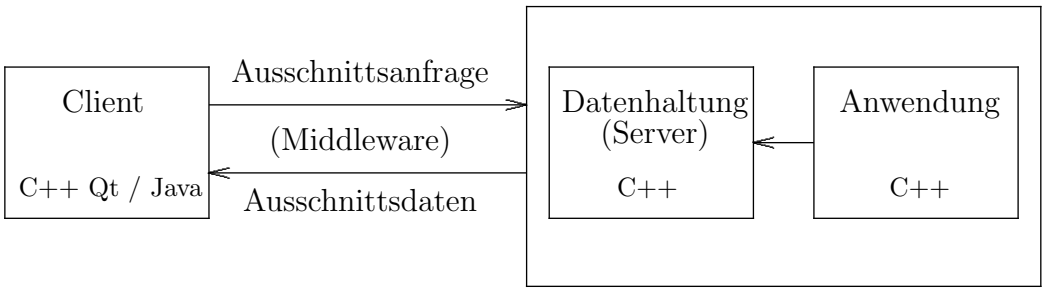


Abbildung 1: Client/Server/Applikation Architektur

1.3 Anwendungs-Integration

Der Datenhaltungs-Server integriert sich in eine bestehende Graphanwendung durch eine flexible C++ API.

Zu Programmstart wird ein Initialgraph mittels optimierten Ladefunktionen eingelesen. Falls benötigt, können danach von der Anwendung Knoten- und Kantenattribute geändert werden oder neue Knoten oder Kanten hinzugefügt werden. Diese Änderungen werden als lokal oder temporär betrachtet und erst durch eine spezielle Funktion permanent gemacht.

Lokale Änderungen werden vom Server protokolliert und in eine Folge von Time-Frames gespeichert. Diese Frame-Änderungsfolge wird dann zum Client übertragen und dort als Animation dargestellt. So können auf einfache Weise die Ergebnisse eines in der Anwendung implementierten Graph-Algorithmus dargestellt werden.

1.4 Darstellung

Der Datenhaltungs-Server stellt dem Client eine Methode bereit, welche alle Daten liefert, um einen vom Benutzer gewählten Graph-Ausschnitt zu zeichnen. Hierbei kann ein Filterausdruck angegeben werden, welcher aus verschiedenen Verknüpfungen der Knoten- und Kantenattribute und Konstanten besteht. Mittels dieses Filters können die gezeichneten Knoten und Kanten eingeschränkt werden, um gewisse Aspekte der Graphanwendung hervorzuheben.

Falls gewünscht kann der Server ein Kanten-Attribut als z-Ordnung verwenden. Dann werden die Knoten und Kanten innerhalb des vom Client festgelegten Ausschnitts ebenenweise betrachtet. Beim Zoomen kann so die Menge der zum Zeichnen gelieferten Knoten und Kanten begrenzt werden, wobei wichtigere Objekte vor unbedeutenderen betrachtet werden.

Die Animationsfolge wird zusammen mit dem Basis-Graph an den Client geliefert und auf Wunsch dem Benutzer vorgeführt.

Weiter kann der Client den angezeigten Graph-Ausschnitt im fig-Format [FigF] exportieren, welches wiederum in eine Vielzahl anderer Graphikformate konvertiert werden kann. Unter anderem können so PDF oder PNG Dateien zum Einbinden in Latex erzeugt werden.

1.5 Technologien

Die Graph-Datenhaltung ist als C++ Bibliothek implementiert, um eine hohe Verarbeitungsgeschwindigkeit zu erreichen. Die Boost.Spirit Parserlibrary [Spirit] wird verwendet, um Filter und Selektionsausdrücke zu parsen.

Es werden zwei Client-Implementationen angestrebt: einen Client in C++ mit der Qt Widgets-Library und einen cross-platform Java-Client. Beim C++/Qt Client wird besonderes Augenmerk auf Geschwindigkeit und Flexibilität gelegt. Hingegen ist der Java-Client auch als Web-Applet ausführbar und kann für ad-hoc Präsentationen verwendet werden.

Als Middleware wird CORBA verwendet, welches eine besonders einfache Übertragung der Daten zwischen C++ und Java ermöglicht.

2 Algorithmische Lösungsansätze und Design

Im folgenden Abschnitt werden verschiedene Lösungsansätze untersucht und verglichen. Dabei wird klar, welche Faktoren bei der Entscheidung für ein Verfahren ausschlaggebend sind. Aus den gewählten Lösungen ergibt sich ein verständliches Design.

2.1 Basis-Datenstruktur für den Graph

Die Darstellungsformen *Adjazenz-Matrix* und *Adjazenz-Listen* sind aus den Lehrbüchern für Algorithmen und Datenstrukturen z.B. aus [Cor01] gut bekannt. Zum Informatik-Grundwissen gehört auch, dass *Adjazenz-Matrizen* für dichte Graphen geeignet sind und direkten Index-Zugriff erlauben, aber immer $|V|^2$ Speicherzellen benötigen. Dahingegen sind *Adjazenz-Listen* für lichte Graphen besser geeignet, benötigen aber eine sequenzielle Suche auf der Kantenmenge eines Knoten.

Bei Auswahl der Datenstruktur für die Graph-Datenhaltung muss die zu erwartende Größe des Graphen das wichtigste Entscheidungskriterium sein. Der Beispiel-Graph von Europa hat ca. 18 Millionen Knoten und 22 Millionen Kanten; er gilt also als lichter Graph. Eine Darstellung als Adjazenz-Matrix ist völlig ausgeschlossen. Weiter muss berücksichtigt werden, dass der Server auf einem 64 Bit Rechner arbeitet, und dort Pointer acht Bytes groß sind.

Eine Überschlagsrechnung zeigt wie groß die Grundstruktur des Straßengraph ohne Attributwerte bei einer einfachen Implementierung als Adjazenz-Liste nach [Cor01] wird:

$$\underbrace{18 \cdot 10^6 \cdot 8 \text{ Bytes}}_{\text{Knoten-Array mit Listenpointer}} + \underbrace{22 \cdot 10^6 \cdot (4+8) \text{ Bytes}}_{\text{Kanten-Objekte mit Next-Pointer und ZielID}} = 408 \text{ Megabytes}$$

Dabei bleiben die Verwaltungsstrukturen von `malloc` und die Speicherfragmentierung durch die vielen Kanten-Objekte unberücksichtigt.

Statt der Adjazenz-Liste wird die weniger bekannte *Adjazenz-Array* Datenstruktur gewählt. Diese bietet nicht die Flexibilität der vorhergehenden Strukturen, erlaubt aber eine wesentlich kompaktere Speicherung. Die Abbildung 2 illustriert die Adjazenz-Array Datenstruktur. Es sind nur die Felder der Basis-Datenstruktur im Diagramm eingezeichnet.

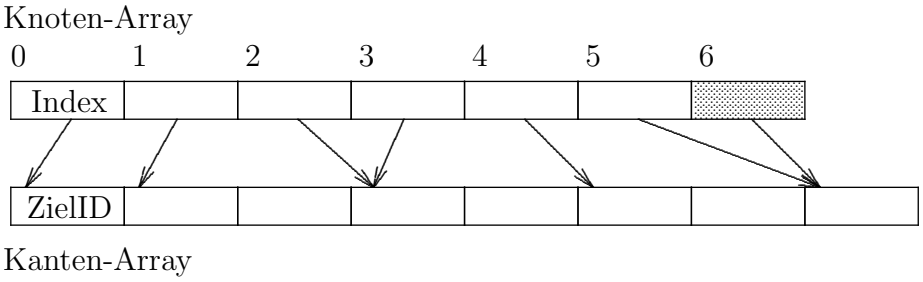


Abbildung 2: Adjazenz-Array

Es werde zwei Arrays verwendet, eines bestehend aus Knoten-Objekten und eines aus Kanten-Objekten. Wie in der Adjazenz-Listenstruktur wird die KnotenID als Index in das Knoten-Objektarray verwendet. Das Knotenobjekt enthält einen Index in das Kantenobjekt-Array. Dieser Index ist der Start der zu dem Knoten gehörigen Kantenobjekte. Alle darauf folgende Objekte gehören ebenfalls zu demselben Knoten, bis der Index des nächsten Knotens erreicht ist. In dem obigen Beispiel hat Knoten 0 also eine Kante, Knoten 1 zwei Kanten und Knoten 2 keine ausgehenden Kanten.

Damit die Kanten eines Knotens x immer in dem Bereich $\text{E}[\text{V}[x].\text{index} \ldots \text{V}[x+1].\text{index} - 1]$ liegen und kein Sonderfall beim letzten Knoten auftritt, wird einen Terminierungsknoten angefügt. Dieser Terminierungsknoten enthält keine gültigen Knotendaten und sein Index zeigt auf das letzte Kantenobjekt. So wird der Wert $\text{V}[x+1].\text{index}$ zu dem benötigten Endindex des letzten Knoten. Die Quell-KnotenID einer Kante braucht nicht nochmals im Kantenobjekt gespeichert sein. Hierdurch werden vier Bytes pro Kante gespart. Es kann jedoch dann nicht mehr von einer eigenen KantenID gesprochen werden, denn es gibt keine Möglichkeit, effizient die QuellID zu finden.

Es sei noch darauf hingewiesen, dass die Kanten-Indexnummern relativ zu dem Kanten-Array sind. Statt absoluten Speicher-Pointern kann ein kleinerer Datentyp wie `unsigned int` benutzt werden. Hierdurch wird nur die maximale Kantenanzahl auf 2^{32} limitiert.

Wiederholt man die obige Überschlagsrechnung für die Adjazenz-Liste, so zeigt sich eine deutliche Speichereinsparung.

$$\underbrace{18 \cdot 10^6 \cdot 4 \text{ Bytes}}_{\text{Knotenarray mit int Indizes}} + \underbrace{22 \cdot 10^6 \cdot 4 \text{ Bytes}}_{\text{Kantenarray mit int ZielID}} = 160 \text{ Megabytes}$$

Es kommen hier keine weiteren versteckten Kosten durch Fragmentierung oder `malloc` Verwaltungsstrukturen hinzu.

Für diese Speichereinsparung nehme ich in Kauf, dass nicht an beliebiger Stelle Kanten eingefügt oder gelöscht werden können, ohne im Array kopieren zu müssen. Demgegenüber erhält man als Bonus, dass man die zwei Arrays einfach als Ganzes in eine Datei schreiben und wieder laden kann, denn sie enthalten keine variablen Speicheraddressen.

2.2 Globale und lokale Graphdaten

Die oben beschriebenen Einschränkungen der Flexibilität durch Einsatz eines Adjazenz-Arrays werden wegen der großen Knoten/Kanten-Datenmengen in Kauf genommen. Der größere Teil des Graphen soll als *statisch* angesehen werden, also relativ wenige Änderungen erfahren. Darum wird die Kompaktheit der Speichermethode höher gewertet als dessen Flexibilität.

Um Daten im großen statischen Graphen zu verändern, muss dieser neu aufgebaut werden: das Adjazenz-Array muss neu geschrieben werden. Daher ist es nahe liegend, mehrere Änderungen im statischen Graphen zu gruppieren und alle auf einmal wirksam zu machen. Auf dieser Idee basiert die `Changelist`, welche eine Liste von *temporären* Änderungen in flexiblen Datenstrukturen hält (siehe den ChangeTimeline Klassenkomplex dazu). Diese Liste kann dann vorsortiert werden und mit den globalen Datenarrays zusammengemergt werden. Dies erfolgt dann ähnlich wie das Zusammenmischen bei Mergesort [Cor01].

Auf der anderen Seite kommt die Trennung von großen statischen Daten und kleineren temporären Änderungen der Animation der Berechnungsergebnisse von Graph-Anwendungsalgorithmen entgegen. Diese Ergebnisse sollen nicht permanent festgehalten werden, sondern nach der Visualisierung wieder zurückgesetzt werden. Weiter soll die Berechnung als fortschreitende *Änderungen* angezeigt werden. Daher liegt es nahe, diese getrennt von den permanenten Graphdaten zu halten, etwa in Form einer Änderungsliste.

Hier kann man den permanenten statischen Bereich als *globalen* Graphen betrachten, auf dem mehrere Instanzen einer Berechnung ausgeführten werden können. Diese tragen ihre Änderungen in eine *lokale* Überlagerung des globalen Graphen ein, welche dann entsprechend als Animation visualisiert werden können.

Die Trennung in globalen Bereich und lokale Änderungen legt nahe, den Server intern nach dem Master/Worker Muster zu organisieren, um mehrere Client parallel bedienen zu können.

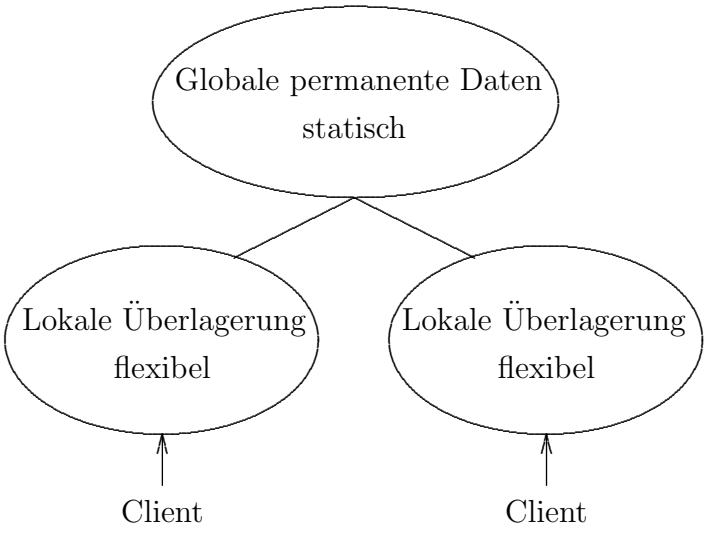


Abbildung 3: Global und lokale Graphdaten

Ein Worker-Thread arbeitet auf dem globalen Graph und erzeugt für den Client eine lokale Änderung. Die beiden Konzepte treffen sich hier, sind aber nicht gleich: lokale Änderungen können erzeugt werden, ohne dass Threads benutzt werden. Zum Beispiel können die Änderungstabellen auch verwendet werden, um schrittweise den globalen Graphen zu laden.

Die Abbildung 3 verdeutlicht die Struktur der getrennten Datenhaltung. Jede lokale Überlagerung entspricht einem Anwendungsfall der Applikations-Algorithmen. Diese werden von einem Worker-Thread erarbeitet und an den Client geliefert.

2.3 Indexstruktur

Bei der Visualisierung von Graphen der Größenordnung 10^7 Kanten ist es bei normaler Bildschirmauflösung selten gewünscht, alle zu zeichnen. Zwei Einschränkungen sind denkbar: nur „wichtige“ Kanten zeichnen und unwichtige weglassen oder nur die Kanten in einem bestimmten gezoomten Ausschnitt zeichnen. Dazwischen gibt es viele gewünschte Abstufungen und so scheint es angebracht, die Graphdaten in einer mehrdimensionalen Indexstruktur zu organisieren.

Die Indexstruktur muss ermöglichen, die Knoten und Kanten in einem gewählten Bereich mit absteigender „Wichtigkeit“ ebenenweise zu extrahieren. Dann kann aus einem Bereich gelesen werden, bis eine gewisse Anzahl Objekte gefunden wurde, und so kann eine gewünschte Objekt-Dichte (der Detailgrad) auf der Zeichenfläche erzielt werden.

Dabei muss entschieden werden, ob man die Knoten in Form von Punkten indiziert oder die Kanten als Liniensegmente. Indiziert man die Knoten, so findet eine normale Bereichssuche keine Kanten, welche den Bereich nur berühren. Bei einer solchen Kante sind beide Knoten außerhalb des Suchrechtecks, also nicht im Ergebnis des Suchindex. Indizierung über den Kanten ist wünschenswert, da in der Regel nur diese gezeichnet werden.

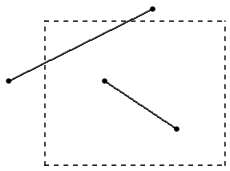


Abbildung 4:
Berührende Kante

Im Bereich der *mehrdimensionalen Zugriffsstrukturen* [Gae98] und der *„geospatial“ Datenbanken* [Rig01] findet man viele Möglichkeiten, den Zugriff auf den Graph zu beschleunigen. Von den vielen verfügbaren Indexstrukturen bieten sich nach erster Untersuchung an:

- Quadtree oder Octtree
- Grid-File
- R-Tree oder dessen Varianten R*-Tree und R^+-Tree

Im Folgenden wird kurz beschrieben, wie diese Indexstrukturen auf die gegebene Situation anwendbar sind und welche Zugriffs-Beschleunigung zu erwarten ist. Weiter soll untersucht werden, ob die Indexstruktur in zwei Dimensionen oder drei Dimensionen angewandt werden kann, um die Berechnung des Detailgrads zu unterstützen.

2.3.1 Quadtree

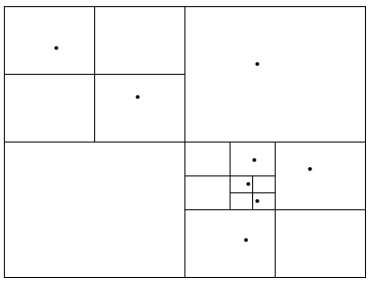


Abbildung 5: Quadtree

Als erste nahe liegende Datenstruktur unterteilt der Quadtree die Fläche rekursiv in vier gleichgroße Zellen. Sind mehr als k Objekte innerhalb einer rechteckigen Zelle, so wird diese wieder rekursiv geteilt. Man erhält einen Baum in dem jeder Knoten vier Kinder hat. In dem in Abbildung 5 gezeichneten Quadtree wird zur Veranschaulichung $k = 1$ gesetzt. In einer Implementierung würde man die Bucketgroße k hoch wählen.

Der Quadtree bietet sich offensichtlich für eine einfache rekursive Bereichssuche an. In [Sam90] findet man eine Analyse der *range query* im Quadtree. Dort wird ein worst-case mit $O(\sqrt{n})$ Zellenzugriffen konstruiert. Dieser worst-case liegt am Rande des Suchraums mit vielen leeren Unterteilungen. Der durchschnittliche Zugriff ist jedoch hinreichend schnell für die Graphvisualisierung. Weiter ist es möglich, das Quadtree-Unterteilungskonzept auf drei Dimensionen zu erweitern: man erhält den Octtree, also eine achtfache Unterteilung des Raums.

Um den Quadtree der Graph-Koordinaten effizienter zu machen, muss statt der äquidistanten Unterteilung eine andere gewählt werden, bei dem der Unterteilungsmittelpunkt möglichst gut die Punkte in vier Flächen teilt. Diesen (doppelt-)balancierten Quadtree herzustellen, ist jedoch wesentlich komplexer und erfordert relativ hohen Aufwand. Ein weiterer Nachteil des Quadtrees ist die hohe Speicherfragmentierung, welche wie bei normalen Binärbäumen durch die vielen kleinen Knotenobjekte entsteht.

Will man für einen Graphen im Quadtree die Kanten als Liniensegmente indizieren, muss in jedem der von einer Linie berührten Quadtree-Unterteilungen eine Referenz auf die Kante gespeichert werden. Bei einer feinen Unterteilung des Quadtrees werden das sehr viele Einträge pro Linie.

Für Indizierung über den Knoten als Punktmenge ist der Quadtree eine geeignete Indexstruktur.

2.3.2 Grid-File

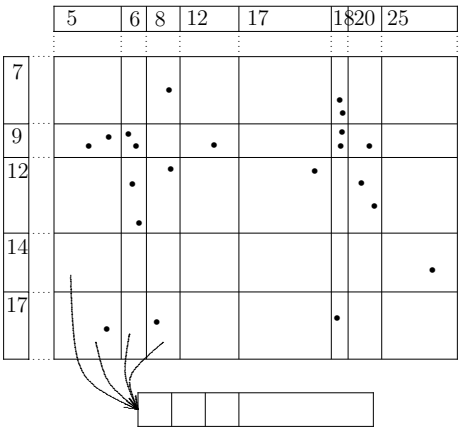


Abbildung 6: Grid-File

Einen ganz anderen Ansatz verfolgt das Grid-File [Niev84]. Als Ausgangspunkt sieht man die zu indizierende Ebene als riesige Matrix, so dass man auf den gesuchten Bereich direkt durch Zeilen und Spaltenindizes zugreifen kann. Diese Matrix ist viel zu groß, also „komprimiert“ man sie an den Spalten und Zeilen-Achsen. Hierbei werden die Bereiche so zusammengefasst, dass in den Feldern bis zu k Objekte enthalten sind. Diese Zellen bilden dann ein adaptives Gittermuster auf der Ebene.

Das Gittermuster wird durch zwei Arrays beschrieben, welche eine aufsteigende Folge der Unterteilungsstellen enthalten. Hierbei wird oft von einem *linearen Hash* auf den Achsen gesprochen. Der Gitterinhalt wird als Zugriffsmatrix (dem eigentlichen *Grid-File*) im Hauptspeicher gehalten, wobei jede Zelle aus einem Zeiger auf eine Bucket-Seite besteht. Hierbei kann durchaus ein Bucket auch für mehreren Zellen stehen.

Um die Punkte in einem Bereich zu finden, stellt man durch zwei binäre Suchen in den Achsenarrays fest, in welchen Zellen der Bereich liegt. Man erhält zwei Intervalle von Indizes in das Gitter. Dann werden die Buckets durchsucht, welche zu den ausgewählten Gitterzellen gehören. Für jeden Eintrag im Bucket muss wieder eine Bereichsprüfung stattfinden.

Das Grid-File erlaubt es, jeden Punkt in zwei Zugriffen zu erreichen: einmal im Grid nach schauen, dann das Bucket laden. Demgegenüber muss im Quadtree jedesmal der Baum von der Wurzel zu einem Blatt abgelaufen werden. Um alle Punkte eines Bereichs im Grid-File zu finden, müssen entsprechend mehr Zellen betrachtet werden.

Durch das Gittermuster werden gleichförmig verteile Punktmengen gut verwaltet. Bei nicht gleichförmig verteilten Daten ergeben sich jedoch sehr viele leere Zellen im Grid-File. Hierdurch wächst die Gittermatrix-Größe schneller als linear bezüglich der Punktanzahl. Dieses Wachstum ist für den Beispielgraphen mit 18 Millionen Punkten nicht mehr akzeptabel. Hierbei würden sich durch die Gitteraufteilung sehr feine Zellen im Bereich der Großstädte bilden; diese feine Aufteilung setzt sich dann jedoch auf der ganzen Karte fort.

2.3.3 R-Tree

Die bisher besprochenen Indexstrukturen basieren auf einer Dekomposition des Raums oder der Ebene in kleinere Teile, in denen dann die Objekte verzeichnet sind. Der R-Tree [Gut84] funktioniert umgekehrt: die Objekte werden gruppiert und Gruppen von Objekten dann wieder rekursiv zusammengefasst. Beim R-Tree werden immer d-dimensionale Rechtecke betrachtet und gruppiert. Um andere Objekte zu indizieren, wird deren Minimum-Bounding-Box (MBB) verwendet. Von gruppierten Rechtecken wird rekursiv wieder die Minimum-Bounding-Box der Gruppe auf der nächsthöheren Tiefe des Baums eingetragen.

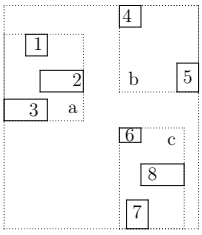


Abbildung 7: Gruppierte Rechtecke

Der R-Tree ist eine mehrdimensionale Weiterentwicklung des B-Baums. Wie beim B-Baum, werden (Hintergrund-)Speicherseiten als Knoten und Blätter mit M Einträgen verwendet. Beim R-Tree sind die Einträge jedoch d-dimensionale Rechtecke statt einfacher Zahlen. Jeder Eintrag in einem Knoten hat einen Verweis auf die in dem Rechteck enthaltene Unterrechtecke. In einem Blatt sind stattdessen die indizierte Daten enthalten. Wie beim B-Baum wird eine Mindest-Füllzahl $m \leq \frac{M}{2}$ festgelegt. Ein R-Tree genügt folgenden Eigenschaften:

1. Jeder Knoten und jedes Blatt enthalten zwischen m und M Einträge, es sei denn es ist die Wurzel.
2. Die Wurzel hat mindestens zwei Kinder oder sie ist ein Blatt.
3. Jeder Rechteckeintrag in einem inneren Knoten ist die MBB der Rechtecke in den Kinderknoten.
4. Alle Blätter liegen in der gleichen Tiefe.

Der R-Tree wird in den bekannten Datenbanksystemen Oracle, PostgreSQL und MySQL benutzt, um den Zugriff auf räumliche Objekte wie Punkte, Linien oder Polygone zu beschleunigen.

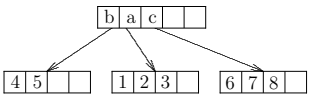


Abbildung 8: R-Tree Knoten

Da der R-Tree mit Rechtecken arbeitet, kann man auf natürliche Weise die Kanten eines Graphen indizieren, ohne Mehrfach-Verweise durch die Raumaufteilung. Jede Kante wird als genau ein Rechteck gespeichert. Dies erleichtert erheblich den Suchaufwand, denn es muss keine Duplikatliste für die gefundenen Kanten geführt werden.

Durch die kompakte Speicherung der Rechteckeinträge als Gruppen in Speicherseiten entsteht wenig Speicherfragmentierung. Für sehr große Graphen können die Parameter M und m groß gewählt werden, um einen kompakten Baum zu erhalten.

Aus diesen beiden Gründen wurde in dieser Arbeit der R(*)-Baum als Indexstruktur für die Graph-Datenhaltung ausgewählt.

Die Bereichssuche im R-Tree kann durch rekursives Absteigen in mit dem Suchrechteck überlappende Rechtecke implementiert werden. Eine einfache worst-case Analyse kann nicht durchgeführt werden, denn die Suchgeschwindigkeit hängt vor allem von der „Güte“ des Baumes ab. In der Abbildung 9 sind nochmals die Rechtecke aus der Abbildung 7 mit einer anderen Gruppierung dargestellt, bei der eine Suche mehr Rechteckgruppen betrachten muss.

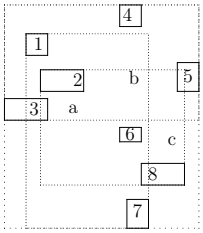


Abbildung 9: Andere Gruppierung

Durch Optimierung der „Güte“ des Baumes während dessen Aufbau erhält man kürzere Zeiten bei einer anschließenden Bereichssuche. Die interessanten algorithmischen Aufgaben des R-Tree bestehen darin, bei beliebigen Einfüge- und Lösch-Operationen möglichst *gute* Aufteilungen oder Verschmelzungen der Rechteckgruppen zu finden.

Hierzu werden im Folgenden vier Variationen des R-Trees untersucht:

1. (original) R-Tree mit quadratischem Split [Gut84]
2. (original) R-Tree mit linearem Split [Gut84]
3. R*-Tree [Beck90]
4. R^+-Tree [Sel87]

Diese Variationen unterscheiden sich nur durch die Algorithmen beim Einfügen und Löschen. Sie unterscheiden sich nicht in der Struktur des Rechteck-Baums. Die ersten drei Variationen sind in der Datenhaltungs-Bibliothek implementiert, die vierte hat für die Graphvisualisierung erhebliche Nachteile und wurde deswegen nicht umgesetzt.

Im Folgenden wird kurz beschreiben, wodurch sich die Variationen beim Einfügen neuer Rechtecke unterscheiden. Auf eine detaillierte Beschreibung des Insert-Vorgangs wird verzichtet und stattdessen die entscheidenden Algorithmen genauer erläutert. Es gibt beim Insert zwei verschiedene Aufgaben zu lösen:

`chooseSubtree`: Aus einer Rechteckgruppe soll eines ausgewählt werden, in dem ein neues Subrechteck eingetragen wird. Dies wird als erste Funktion von Insert verwendet, um das Blatt auszuwählen, welches die neue Kante enthalten wird.

`splitNode`: Zu einer vollen Gruppe mit M Elementen soll ein weiteres hinzugefügt werden. Dies erfordert, dass die Gruppe aufgeteilt wird, und die neue Rechteckgruppe in derselben Ebene eingefügt wird.

R-Tree mit quadratischem Split

Im original R-Tree [Gut84] wird für `chooseSubtree` das Rechteck ausgewählt, dessen *Fläche* sich durch Hinzufügen des neuen Subrechtecks am wenigsten vergrößert. Hierzu ist ein Durchgang durch alle Rechtecke einer Gruppe notwendig, kann also in $O(M)$ Zeit berechnet werden. Zur Verdeutlichung ist in der Abbildung 10 das neue Rechteck schwarz gefüllt und die beiden Vergrößerungen schraffiert eingezeichnet.

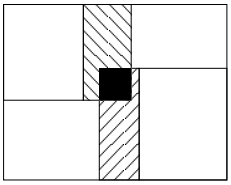


Abbildung 10: Vergrößerung der Fläche

Bei `splitNode` werden die $M+1$ Rechtecke in zwei Gruppen aufgeteilt. Dazu werden zuerst zwei Initialrechtecke (Seeds) ausgewählt: für jedes Rechteck-Paar wird berechnet, wie viel Fläche *außerhalb* der beiden begrenzt durch ihre MBB liegt (in der Abbildung 11 schraffiert gezeichnet). Die zwei Rechtecke mit maximaler „Außenfläche“ können in $O(M^2)$ Schritten berechnet werden: jede Kombination der Rechtecke muss betrachtet werden. Dann werden die restlichen Rechtecke schrittweise in die Gruppe eingefügt, dessen Fläche sie am wenigsten vergrößern. Da dies $M-1$ mal durchgeführt wird und jedes mal nur die Flächenvergrößerung der zwei Rechteckgruppen berechnet werden muss, erfordert es $O(M)$ Zeit.

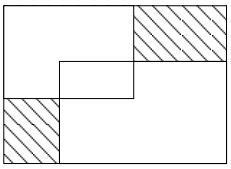


Abbildung 11: Außenfläche

R-Tree mit linearem Split

In [Gut84] wird neben dem quadratischen Algorithmus ein weiterer mit linearer Laufzeit beschrieben. Der Algorithmus für `chooseSubtree` wird nicht geändert.

Um bei `splitNode` Rechenzeit zu sparen, werden die Initialrechtecke mit einer einzigen Iteration über die Subrechtecke bestimmt. Dazu werden in allen Dimensionen das Maximum der niedrigeren und das Minimum der höheren Rechteckkante bestimmt. Dieser Abstand wird zwecks Normalisierung durch die Gesamtbreite aller Rechtecke geteilt. Die beiden Initialrechtecke sind die am weitesten entfernten bezüglich der Dimension mit dem größten normalisierten Abstand. Die *Seeds* werden also in $O(M)$ Zeit berechnet. Die weiteren Rechtecke werden wieder wie beim quadratischen Split in $O(M)$ Schritten gruppiert.

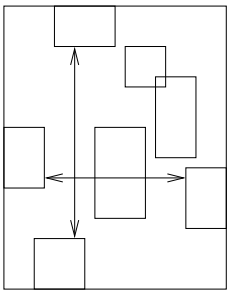


Abbildung 12: Abstand der äußeren Recktecke

R*-Tree

Durch Experimente mit verschiedenen Auswahlkriterien und Datensätzen wurden in [Beck90] die Algorithmen des R*-Tree bestimmt. Dieser führt das Wiedereinfügen (*Reinsert*) von Rechtecken statt eines Splits ein.

In der `chooseSubtree` Funktion wird das Rechteck ausgewählt, dessen Überlappung mit allen anderen Rechtecken der Gruppe am wenigsten wächst, wenn das neue Rechteck hinzugefügt wird. Dies soll die Überlappung des Baums reduzieren. Im Abschnitt R-Tree Varianten wird diese Überlappung als Gütekriterium des R*-Tree durch Messungen untersucht.

Die Berechnung von `chooseSubtree` ist relativ aufwendig: es wird für jedes Rechteck durch Iteration über alle anderen die Vergrößerung der Überlappung berechnet. Damit beträgt die Laufzeit $O(M^2)$ mit hoher Konstante wegen der komplexen Berechnung. In der Abbildung 13 ist das neue Rechteck schwarz und die Vergrößerung des Overlap schraffiert gezeichnet. Daher wird beim R*-Tree diese Überlappungsberechnung nur für die inneren Knoten durchgeführt, deren Einträge auf Blätter zeigen. Auf allen anderen Ebenen wird die `chooseSubtree` des original R-Tree verwendet.

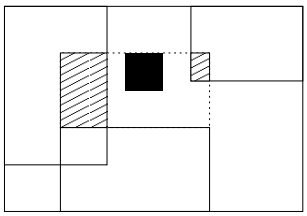


Abbildung 13: Vergrößerung der Überlappung

Im R*-Tree ist auch `splitNode` wesentlich aufwendiger. Bezüglich jeder Dimension werden zuerst alle Einträge einmal nach der höheren Kante und einmal nach der niedrigeren Kante sortiert. Aus dieser Sortierung bildet man $M - 2m + 2$ Verteilungen, indem für die k-te Verteilung der ersten Rechteckgruppe die ersten Einträge $(m-1)+k$ zugeordnet und der zweiten Gruppe die restlichen. Hierdurch entstehen in jeder Verteilung zwei Gruppen mit mindestens m Einträgen.

Für jede dieser Verteilungen wird die Überlappung der beiden Gruppen berechnet. Die Verteilung mit der niedrigsten Überlappung wird verwendet.

In der Abbildung 14 ist eine Verteilung gezeichnet, welche sechs Rechteck in der ersten Gruppe und zehn in der zweiten Gruppe hat. Für $M = 16$, $m = 6$ ist dies also die zweite Verteilung. Die Überlappung der MBB der beiden Gruppen ist schraffiert.

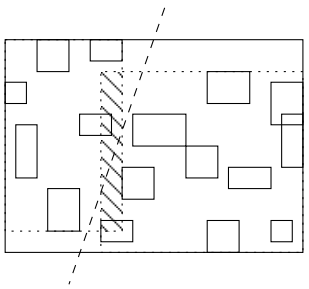


Abbildung 14: Überlappung der zweiten Verteilung

In [Beck90] wird der Zeitaufwand berechnet: Bezüglich jeder Dimension wird zwei Mal sortiert, was $O(M \log(M))$ Zeit bedarf. Die Berechnungen auf den Verteilungen brauchen weitere $O(M^2)$ Schritte.

Um eine bessere Neuverteilung der Rechtecke zu erlauben, die am Anfang eingetragen werden, wird die *Reinsert* Technik verwendet. Bevor ein (relativ aufwendiger) `splitNode` benutzt wird, werden stattdessen r Rechtecke entfernt und neu eingefügt. Es muss $0 < r < M - m$ gelten. Dies wird auf jeder Ebene des Baums höchstens einmal pro Einfügevorgang ausgeführt, dann wird normal `splitNode` benutzt, um überlaufende Knoten zu behandeln.

Um die r Rechtecke zu ermitteln, wird von jedem der $M+1$ Rechteck der Mittelpunkt bestimmt. Es werden die r Rechtecke benutzt, deren Mittelpunkt am weitesten von dem Mittelpunkt der ganzen Gruppe entfernt liegt. Die r ausgewählten Rechtecke müssen mitsamt ihrer Kinder auf derselben Ebene im Baum neu eingefügt werden. Damit auf einer Ebene neue Knoten entstehen können, darf *Reinsert* nicht nochmals benutzt werden. Laut [Beck90] ergibt sich ein 20% – 50% Leistungsgewinn durch die Wiedereinfügungen.

In Abschnitt 4.3 wird die Aufbauzeit und die Anfragegeschwindigkeit des R*-Tree mit den ersten zwei Varianten experimentell verglichen.

R$^+$-Tree

Der Hauptansatz des R$^+$-Tree [Sel87] ist die Überlappung der Rechtecke innerhalb eines Knoten durch Aufbrechen der Rechtecke weiter zu reduzieren.

Eine Überlappung bedeutet, dass bei einer Bereichsanfrage zwei Pfade im Baum betrachtet werden müssen. Die Geschwindigkeit einer Suchanfrage hängt wesentlich von der Anzahl dieser Verzweigungen ab.

Der R$^+$-Tree erlaubt es ein Rechteck aufzuteilen, um eine Verteilung zu finden, welche keine Überlappung hat. Hierfür muss jedoch ein Eintrag mehrmals in dem Baum gespeichert werden. Damit werden auch Einträge bei der Suche mehrfach vorkommen und es muss eine externe Duplikatprüfung stattfinden. Dadurch reduziert sich in der Graphanwendung der Gewinn der Aufteilung wieder. Aus diesem Grund wurde der R$^+$-Tree in dieser Arbeit nicht implementiert.

3 Umsetzung - Implementierung

3.1 Attributkodierung

In dem Graph-Datenhaltungsserver werden neben den in 2.1 beschriebenen Grundstrukturen des Graphen verschiedene Speicherstrukturen für die Attribute der Knoten und Kanten definiert. Ein Attribut hat einen Namen, einen Typ und eventuell einen Defaultwert.

3.1.1 AnyType

Als ersten Baustein wird eine Klasse benötigt, welche einen Wert zusammen mit dessen Typ speichert. Erst durch diese Klasse kann in weiteren Funktionen mit Attributwerten von beliebigem Typ gearbeitet werden.

Hierzu kapselt die `AnyType` Klasse Operationen auf verschiedene Attribut-Typen. Intern besteht die Klasse aus einem Typ-Identifier für einen der in Tabelle 1 aufgeführten Typen und einer `union` der dazugehörigen C++ Datentypen.

`bool`	1 Bit		
`char`	1 Byte	`byte`	1 unsigned Byte
`short`	2 Bytes	`word`	2 unsigned Bytes
`integer`	4 Bytes	`dword`	4 unsigned Bytes
`long`	8 Bytes	`qword`	8 unsigned Bytes
`float`	4 Bytes	`double`	8 Bytes
`string`	1 Byte + n Zeichen mit $n < 256$		
`longstring`	4 Bytes + n Zeichen mit $n < 2^{32}$		

Tabelle 1: `AnyType` Typentabelle

Ein Objekt der Klasse `AnyType` belegt immer mindestens zwölf Bytes. Die `union` benötigt soviel Platz wie der größte Datentyp, wobei Zeichenketten extern mittels `std::string` verwaltet werden. Dazu kommt noch ein Feld für den aktuellen TypId, das vom Compiler zwecks `struct` Alignment je nach Architektur auf vier bzw. acht Bytes hoch gerundet wird.

Die Klasse beinhaltet verschiedene Funktionen, wie `setString`, `getInteger`, überladene arithmetische Operatoren und spezielle Vergleichsoperatoren, um mit den enthaltenen Werten zu arbeiten. Die Typen werden bei Bedarf konvertiert, wobei eine `ConversionException` auftritt, wenn beispielsweise „abc“ zum Integer konvertiert werden soll. Eine genaue Konvertierungstabelle findet man im Anhang A.2. Bei binären Operatoren wird versucht, in dem „höheren“ Datentyp zu rechnen. Dabei ist alles erlaubt, was intuitiv typkonform ist.

3.1.2 AttributeProperties und Attribut Speicherung

Als nächsten Baustein bedarf es einer Klasse, um die Attribut-Definition darzustellen. Eine Attribut-Definition besteht zunächst aus einem Namen für das Attribut, ein Datentyp aus der AnyType-Typentabelle und einem Defaultwert. Da der Defaultwert genau den Datentyp des Attributs hat, werden diese zusammen gelegt. Aus diesem Grund ist die `AttributeProperties` Klasse von `AnyType` abgeleitet. Diese drei Werte bilden einen ersten Ansatz für die Attribut-Definition.

Diese `AttributeProperties` Objekte werden dann mittels `std::vector` zu einer `AttributePropertiesList` aneinander gehängt. Diese Liste enthält die vollständige Attributdefinition von Knoten oder Kanten.

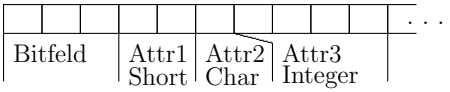


Abbildung 15: Form eines Attributdatensatz

Zu jedem Objekt des Graphen ist ein Satz der Attribute zu speichern. Hierbei können einige dem jeweiligen Defaultwert entsprechen, und diese Attribute sollen kompakter dargestellt sein. Daher muss für jedes Attribut ein Default-Bit im Datensatz enthalten sein; ist das Bit gesetzt, so hat das Attribut den Defaultwert. Ein Attribut-Datensatz beginnt mit einem Default-Bitfeld. Alle anderen Attributwerte, welche nicht dem Default entsprechen, werden hinter dem Default-Bitfeld ohne zusätzliche Strukturierung aneinander gereiht. In der Folge werden alle Attributwerte weggelassen, die gleich dem Defaultwert sind. Dabei werden `bool` Attribute direkt im Bitfeld gespeichert. Diese Aneinanderreihung ist möglich, da durch die Attributdefinition die Größen der folgenden Datenwerten festgelegt sind.

Um einen Attributwert auszulesen, muss für jedes der vorhergehenden Attribute zuerst im Default-Bitfeld nachgeschaut werden, ob dieses in der Datenfolge enthalten ist. Auch um die Länge des Datensatzes zu bestimmen, muss das Default-Bitfeld ausgewertet und die Längen von variablen Datentypen geprüft werden.

`Varying` Attribute

Dieses Grundkonzept wird erweitert, denn es gibt im Graphen mindestens zwei wichtige Attribute, für die es keinen Defaultwert gibt: die Koordinaten. Gerade auf diese soll auch der Zugriff beschleunigt werden. Dazu werden `varying` Attribute eingeführt, deren Defaultwert nicht benutzt wird. Diese stehen immer in einem Attributdatensatz und so ist direkter Indexzugriff auf diejenigen möglich, welche vor dem ersten non-`varying` stehen.

Daher sollten `varying` Attribute nur am Anfang der Definitionsliste verwendet werden. Für sie wird kein Bit im Defaultbitfeld reserviert. In der Klasse `AttributeProperties` müssen zwei weitere Variablen eingefügt werden, um den Zugriff für `varying` zu beschleunigen und um das zu einem non-`varying` Attribute gehörigen Default-Bit zu kennzeichnen. Diese Werte müssen in der Liste vorberechnet werden.

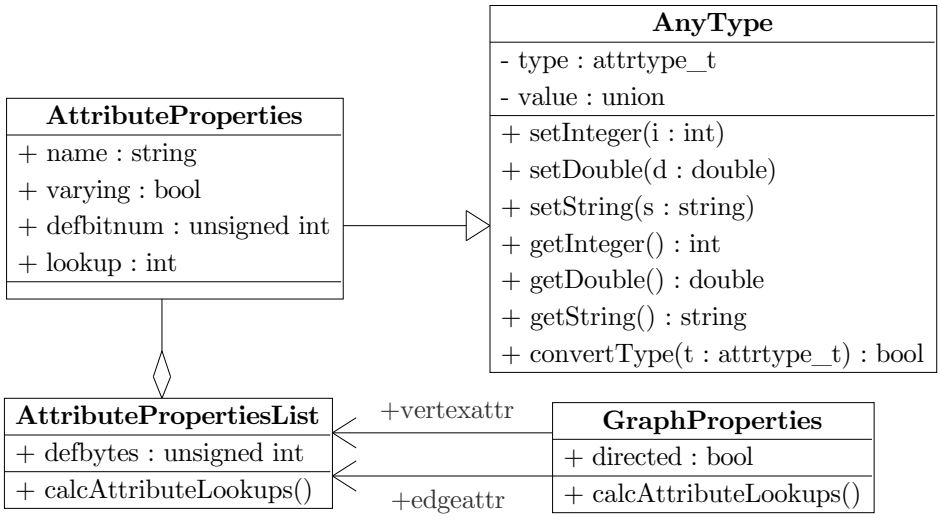


Abbildung 16: AttributeProperties UML Diagramm

Zwei der `AttributePropertiesList` zusammengesetzt ergeben ein `GraphProperties` Objekt: eine Attribut-Defintionenliste für Knoten und eine für Kanten. Darüber hinaus kann der Graph als ungerichtet markiert werden. Ist diese Einstellung aktiv, werden bei allen Funktionsaufrufen der Applikation die Reihenfolge der Kanten QuellID und ZielID sortiert.

UML Diagramm

Zusammengefasst ergeben die Attribut-Klassen die in Abbildung 16 dargestellte Klassenstruktur, welche im Datenhaltungs-Server die Eigenschaften des Graphen repräsentieren.

3.1.3 Attribut-Datensatzarray

Jedem Knoten und jeder Kante im Graph kann ein Attributdatensatz zugeordnet werden. Diese Datensätze müssen auf möglichst platzsparende Weise organisiert sein. Dazu wird dieselbe Datenanordnung wie beim *Adjazenz-Array* verwendet.

Alle Attributdatensätze werden nach KnotenID aufsteigend in einem Bytearray gespeichert. In das Knoten-Objekt des Adjazenz-Array wird die Indexnummer des Datensatzes im Array eingetragen. Der Index des nachfolgenden Knoten-Objekts bezeichnet dann den Beginn des nächsten Datensatzes und gleichzeitig das Ende des vorherigen. So kann die Länge des Attribut-Datensatzes festgestellt werden ohne diesen zu analysieren.

Für Kanten wird dasselbe Verfahren verwendet, indem im Kanten-Objekt eine Indexnummer eingetragen wird.

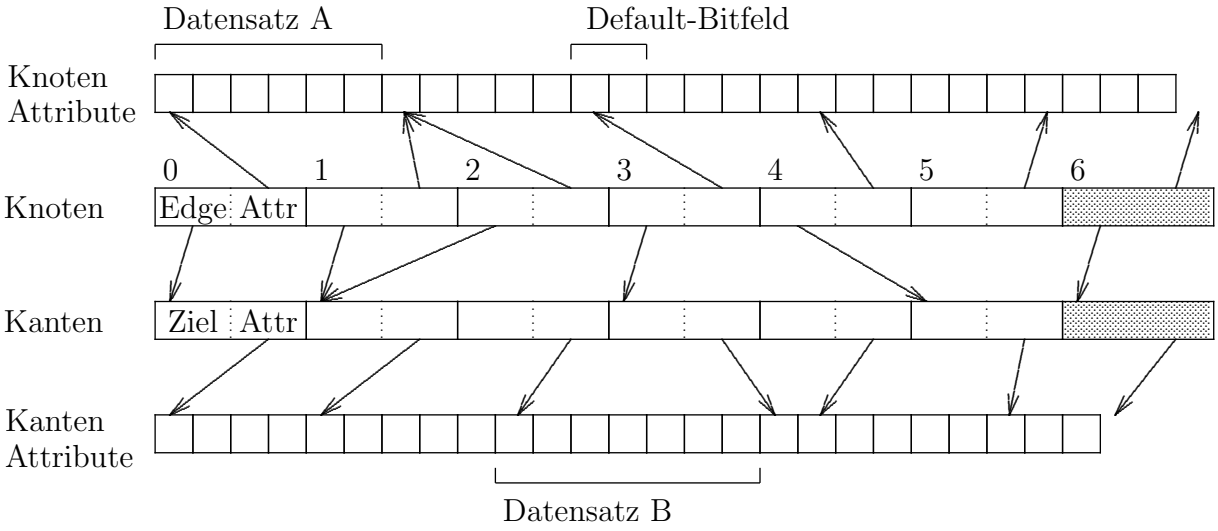


Abbildung 17: Graphdatenarrays

Die Abbildung 17 zeigt, wie die vier Arrays indiziert werden. Deutlich ist zu erkennen, dass Knoten 0 der Attribut-Datensatz A zugeordnet ist. Die zweite Kante des Knoten 2 hat den Datensatz B. Jeder dieser Datensätze beginnt mit dem Default-Bitfeld.

Wenn ein Knoten als gelöscht gelten soll, wird ein leerer Attributdatensatz gespeichert, wie oben für Knoten 1. Daher muss jeder gültige Attributdatensatz mindestens ein Byte als Defaultbitfeld enthalten.

Die Terminierungsknoten von Knoten- und Kantenarray zeigen auf das erste ungültige Byte der Attributarrays. Diese Indizes entsprechen der Gesamtgröße der jeweiligen Attributarrays.

Diese Darstellung braucht neben den eigentlichen Datenbytes ein vier Bytes Index in jedem Knoten- und Kanten-Objekt. Für den Straßengraph bedeutet dies etwa $(18+22)\cdot 10^6 \cdot 4$ Bytes $=$ 160 Megabytes für die Indizes in den Objekten (18 Millionen Knoten und 22 Millionen Kanten).

3.2 ChangeTimeline Klassenkomplex

Die globalen Graph Daten sind in möglichst platzsparenden Datenstrukturen gespeichert, welche jedoch wenig Flexibilität bieten. Um dies auszugleichen, sind die lokalen Änderungen in der flexiblen `Changelist` organisiert. Diese lokale Überlagerung enthält mehrere Hashmaps, in denen die geänderten Attribut-Datensätze als ganzes gespeichert werden und nicht jeder geänderte Attributwert einzeln.

Beim Zugriff auf einen Knoten oder Kante wird zuerst in der Hashmap nachgeschaut, ob eine Änderung vorliegt. Wenn ja, so wird der gefragte Attributwert dort ausgelesen, sonst wird der Wert in den globalen Graphdaten verwendet. Wird in der Hashmap ein leerer Datensatz ohne Defaultbitfeld gespeichert, so gilt der Knoten als gelöscht.

Die Klasse `Changelist` enthält einen einfachen Überlagerungs-Graphen. Um Animationen zu speichern, wird die `Changelist` erweitert zu der `ChangeTimeline`. Sie repräsentiert eine Folge von *Animations-Frames*, in denen jeweils eine Folge von Änderungen aufgezeichnet ist. Die Abbildung 18 zeigt eine solche Folge von Frames.

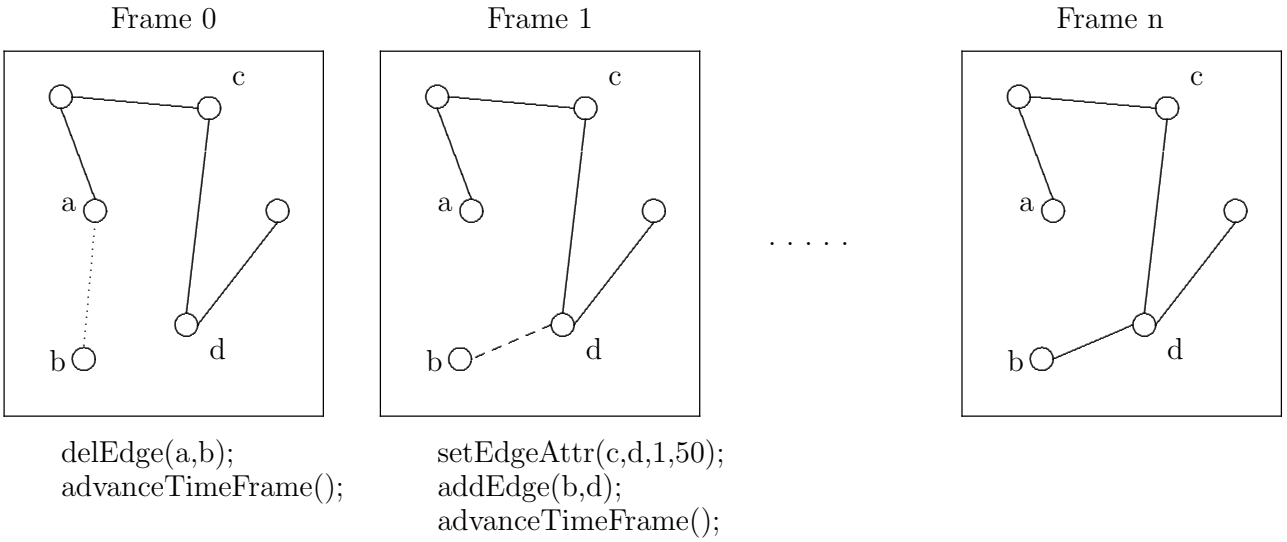


Abbildung 18: Change Timeline

Die Klasse `ChangeTimeline` enthält zwei `Changelist` Instanzen: eine für den Frame 0 und einen für den letzten Frame n, welcher alle Änderungen enthält.

Die `Changelist` für Frame 0 wird bei `getArea`-Anfragen verwendet. Änderungen im Frame 0 werden direkt in den Basis-Graph der Rückgabe eingearbeitet und nicht als Animationsfolge übertragen.

Die zweite Instanz für Frame n wird gebraucht, wenn die Applikation Informationen anfragt oder neue Änderungen einträgt. Wird beispielsweise eine bereits modifizierte Kante wieder geändert, so kann mit der zweiten `Changelist` direkt auf den modifizierten Datensatz zugegriffen werden.

Die Folge der von der Anwendung aufgerufenen Änderungsfunktionen wird direkt in das vom Client verstandene Animations-Protokoll umgesetzt. Die Änderung wird in ein `ChangeEntry` umgewandelt, und in der sequentiellen Speicherstruktur des aktuellen `ChangeFrame` eingetragen. Die `ChangeTimeline` enthält dann die Folge der `ChangeFrame`-Objekte.

Im Anhang A.3.1 befindet sich ein UML Diagramm, welches die Beziehungen zwischen diesen Klassen verdeutlicht.

3.3 Graph Datenhaltungsklassen

In der Aufgabenstellung 1.2 wird beschrieben, wie temporäre Änderungen von permanenten Graphdaten getrennt werden soll. Intern wird dies durch die Aufteilung in globale und lokale Graphdaten nach Abschnitt 2.2 durchgeführt. In der Implementierung ergeben sich drei Datenhaltungs-Klassen: `GraphData`, `GraphContainer` und `GraphConnection`. Das UML Diagramm im Anhang A.3.2 zeigt die Beziehungen zwischen diesen Klassen. Im Folgenden soll deren Bedeutung erläutert werden.

Als erste Basisklasse dient `GraphData`, welche die vier Datenarrays gruppiert. Sie stellt die Grunddatenstrukturen mit Attributwerten des globalen Graphen dar.

GraphLoader

Von `GraphData` ist eine Klasse `GraphLoader` abgeleitet, welche ein beschleunigtes Laden der globalen Daten ermöglicht. Doch hat diese Klasse Einschränkungen bezüglich der Reihenfolge der geladenen Knoten und Kanten, denn sie schreibt die vier Datenarrays direkt. Dadurch ergeben sich folgende Bedingungen an die Ladereihenfolge:

- Die KnotenIDs in `addVertex` müssen aufsteigend sein.
- Die Tupel (QuellID, ZielID) müssen für `addEdge` lexikographisch aufsteigend sein.
- Bezüglich eines Knoten oder einer Kante müssen die Attributnummern bei `setVertexAttr`- bzw. `setEdgeAttr`-Aufrufen aufsteigen.
- Es wird nicht geprüft, ob die Knoten einer Kante existieren. Dadurch sind die Folgen von `addVertex` und `addEdge` unabhängig.

Nachdem alle Werte in den `GraphLoader` geladen wurden, müssen mittels `terminate` oder `finish` die Terminierungsobjekte angefügt werden.

GraphContainer

Da `GraphData` nur die vier Grundstrukturen enthält, muss diese um die Indexstruktur erweitert werden. Dazu ist `GraphContainer` von `GraphData` abgeleitet und enthält einen R-Tree für jeden Wert der z-Koordinate. In Abschnitt 3.4 wird der Aufbau dieser R-Trees erklärt.

Die Funktion der `GraphContainer` Klasse ist, die Zugriffe auf Grundstruktur und Indexstruktur zu koordinieren. Insbesondere enthält sie daher neben Methoden, welche den R-Tree aufbauen, die eigentliche Implementierung von `getArea` und `getNearestNeighbor` mittels Range Queries.

Alle Methoden, welche auf Graphdaten arbeiten, nehmen als Parameter eine `Changelist`. Die darin gespeicherten Änderungen können dann in die berechneten Ergebnisse einfließen.

Weiter bietet die Klasse zwei Methoden, um den kompletten globalen Graphen in Form einer Snapshot-Datei zu speichern. Dieser Snapshot enthält die `GraphProperties`, die vier Datenarray und alle R-Trees. Da alle Daten in der vom Server verarbeiteten Form gespeichert werden, kann die Snapshotdatei sehr schnell wieder geladen werden.

Die `GraphContainer` Klasse bietet auch drei Funktionen, um Knoten und Kanten auszulesen. Diese liefern `VertexRef-` bzw. `EdgeRef-`Objekte zurück, welche aus einem Verweis auf die `GraphData`, der Knotennummer bzw. den Kantennummern und möglicherweise aus den in der `Changelist` gespeicherten Änderungen für dieses Objekt bestehen. `VertexRef` und `EdgeRef` erlauben effizienten Zugriff auf mehrere Attributwerte nacheinander.

GraphConnection

Die Klasse `GraphConnection` stellt einen *Kontext* dar, in dem die Applikation den Anwendungsfall eines Client bearbeitet. Verbindet sich ein Client zum Server, so wird eine Instanz der `GraphConnection` erzeugt, welche die lokalen Änderungen der Applikation speichert und durch `getArea` dem Client verfügbar macht.

Dazu enthält die Klasse ein `ChangeTimeline` Objekt, welches die Animationsfolge aufzeichnet.

Die Klasse `GraphConnection` bietet sowohl die lesenden Funktionen des `GraphContainer`, als auch die schreibenden Funktionen der `ChangeTimeline`. Dabei wird der letzte Frame der `ChangeTimeline` als Argument für die Funktionen von GraphContainer verwendet, die eine `Changelist` benötigen.

`GraphConnection` setzt die vorangehenden Grundklassen zusammen, um einen Änderungskontext für die Applikation zu bieten. Eine Applikation leitet diese Klasse ab und erweitert sie durch die gewünschte Berechnungsfunktionen und eventuell dazu benötigte Variablen.

3.4 R-Tree

Als Indexstruktur für die Graphdaten wird der R*-Tree verwendet. Eine Übersicht über die Funktionsweise des R-Tree wird in Abschnitt 2.3.3 gegeben.

Bevor eine Neuimplementierung des R*-Tree vorgenommen wird, werden folgende vier frei im Internet zugänglichen, existierenden Implementierungen der R-Tree Variationen auf Eignung für die Graphvisualisierung untersucht.

- Die urspüngliche Implementierung aus 1984 von A. Guttman zu [Gut84] ist sehr schlicht, kann aber nicht direkt eingebunden werden, denn es handelt sich um Testprogramme.
- PostgreSQL [PgSQL] enthält im Quellcodeverzeichnis `src/backend/access/rtree/` eine relativ komplexe C Implementierung, die auf anderen internen Modulen aufbaut.
- Ondrej Pavlata hat in 2003 eine Visual C++ Implementierung [RTree2] geschrieben, welche jedoch nicht die R* Variante enthält.
- An der University of California, Riverside, wurde eine gute dokumentierte C++ Library [RTree1] von *spatial indices* erstellt. Sie enthält neben R*-Tree auch MVR-Tree und TPR-Tree. Der Quellcode ist gut geschrieben und dokumentiert, besteht jedoch aus einem komplexen Klassengerüst mit viele virtuelle Funktionsaufrufe in den Kernschleifen.

Keine der obigen Bibliotheken kann verwendet werden, denn die Graphvisualisierung erfordert höhere Speichereffizienz und Performance, als sie durch Einbinden der existierenden Quelldateien erreicht werden kann.

Auf Grundlage dieser Implementierungen und den Originalveröffentlichungen ([Gut84] und [Beck90]) wurde für diese Arbeit eine neue speziell optimierte C++ Templateklasse geschrieben, welche auch auf andere Objekte als Kanten angewandt werden könnte.

Die Abbildung 19 zeigt einen von der Klasse aus dem Autobahnnetz von Deutschland erzeugten R*-Tree. Jeder Knoten des Baums enthält zwischen 12 und 20 Rechtecke. Die 31829 hellgrauen Rechtecke sind die Kanten, welche die Autobahnen repräsentieren. Die 1431 Rechtecke der Blätter sind grün eingezeichnet und gruppieren durchschnittlich 22 Autobahnsegmente. Die darüberliegende Ebene ist blau dargestellt und die oberste Ebene rot. Insgesamt hat der R-Tree 4 Ebenen und enthält 33326 Rechtecke.

Um die Kanten nach Wichtigkeit zu indizieren, wird im Graph-Server für jede Ebene der z-Achse ein eigener R-Tree verwendet. Bei der `getArea` Query werden diese in einer festgelegten Reihenfolge abgefragt, bis genügend Kanten in der Ergebnismenge enthalten sind.

Auch `getNearestNeighbor` wird mittels der Bereichssuche im R-Tree implementiert. Im Unterschied zu getArea wird nur ein Knoten und eine Kanten zurückgeliefert.

In den Blättern des R-Trees stehen normalerweise die Koordinaten der Datenrechtecke und Verweise auf die indizierten Objekte. Für die Graphdaten besteht die Information aus Quell- und ZielID der Kante und das Datenrechteck aus den Koordinaten der adjazenten Knoten. Dadurch werden die Koordinaten doppelt gespeichert oder gar mehrfach, wenn ein Knoten mehrere inzidente Kanten hat. Um diese Doppelspeicherung zu verhindern, werden in den Blättern keine Rechtecke in den Einträgen gespeichert und bei Bedarf die MBB einer Kante durch Callback-Funktionen anhand der Quell- und ZielID direkt aus den Graphdaten erstellt. Da die Korrektheit der Struktur des R-Trees von den Graphdaten abhängig geworden ist, muss besondere Sorgfalt bei der Reihenfolge von Veränderungen walten. Der Speicherbedarf des R-Trees verringert sich durch diese Technik um circa 65%.

3.5 Filter- und Selektionsparser

Als Parameter für die Client-Funktionen `getArea` und `getNearestNeighbor` muss eine Liste von gewünschten Attributen übergeben werden.

In der Regel sind die Koordinaten der Knoten mit einer höheren Genauigkeit (z.B. als `int`) gespeichert, als es zum Zeichnen auf dem Client notwendig ist. Anstatt die Koordinatenwerte zum Client zu übertragen, ist es angezeigt die Koordinatentransformation auf dem Server zu berechnen und die Pixelkoordinaten in einem kleineren Datentyp zu übertragen. Dies muss in den Parametern zu `getArea` kodiert werden.

Eine Möglichkeit ist die Parameterliste binär zu kodieren, etwa als Liste von Attributnummern. Dabei wird der Koordinatentransformation eine spezielle Nummer gegeben und dahinter die zur Transformation benötigten Parameter angehängt. Diese Kodierung erscheint für diese Arbeit zu unflexibel.

Stattdessen implementiert der Datenhaltungs-Server einen Anfragestring-Parser, welcher allgemeinere arithmetische Operationen erkennen kann. Die benötigte Koordinatentransformation besteht aus einer Subtraktion und einer Multiplikation mit einer Gleitkommazahl. Zum Parsen

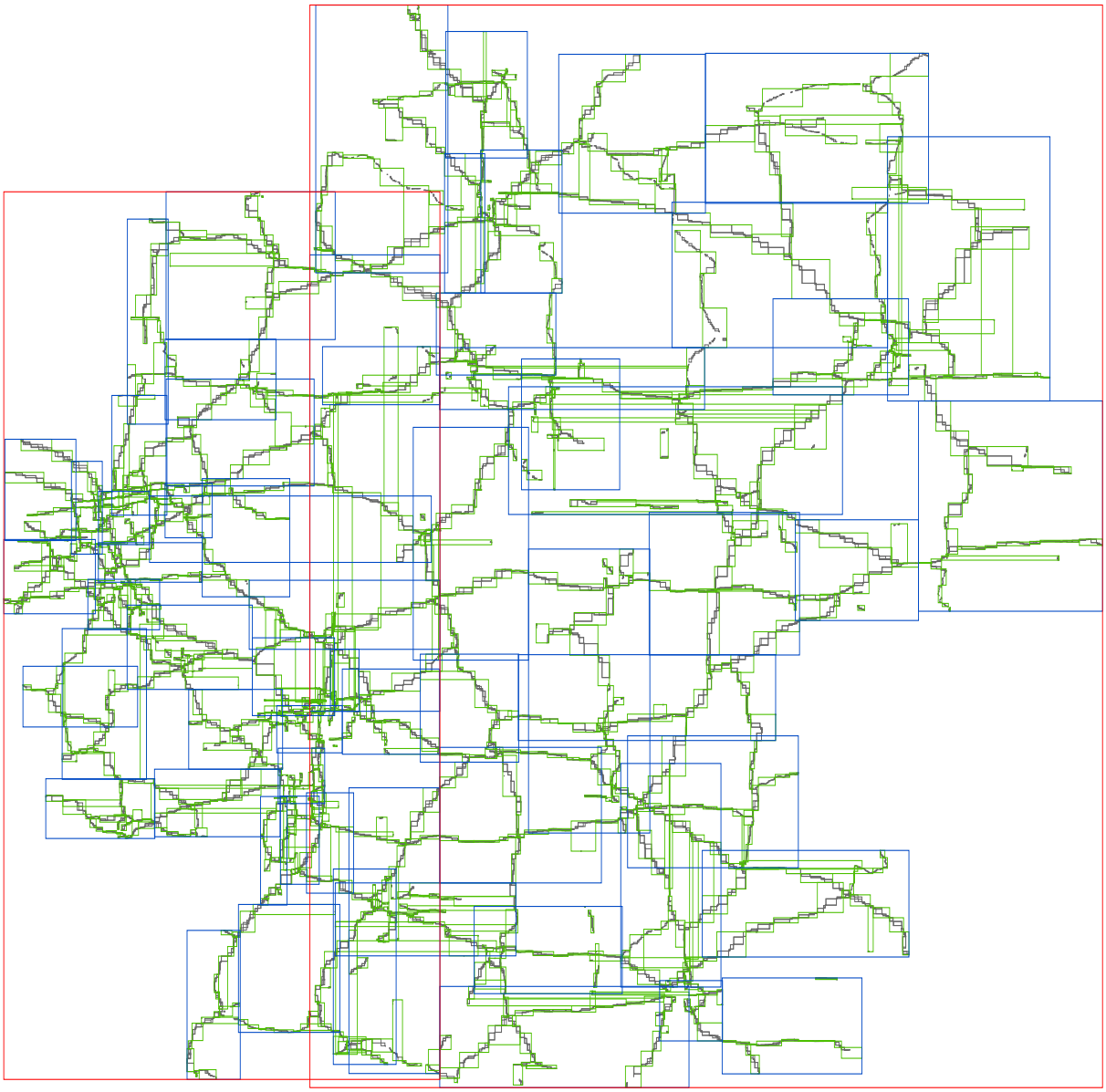


Abbildung 19: R-Tree der deutschen Autobahnen

verwendet der Server die C++ Parser Library Boost.Spirit [Spirit], mit der die Grammatik direkt im C++ Code durch operator-Überladung geschrieben werden kann.

Es werden zwei verschiedene Parser benötigt: einen für die Auswahl der zurückgelieferten Attribute und einen als Filter der betrachteten Knoten und Kanten.

Der Parser kann arithmetische und boolsche Operatoren, Vergleichsbedingungen, Klammern, Konstanten und Attributnamen erkennen. Nach dem Parsen werden diese Grundoperationen in einem Parsebaum organisiert, welcher für jede der ausgewählten Knoten und Kanten den Ergebniswert berechnet. Konstante Teile des Berechnungsbaums werden automatisch zusammengefaltet.

Selektionsausdrücke

Der Selektionsausdruck besteht aus einer Folgen von komma-getrennten Berechnungsausdrükken, oder aus einem einzigen "`*`", falls alle verfügbaren Attribute geliefert werden sollen.

Berechnungsausdrücke bestehen im einfachsten Fall aus dem Namen eines Attributs. Mit diesen „Variablen“ kann dann auf die übliche Art mit `+`, `-`, `*`, `/`, den Klammern und Konstanten gerechnet werden, wobei die Punkt-vor-Strich Regel gilt. Falls die Operanden nicht denselben Attributtyp haben, so wird in den „größeren“ der beiden konvertiert und dort gerechnet. Um zur Übertragung die Attributwerte besser zu packen, sollte nach der Rechnung wieder auf einen kleinen Datentyp mittels "`cast` *typ*" herunter konvertiert werden.

Hier eine vereinfachte BNF Grammatik der Knoten-Selektionsausdrücken:

```
select    ::= "*" | attrlist
attrlist  ::= attrexpr { "," attrexpr }
attrexpr  ::= orexpr
orexpr    ::= andexpr | (andexpr "or" andexpr)
andexpr   ::= compexpr | (compexpr "and" compexpr)
compexpr  ::= castexpr | (castexpr ("==" | "!=" | "<" | "<=" | ">" | ">=") castexpr)
castexpr  ::= addexpr ["cast" ("bool" | "char" | "short" | ... )]
addexpr   ::= mulexpr | (mulexpr ("+" | "-") mulexpr)
mulexpr   ::= unaryexpr | (unaryexpr ("*" | "/") unaryexpr)
unaryexpr ::= ["+" | "-" | "!" | "not"] atomexpr
atomexr   ::= constant | ( "(" attrexpr ")" ) | attrname
attrname  ::= "a"-"z" { "a"-"z" | "0"-"9" | "_" }
```

Für den Selektionsausdruck für Kanten kann nicht nur auf Kantenattribute, sondern auch auf die Knotenattribute der beiden adjazenten Knoten zugegriffen werden. Diese werden durch "`src.`*attr*" oder "`tgt.`*attr*" als „Variablen“ referenziert.

Die zum Zeichnen benötigte Koordinatentransformation in den Bildschirmausschnitt sieht dann folgendermaßen aus: `(src.x -` *ursprungX*`) *` *skalierungX* `cast short, ...`

Filterausdrücke

Der Filterausdruck kann verwendet werden, um gewisse Knoten oder Kanten im ausgewählten Bereich auszublenden. Er wird für jedes betrachtete Objekt ausgewertet.

Der Filterstring beginnt entweder mit „`vertex:`“ oder „`edge:`“, wodurch festgelegt wird, ob nach Knotenattribute oder nach Kantenattributen gefiltert wird. Darauf folgt ein Berechnungsausdruck, welcher einen boolschen Wert (ein `AnyType` mit Typ `bool`) zurück liefern muss. Daher muss ähnlich wie bei Java oft eine Vergleichsoperation `!= 0` benutzt werden.

Die BNF Grammatik für Filterausdrücke ist eine einfache Erweiterung der obigen.

```
filter    ::= ("vertex:" | "edge:") attrexpr
```

Die Filterausdrücke können beliebig komplex werden und bieten ein gutes Hilfsmittel, um einen visualisierten Graph zu untersuchen.

3.6 Daten Serialisierung

Die Ergebnisse von `getArea` und `getNearestNeighbor` bestehen aus einer Menge von Knoten und Kanten, von denen durch die Selektionsausdrücke berechnete Attributwerte zurück geliefert werden sollen. Dieses Ergebnis soll auf platzsparende Weise kodiert werden, so dass im Client während der Deserialisierung gleich die Knoten und Kanten gezeichnet werden können.

Während dem Parsen der Selektionsausdrücke wird eine `AttributePropertiesList` erstellt, welche den Auswahlausdrücken entspricht. Jedem Berechnungsausdruck wird ein Attribut zugeordnet, und aus den „Variablen“ der zurückgelieferten Knoten oder Kanten wird ein Attribut-Datensatz berechnet. Das Format dieses Datensatzes wird durch die `AttributePropertiesList` des Selektionsausdrucks bestimmt.

Für die Übertragung zum Client wird eine Bytestrom-Kodierung festgelegt, die aus fünf Blöcken besteht:

1. Eine binärkodierte `AttributePropertiesList` des Knoten-Selektionsausdrucks.
2. Die gleiche Struktur für den Kanten-Selektionsausdruck.
3. Direkt aufeinander folgende Attribut-Datensätze der ausgewählten Knoten.
4. Die Attribut-Datensätze der ausgewählten Kanten in derselben sequentiellen Anordnung.
5. Die Animationsfolge in einer speziellen Kodierung.

Jeder Datenblock legt intern fest, wann er endet. So können die Datenblöcke durch einfache ein-Byte Identifikatoren begonnen werden.

Eine genauere Beschreibung der Kodierung findet man in der `Kodierung.pdf`, die dem Quellcode beliegt. Der C++ Quellcode enthält weiter eine Klasse `GraphParser`, welche Funktionen in einem Callback-Objekt für die in dem Bytestrom kodierten Ereignisse aufruft.

3.7 Java Client

Zusammen mit dem Datenhaltungs-Server entsteht auch ein Testclient in Java. Dieser dient einerseits als Testwerkzeug für den Datenhaltungs-Server, insbesondere für die Datendeserialisierung. Andererseits bietet er eine Grundlage, um ein spezielles Web Applet für die Routenplanungsanwendung des Lehrstuhls für Theoretische Informatik zu schreiben.

Der Java Client sendet `getArea`-Anfragen per CORBA an einen einfachen C++ CORBA-Server, welcher die Datenhaltungs-Bibliothek eingelinkt hat. Der CORBA-Server lädt nur die Kartendaten und stellt sie dem Client per CORBA Interface zur Verfügung.

Im CORBA-Server wird die open-source Bibliothek omniORB verwendet. Java hat seine eigene ORB Implementierung in der Runtime-Library. Das Zusammenspiel dieser beiden CORBA-Implementierungen funktioniert problemlos.

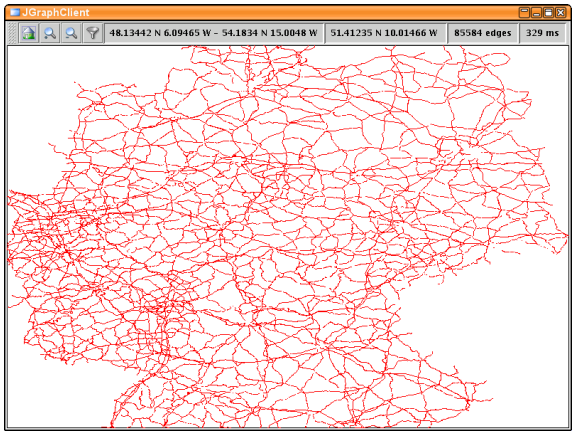


Abbildung 20: Screenshot des Java Client

4 Experimente

In diesem Abschnitt werden verschiedene experimentell ermittelte Daten und Folgerungen daraus dargestellt. Die Versuche wurden alle auf einem Rechner des Lehrstuhls für Theoretische Informatik an der Universität Karlsruhe ausgeführt. Der Rechners war ein Dual Core AMD Opteron 270 mit 2 GHz Taktrate und 4 GB RAM. Die Programme wurden mit `g++ 4.0.2` im 64 Bit-Modus mit der Optimierungsstufe `-O3` kompiliert.

4.1 Kartengrößen

Die am Lehrstuhls für Theoretische Informatik entwickelten Routenplanungsalgorithmen können auf Straßengraphen von verschiedenen Ländern arbeiten. Diese kleineren Karten und der Gesamtgraph von Europa werden von dem Datenhaltungs-Server eingelesen und verarbeitet. Die Tabelle 2 zeigt die Größe der verschiedenen Datenhaltungs-Strukturen.

Karte	Knoten	Kanten	Objekt-Arrays	Attributdaten	R-Trees	Gesamt
Luxemburg	30 747	38 143	538 KB	531 KB	517 KB	1586 KB
Belgien	463 795	594 715	8 269 KB	8 142 KB	7 895 KB	24 307 KB
Niederlande	893 407	1 144 337	15 920 KB	15 675 KB	15 174 KB	46 769 KB
Deutschland	4 378 447	5 504 454	77 210 KB	76 111 KB	73 643 KB	226 964 KB
Europa	18 029 722	22 339 557	315 385 KB	311 176 KB	301 322 KB	927 883 KB

Tabelle 2: Kartengrößen

Die folgenden Messergebnisse werden alle auf dem Europa Straßengraph ermittelt.

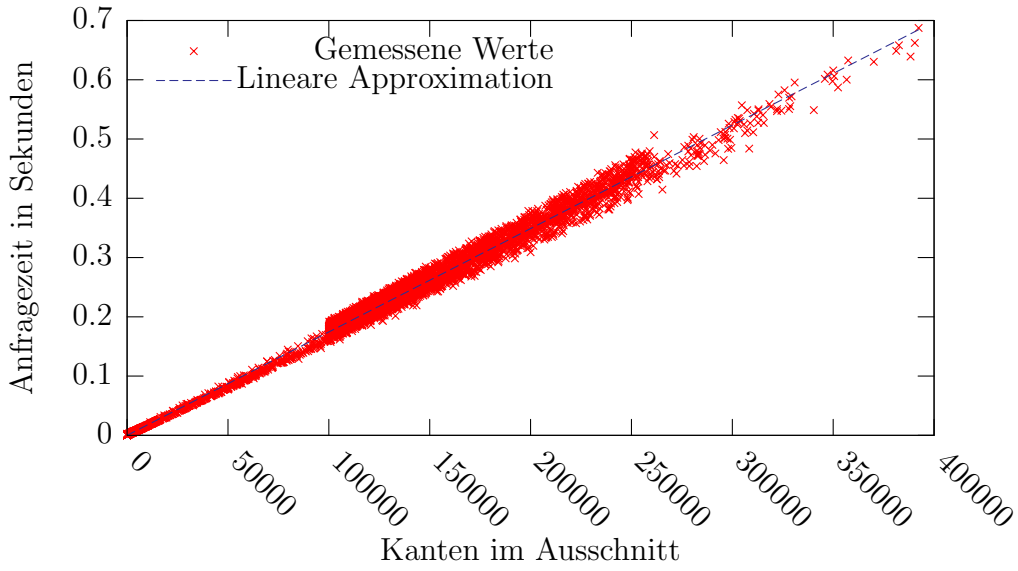
4.2 Gesamtgeschwindigkeit

Zur Leistungsfähigkeit des Datenhaltungs-Server werden verschiedene Messungen durchgeführt. Im Vordergrund steht dabei die Optimierung der Antwortzeit auf eine `getArea`-Anfrage.

Diese Zeit ist abhängig von der Güte des R-Tree und von den Selektionsparsebäumen, welche die Ergebniswerte berechnen. Die Berechnungsfunktionen der Parsebäume wurden durch Profiling so weit wie möglich optimiert.

Zur Untersuchung der Gesamtgeschwindigkeit wird ein Programm verwendet, das pseudozufällige Ausschnitte innerhalb des Europa-Straßengraphen anfragt. Diese Ausschnitte sind quadratisch und die Fläche wird randomisiert (nahezu gleichverteilt). Die Abbildung 21 zeigt das Verhältnis der im Ergebnis gelieferten Kanten zur Antwortzeit.

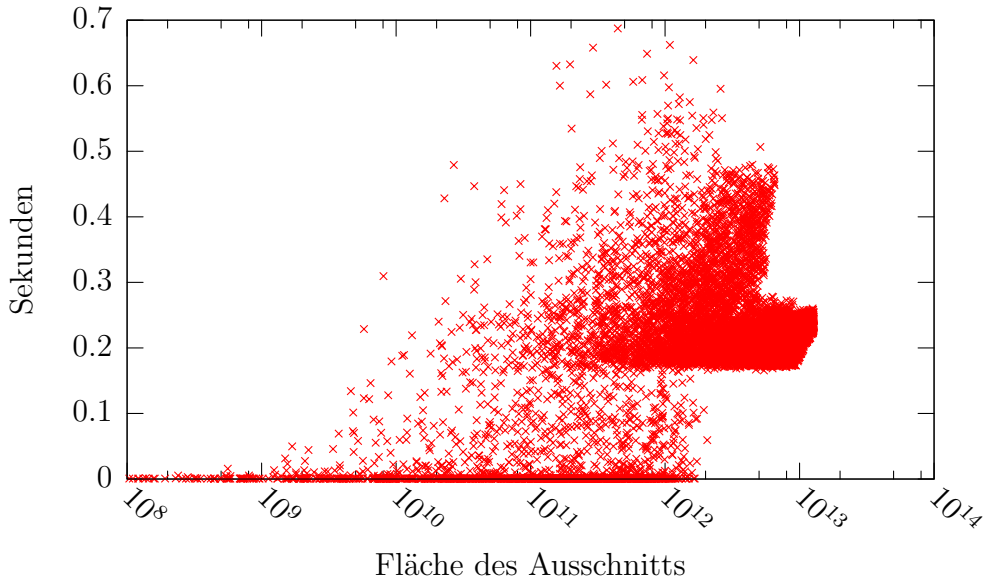
Abbildung 21: 1000 zufällige getArea Queries



Deutlich sieht man, dass die Antwortzeit linear mit den zurück gelieferten Kanten zusammenhängt. Ab der Grenze 100 000, die als `edgeminlimit` Parameter (minimale Anzahl zurück gelieferter Kanten) verwendet wird, zeigt sich eine Streuung der Antwortzeit. In der zusätzliche Verarbeitungszeit wird angefangen, eine neue Ebene zu lesen, die jedoch nicht in das Ergebnis einfließt, da das Maximum der gelieferten Kanten erreicht wird. Es werden nie mehr als 400 000 Kanten zurückgeliefert, da dies der `edgemaxlimit` Parameter ist.

Die Abbildung 22 zeigt wieviel Zeit eine getArea Query in Abhängigkeit von der Fläche des Ausschnitts benötigt.

Abbildung 22: 1000 zufällige getArea Queries



Für kleine Ausschnitte ist die Verarbeitungszeit sehr klein. Ab einem Grenzwert bei 10^{12} ist die

Fläche so groß, dass es keine Quadrate innerhalb des Kartenraums mehr gibt, die wenige Kanten enthalten und so sehr schnell bearbeitet werden können. Alle Anfragen können innerhalb von 0,7 Sekunden beantwortet werden.

4.3 R-Tree Varianten

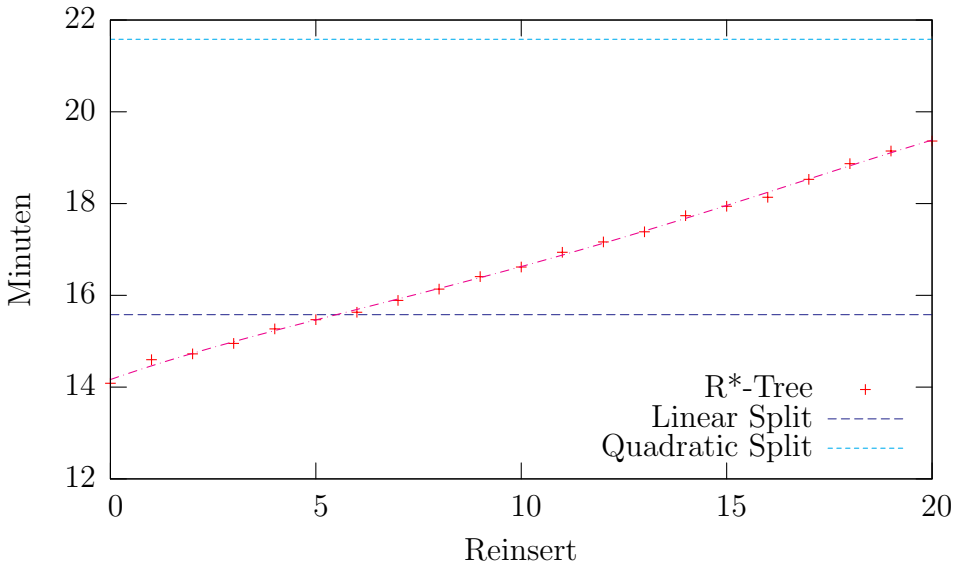
Um die Güte der verschiedenen R-Tree Variationen zu bewerten, wird das vorherige Programm auf verschiedene R-Trees angewandt. Es werden wieder zufällige quadratische Ausschnitte innerhalb des Kartenraums anfragt, wobei die Attributberechnung deaktiviert wird, damit die reine `rangeQuery` Zeit des R-Tree betrachtet wird.

Zu erwarten ist, dass ein R-Tree bei dem die Überlappung der Rechtecke innerhalb einer Gruppe kleiner ist, schneller `getArea` Anfragen beantworten kann, denn es müssen weniger Pfade im Baum durchsucht werden. Insbesondere versucht der R*-Tree dies durch die aufwendige Overlap-Berechnung zu reduzieren.

Ein anderes Qualitätsmerkmal eines guten R-Tree sollte die Fläche sein, welche innerhalb der MBB einer Gruppe, aber in keinem der Rechtecke der Gruppe liegt. Dieser „verschwendete Platz“ sollte möglichst gering sein, um keine unnötigen Pfade im Baum betrachten zu müssen.

Eindeutig zeigt die Abbildung 23, dass die Aufbauzeit des R*-Tree von dem Reinsert-Parameter abhängt. Die Variante mit quadratischem Split benötigt jedoch noch mehr Zeit als jeder R*-Tree.

Abbildung 23: Aufbauzeit der R-Tree Varianten



Trägt man jedoch in Abbildung 24 die Query-Geschwindigkeit in Kanten pro Sekunde als Durchschnitt von 10 000 zufälligen Bereichsanfragen auf, so ist keine Verbesserung durch größere Reinsert-Anzahl sichtbar: das Optimum liegt in der Mitte bei $r = 9$. Die original R-Tree Varianten sind beide etwas langsamer als der R*-Tree. Insgesamt liegt die Geschwindigkeitsverbesserung des R*-Tree zwischen 5% und 8%, welche jedoch kaum ins Gewicht fällt, wenn die Attributwerteberechnung aktiviert ist.

Der R*-Tree ist auch bezüglich aller theoretischen Qualitäts-Merkmale besser. Er hat geringere Überlappungen, einen höhere durchschnittlichen Füllstand der Knoten und so einen kompakteren Baum.

Abbildung 24: Query Speed der R-Tree Varianten

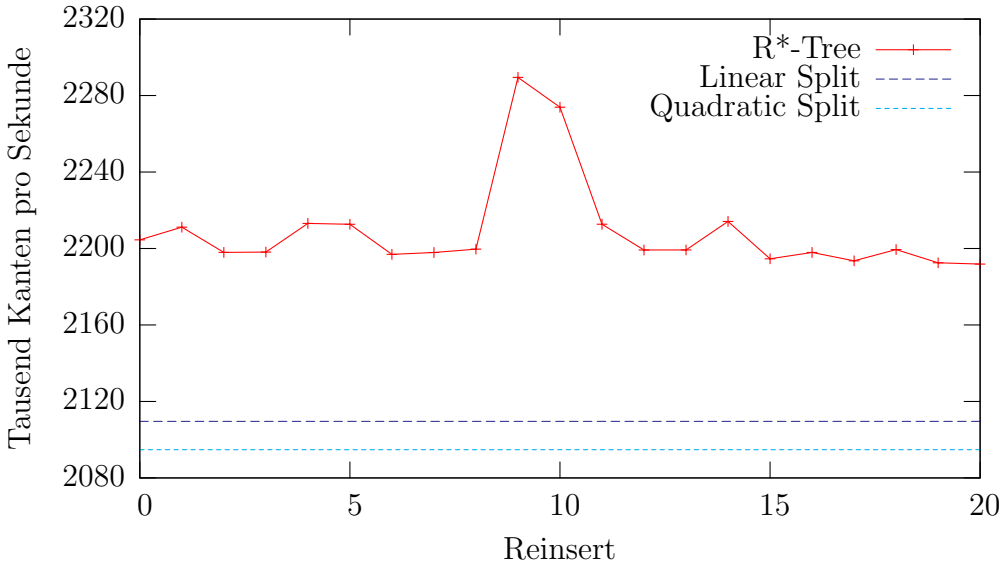
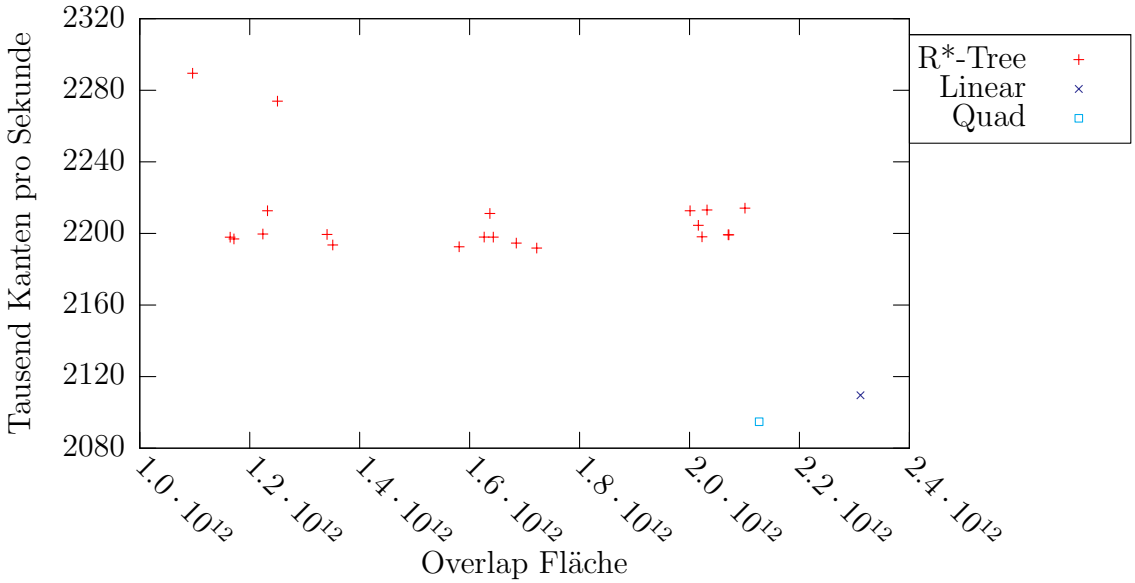


Abbildung 25: Überlappungsgröße der R-Tree Varianten



Die Gesamtsumme aller Überlappung innerhalb einer Gruppe wird in der Abbildung 25 bezüglich der Anfragegeschwindigkeit dargestellt. So sind die Überlappungsgrößen der linearen und quadratischen Variante erheblich höher als die der R*-Trees. Einen direkten Zusammenhang zwischen Überlappungsfläche und Anfragegeschwindigkeit gibt es jedoch nicht.

Insgesamt ist für das Straßen-Datenmaterial der R*-Tree mit 50% Reinsert etwas besser geeignet als die original R-Tree Algorithmen.

4.4 Geschwindigkeit fester Ausschnitte

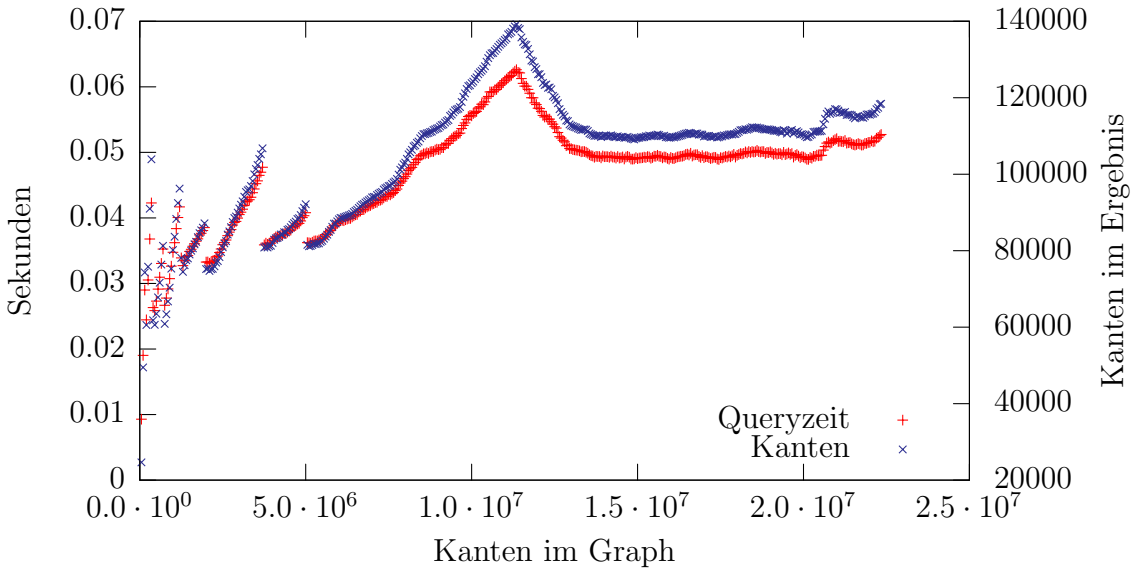
Als letztes Experiment wird der Graph schrittweise aufgebaut: in jedem Schritt werden 50 000 Kanten hinzugefügt. Während des Aufbaus wird nach jedem Schritt eine Reihe von Bereichsanfragen gestellt. Bei den Bereichsanfragen wird die tatsächliche Berechnung der Attributwerte deaktiviert, um ausschließlich die R-Tree-Suchzeit zu betrachten.

Es werden wie in dem letzten Experiment pro Schritt 1 000 zufällige quadratische Bereichsanfragen gemessen. Darüberhinaus werden jeweils zehn Anfragen auf vier feste Ausschnitte gemessen: Karlsruhe Stadt, Karlsruhe mit Umland, Baden-Württemberg und ganz Deutschland. Die 40 feste Ausschnitte werden mit den 1 000 zufälligen Ausschnitten vermischt, damit keine unerwünschte Cacheeffekte auftreten.

Bei den 40 festen Ausschnitten wird der `edgemaxlimit` Parameter von `getArea` auf unendlich gesetzt, damit die zurückgelieferte Kantenanzahl monoton wächst. Bei den 1 000 zufälligen Ausschnitten wird wie vorher `edgeminlimit` auf 100 000 und `edgemaxlimit` auf 400 000 Kanten gesetzt.

Die Abbildung 26 zeigt, wie lange die zufälligen Ausschnitte durchschnittlich brauchen. Zusätzlich ist die durchschnittliche Anzahl der Kanten im Bereich eingetragen. Am Anfang erkennt man die Sprünge, die durch Einfügen neuer Kanten in den zufälligen Ausschnitten entstehen: erreichen die Kanten aus dem R-Tree der vorletzte z-Ebene die `edgeminlimit` Grenze, wird die letzte Ebene nicht mehr betrachtet und die notwendige Rechenzeit eingespart.

Abbildung 26: Antwortzeit von 1000 zufälligen quadratischen Ausschnitten

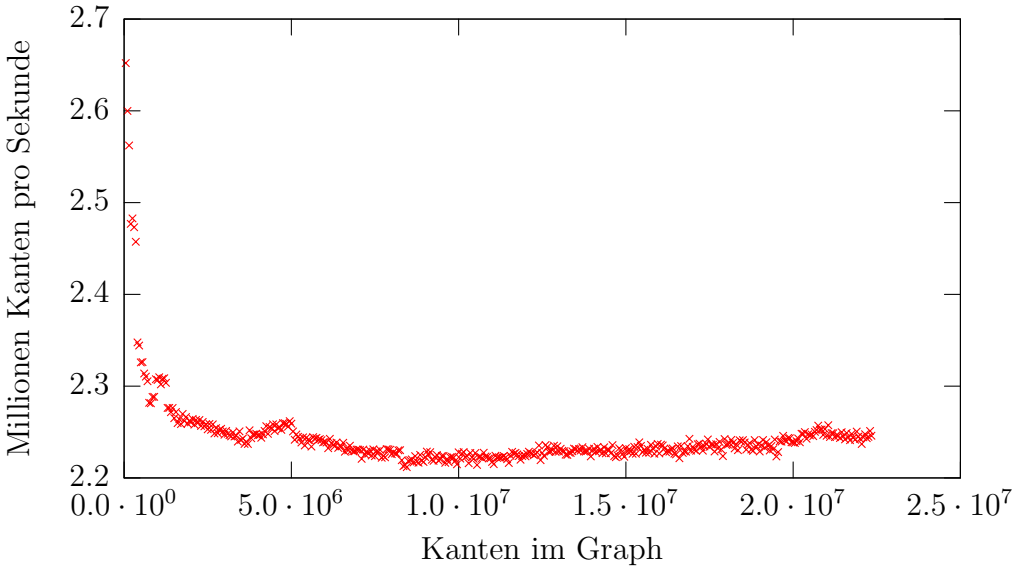


Trägt man den Quotient aus Kanten und Anfragezeit wie in Abbildung 27 auf, so erhält man das folgende Diagramm. Der Quotient entspricht der Steigung der linearen Kurve im ersten Diagramm, also der Gesamteffizienz der Indexstruktur. Am Anfang mit wenigen Kanten wäre ein lineares Suchen im Kantenarray schneller. Schon ab 100 000 Kanten wird der R-Tree schneller und immer effizienter. Ab 10^7 Kanten fällt die Effizienz nur noch sehr langsam ab.

Die letzten vier Messdiagramme in Abbildung 28 zeigen die durchschnittliche Anfragezeit in den vier festen Bereiche.

Im Intervall zwischen 0 und etwa $6 \cdot 10^6$ sind im Graph der kleinen Karlsruher Bereiche noch keine Kanten eingetragen, und folglich liefert die Anfrage ein leeres Ergebnis. Interessant ist die Anfragezeit dennoch, denn sie zeigt wie effizient solche leeren Ausschnitte bearbeitet werden: selbst bei einer Millionen Kanten im R-Tree wird noch kein großer Anstieg der Bearbeitungszeit gemessen. Dann werden die Kanten im Bereich Karlsruhe hinzugefügt, und die Bearbeitungszeit steigt an. Doch nach diesem Anstieg folgt praktisch keine Veränderung der Anfragezeit, obwohl mehr als zehn Millionen neue Rechtecke im R-Tree eingetragen werden.

Abbildung 27: Durchschnittliche Anfragegeschwindigkeit von 1000 zufälligen Ausschnitten

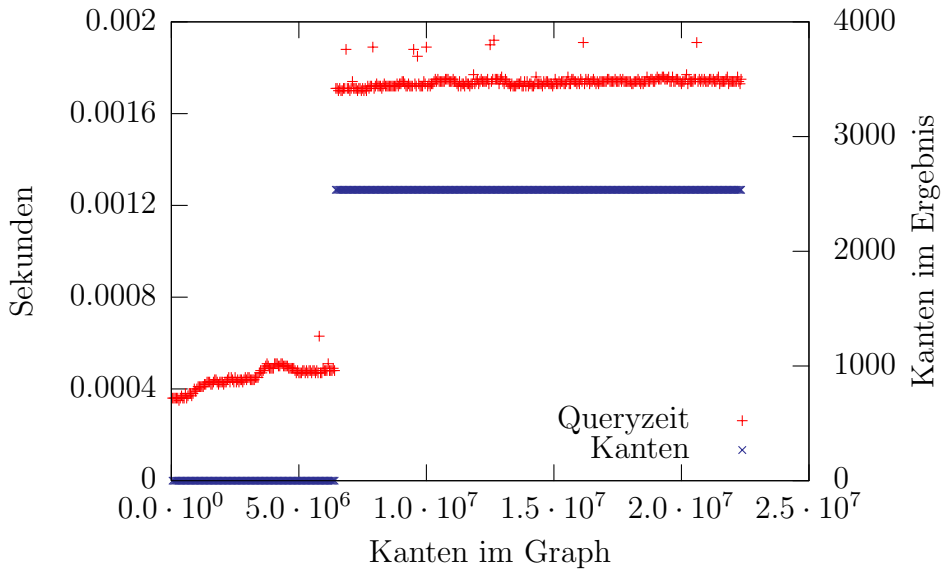


Für die beiden größeren Ausschnitte gibt es einen Bereich in dem die Kanten eingetragen werden: dort steigt die gelieferte Kantenanzahl pro Schritt um etwa 50 000. Weiter rechts sieht man ebenfalls noch einen geringen Anstieg der Kantenanzahl, da noch Kanten andere Länder nachgetragen werden. Ebenso wie bei den Karlsruhe-Ausschnitten sieht man keine Verlangsamung innerhalb der Intervalle, in denen keine neuen Kanten im Ausschnitt eingetragen werden.

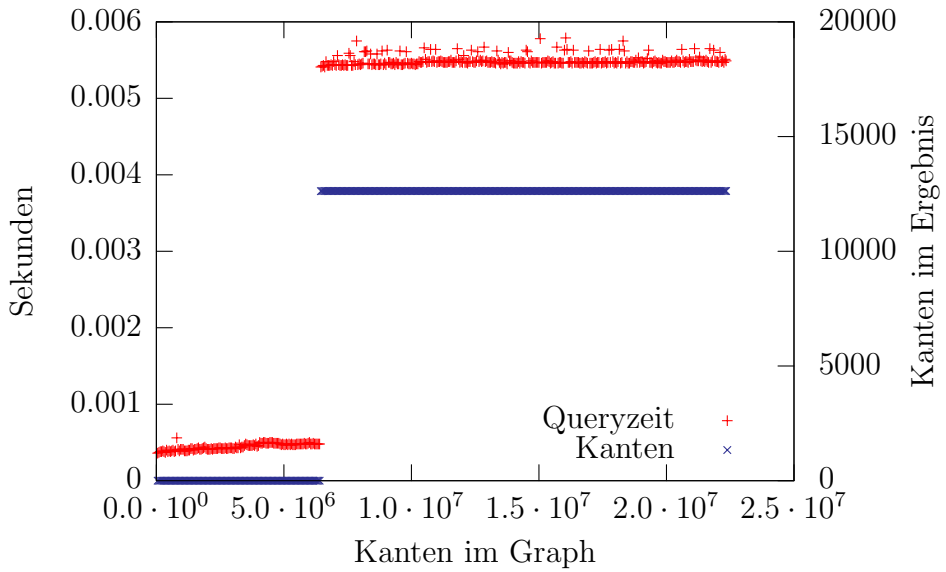
Diese letzten Versuche zeigen, wie effizient die R-Tree Indexstruktur kleine und große Ausschnittsanfragen beantworten kann. Die Antwortzeit wird praktisch nicht länger, wenn mehr als zehn Millionen weitere Kante eingefügt werden.

Abbildung 28: Antwortzeit von vier festen Ausschnitten bei schrittweisem Aufbau

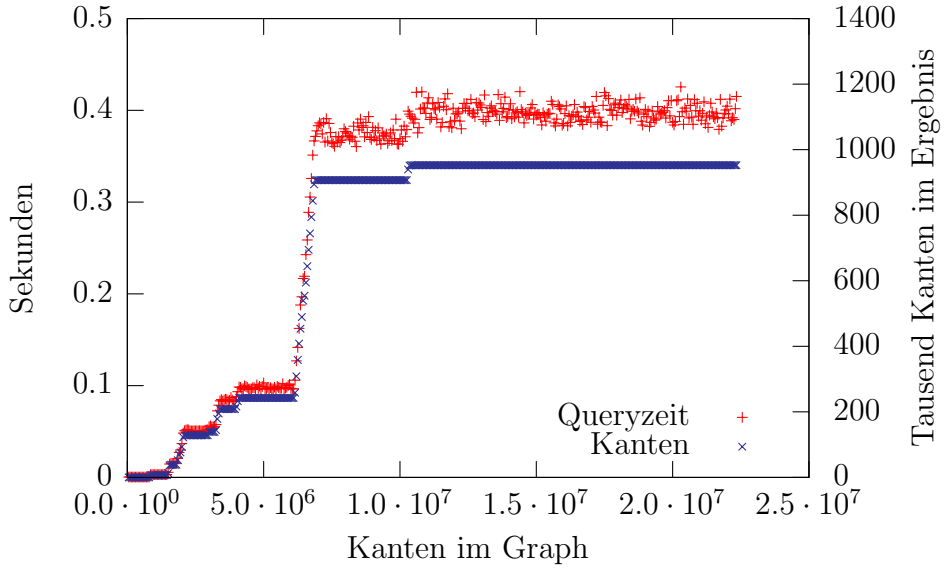
(a) Ausschnitt Karlsruhe Stadt



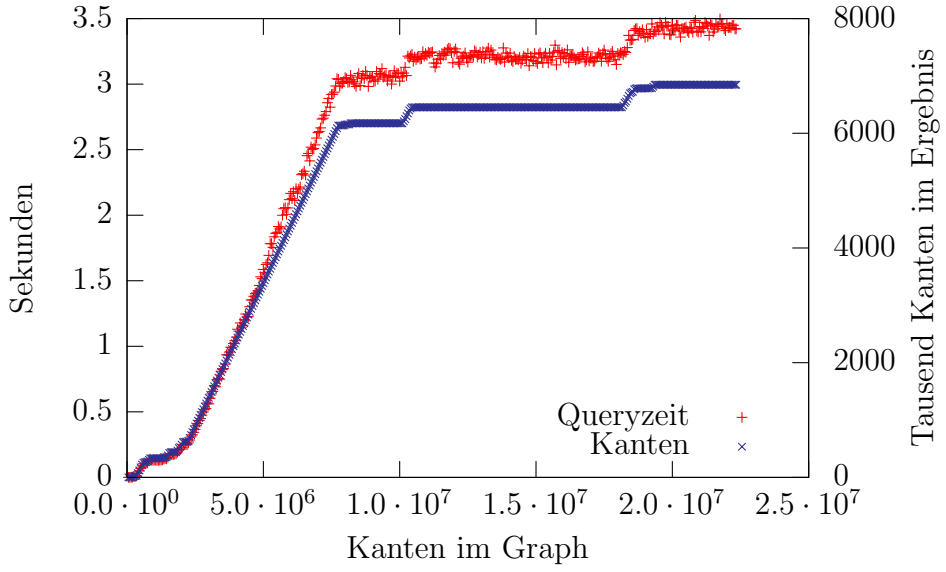
(b) Ausschnitt Karlsruhe Umland



(c) Ausschnitt Baden-Württemberg



(d) Ausschnitt Deutschland



5 Abschließende Bemerkungen

5.1 Schlussfolgerungen

In dieser Studienarbeit entstand ein Client/Server Rahmenwerk, das komfortable und schnelle Visualisierung von sehr großen Graph ermöglicht, bei denen die Koordinaten durch Attributwerte vorgegeben sind. Weitere Attribute können beeinflussen, wie eine Kante gezeichnet wird. Der Grundgraph und die Datenhaltung der Attributwerte wurde auf möglichst kompakte Art und Weise mit einer Adjazenz-Array Datenstruktur realisiert. Um die Geschwindigkeit einer Ausschnittsanfrage zu beschleunigen, wurde die mehrdimensionale Indexstruktur R-Tree verwendet. Diese ermöglichte Anfragezeiten, die linear zur Anzahl der zurückgelieferten Kanten und unabhängig vom gewählten Ausschnitt sind. Durch Aufzeichnung von temporären Änderungen im Graph durch eine Applikation wurde eine flexible Animation der Berechnungsergebnisse von Graphalgorithmen ermöglicht. Die gesamte Graphdatenhaltung-Bibliothek wurde so ausgelegt, dass bestehende Anwendungen problemlos integriert und visualisiert werden können.

5.2 Mögliche Erweiterungen

Die in der Datenhaltung verwendete R-Tree Indexstruktur wird in der Implementierung schrittweise aufgebaut: jede Kante wird einzeln in den Baum eingefügt. Dieses Vorgehen ist wenig effizient, wenn alle Daten auf einmal vorliegen und eingetragen werden sollen. Hier sind *r-tree packing* Algorithmen wie [Leut97] besser geeignet, welche einen neuen R-Tree durch bottom-up Gruppieren von Rechtecken erstellen. Dennoch sind weiterhin Funktionen notwendig, um einzelne Rechtecke einzufügen und wieder zu löschen. Diese Funktionalität wird benötigt, um die Indexstruktur bei Änderungen synchron mit den Graphdaten zu halten.

Die gesamte Graph-Bibliothek ist allgemein gehalten, so dass auch andere Graphen mit zweidimensionalen Knoten-Koordinaten visualisiert werden können. Die Referenzanwendung des Lehrstuhls für Theoretische Informatik behandelt zur Routenplanung Graphen, bei denen die Knotenkoordinaten in Längen- und Breitengraden angegeben sind. Diese Koordinaten werden im Java Client direkt als Zeichenkoordinaten benutzt. Längen- und Breitengrade sind aber Kugelkoordinaten und sollten durch verschiedene Projektionsfunktionen auf der Ebene abgebildet werden. In der Kartographie werden wegen der unvermeidlichen Verzerrungen eine Vielzahl von Kugel-Projektionen verwendet, wie z.B. Azimutalprojektionen (Zentral- oder Parallelprojektion), Kegelprojektionen oder Zylinderprojektionen (Mercator-Projektion). Sowohl Datenhaltungs-Server als auch Java Client können angepasst werden, um diese Projektionen zu realisieren.

A Anhang

A.1 Beispielausgabe im Fig-Format

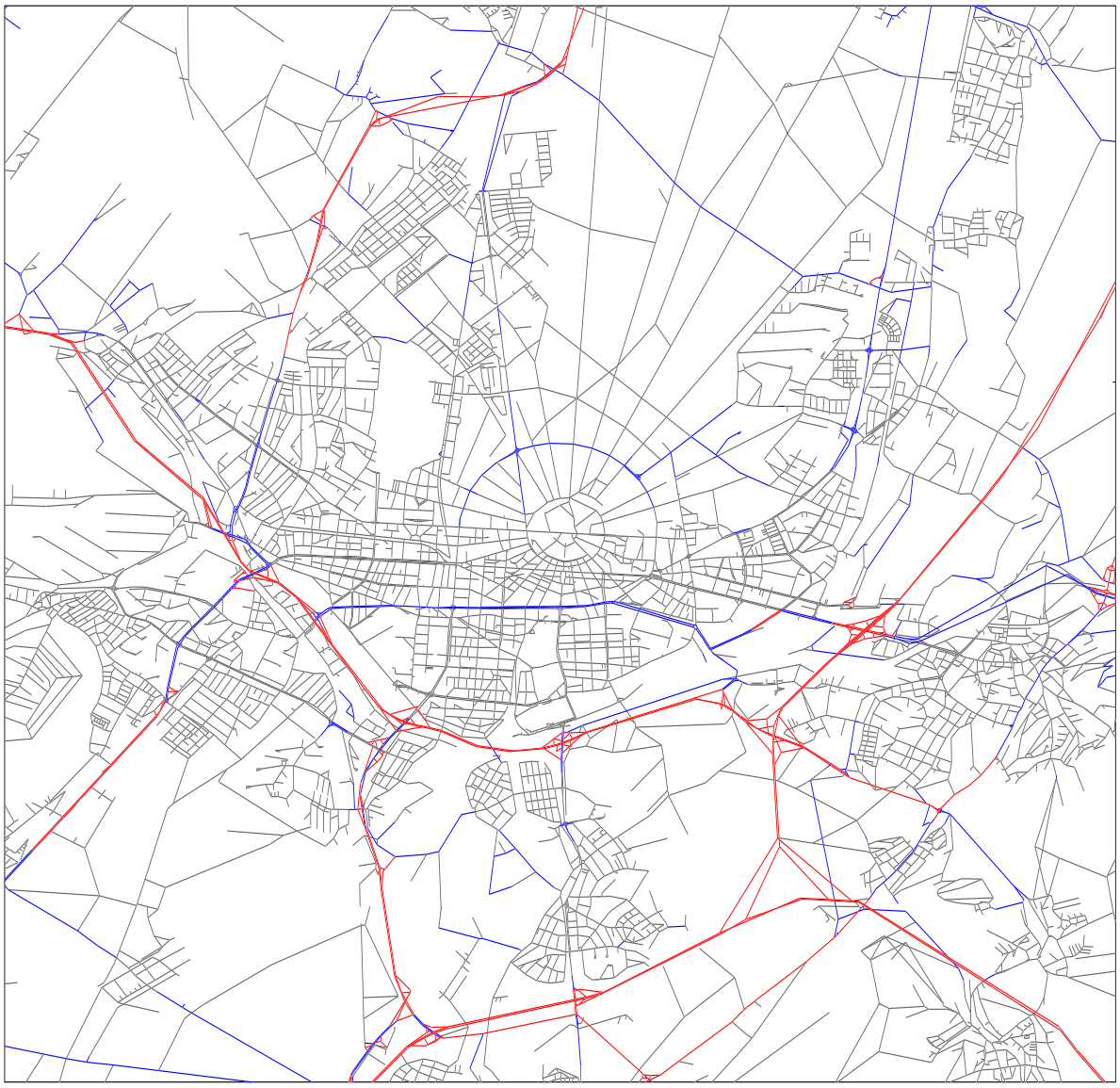


Abbildung 29: Beispiel einer Fig-Ausgabe von Karlsruhe

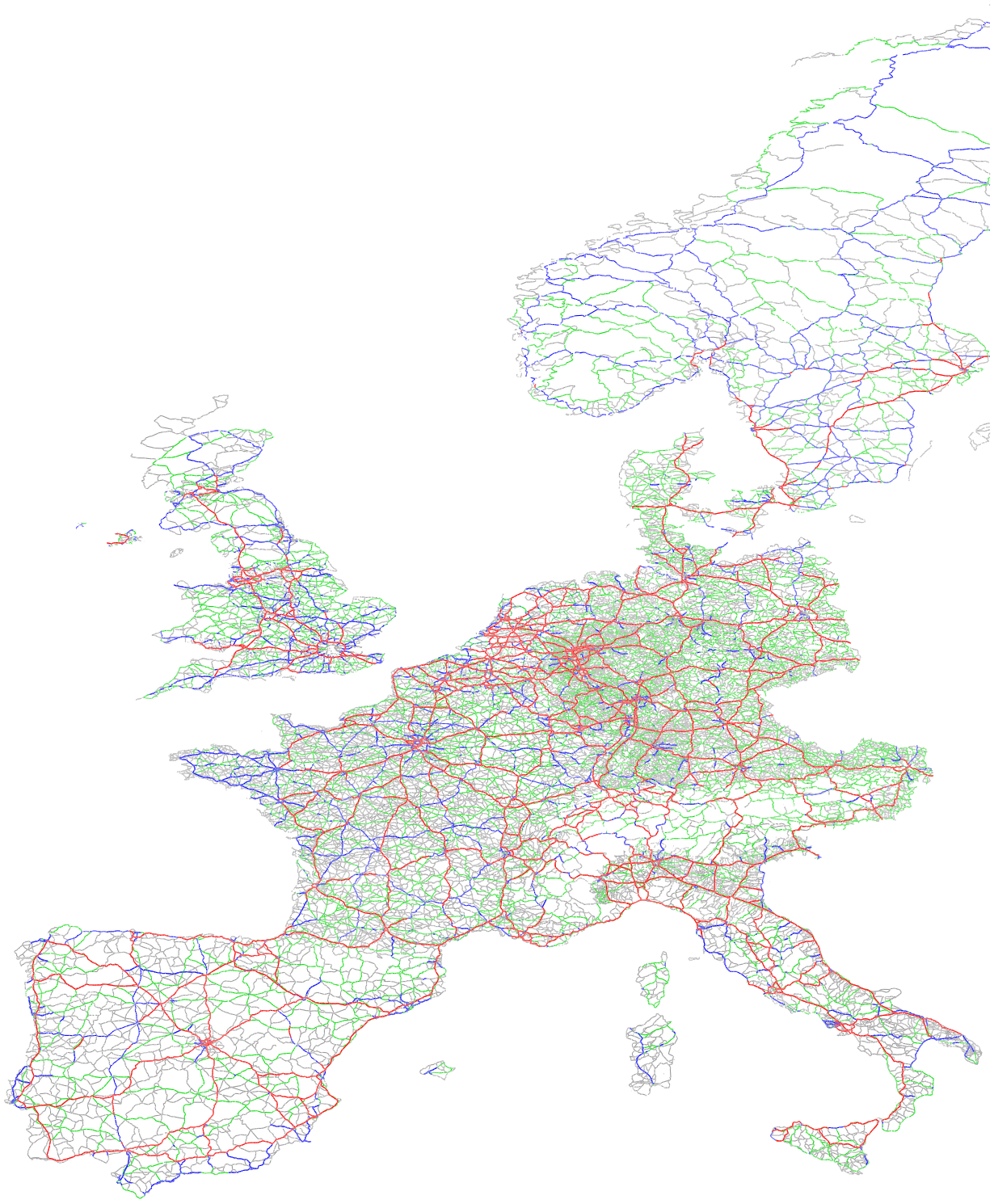


Abbildung 30: Beispiel einer Fig-Ausgabe von Europa

A.2 Konvertierungstabellen

from \ to	bool	char	byte	short	word	integer	dword
bool	=	>	>	>	>	>	>
char	(val != 0)	=	unsigned cast	>	>	>	>
byte	(val != 0)	signed cast	=	>	>	>	>
short	(val != 0)	limits	limits	=	unsigned cast	>	>
word	(val != 0)	limits	limits	signed cast	=	>	>
integer	(val != 0)	limits	limits	limits	limits	=	unsigned cast
dword	(val != 0)	limits	limits	limits	limits	signed cast	=
long	(val != 0)	limits	limits	limits	limits	limits	limits
qword	(val != 0)	limits	limits	limits	limits	limits	limits
float	(val != 0.0)	floor cast	floor cast	floor cast	floor cast	floor cast	floor cast
double	(val != 0.0)	floor cast	floor cast	floor cast	floor cast	floor cast	floor cast
string	match	strtol, limits	strtoul, limits	strtol, limits	strtoul, limits	strtol, limits	strtoul, limits
longstring	match	strtol, limits	strtoul, limits	strtol, limits	strtoul, limits	strtol, limits	strtoul, limits

from \ to	long	qword	float	double	string	longstring
bool	>	>	>	>	“false”/”true”	“false”/”true”
char	>	>	>	>	stringify	stringify
byte	>	>	>	>	stringify	stringify
short	>	>	>	>	stringify	stringify
word	>	>	>	>	stringify	stringify
integer	>	>	>	>	stringify	stringify
dword	>	>	>	>	stringify	stringify
long	=	unsigned cast	>	>	stringify	stringify
qword	signed cast	=	>	>	stringify	stringify
float	floor cast	floor cast	=	>	sprintf %.2f	sprintf %.2f
double	floor cast	floor cast	lose precision	=	sprintf %.2f	sprintf %.2f
string	strtoll, limits	strtoull, limits	strtod	strtod	=	>=
longstring	strtoll, limits	strtoull, limits	strtod	strtod	shortend, when saved	=

Tabelle 3: AnyType Konvertierungstabelle

Die Tabelle 3 beschreibt, wie die Methoden der `AnyType` Klasse Werte eines Datentyps in einen anderen konvertieren. Es folgt eine genauere Erklärungen der Konvertierungskürzel:

= Gleichheit der Typen

> Der rechte Typ ist größer als der linke und daher ist eine Konvertierung verlustfrei.

(val != 0) Der Wert wird wie in C++ mit 0 vergleichen. Das Ergebnis ist der boolsche Wert.

match Bei der String-auf-Boolean-Konvertierung werden folgende Strings zu *false* konvertiert: „0“, „f“, „false“, „n“, „no“. Folgende Strings konvertieren zu *true*: „1“, „t“, „true“, „y“, „yes“. Alle anderen Strings ergeben eine `ConversionException`.

signed cast Es wird von der Hardware von unsigned nach signed gecasted. Hierbei werden große positive Werte zu negative Werte.

unsigned cast Es wird von der Hardware von signed nach unsigned gecasted. Hierbei werden negative Werte zu großen unsigned Werten.

limits Die Eingabegröße wird mit dem verfügbaren Zahlenraum des Ausgabetyps verglichen. Überschreitet die Eingabe eine der Grenzen, so wird diese Grenze gesetzt. Beispielsweise wird (1000 `cast char`) = 128.

floor cast Von der Hardware werden Gleitkommazahlen auf den größten Ganzzahlwert kleiner gleich der Gleitkommazahl gesetzt.

strtol, limits Um Strings in Ganzzahlen zu konvertieren werden die jeweiligen Standard-Bibliotheksfunktionen (`strtol`, `strtoul`, `strtoll`, `strtoull`) verwendet und anschließend die `limits` Prüfung angewandt. Kann der String nicht vollständig als Zahl gelesen werden, tritt eine `ConversionException` auf.

strtod String auf Gleitkommazahlen werden mit der Standard-Bibliotheksfunktionen `strtod` konvertiert.

lose precision Hardware Cast von `double` nach `float`.

"false"/"true" Ein Boolean wird als der String „false“ oder „true“ dargestellt.

stringify Um Ganzzahlwerte in Strings umzuwandeln werden spezielle eingebaute Funktionen verwendet.

sprintf %.2f Gleitkommazahlen werden mit der Standard-Bibliotheksfunktionen `sprintf` mit 2 Nachkommastellen dargestellt.

A.3 UML Diagramme

A.3.1 ChangeTimeline Klassen

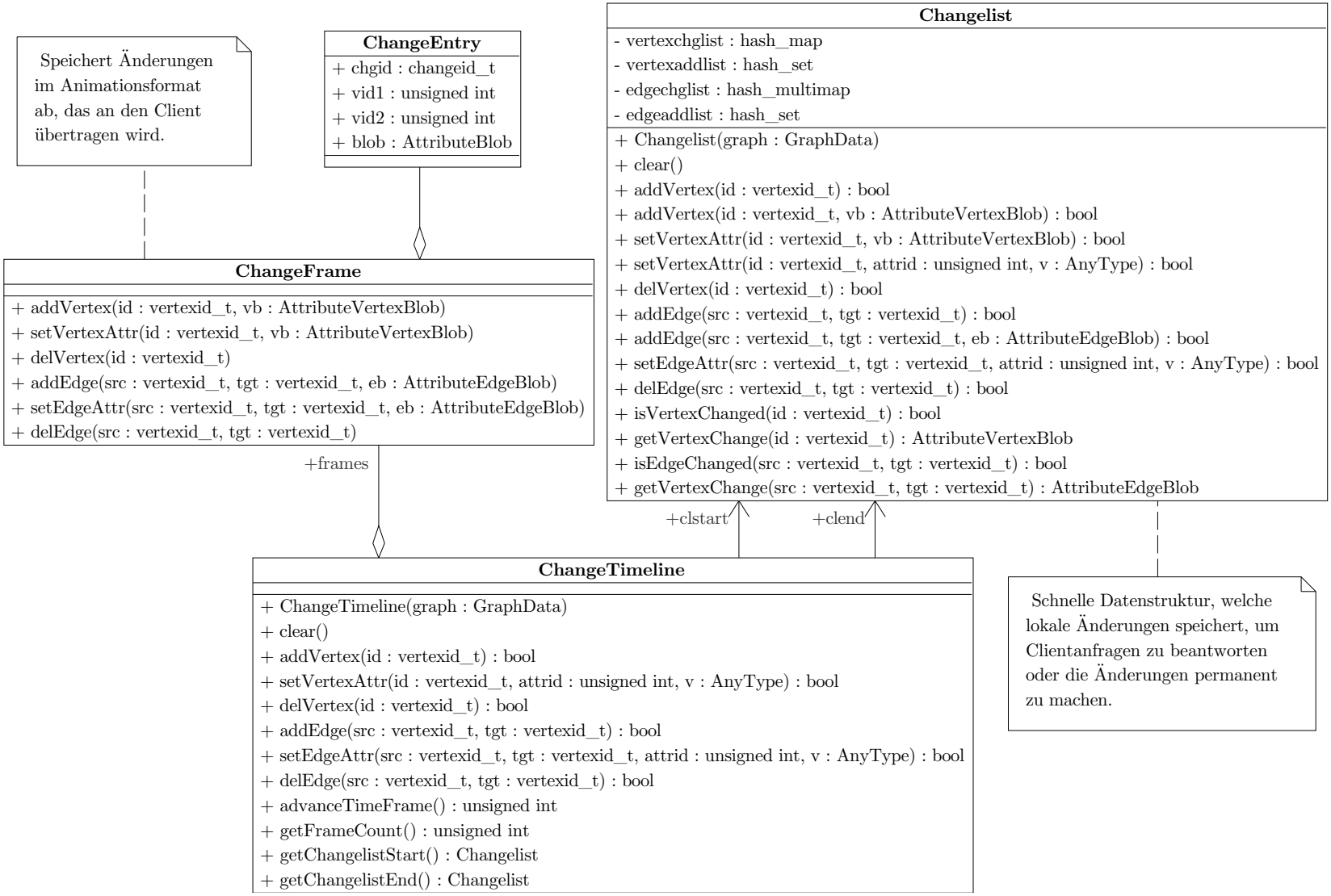


Abbildung 31: UML Diagramm des ChangeTimeline Klassenkomplex

A.3.2 GraphContainer Klassen

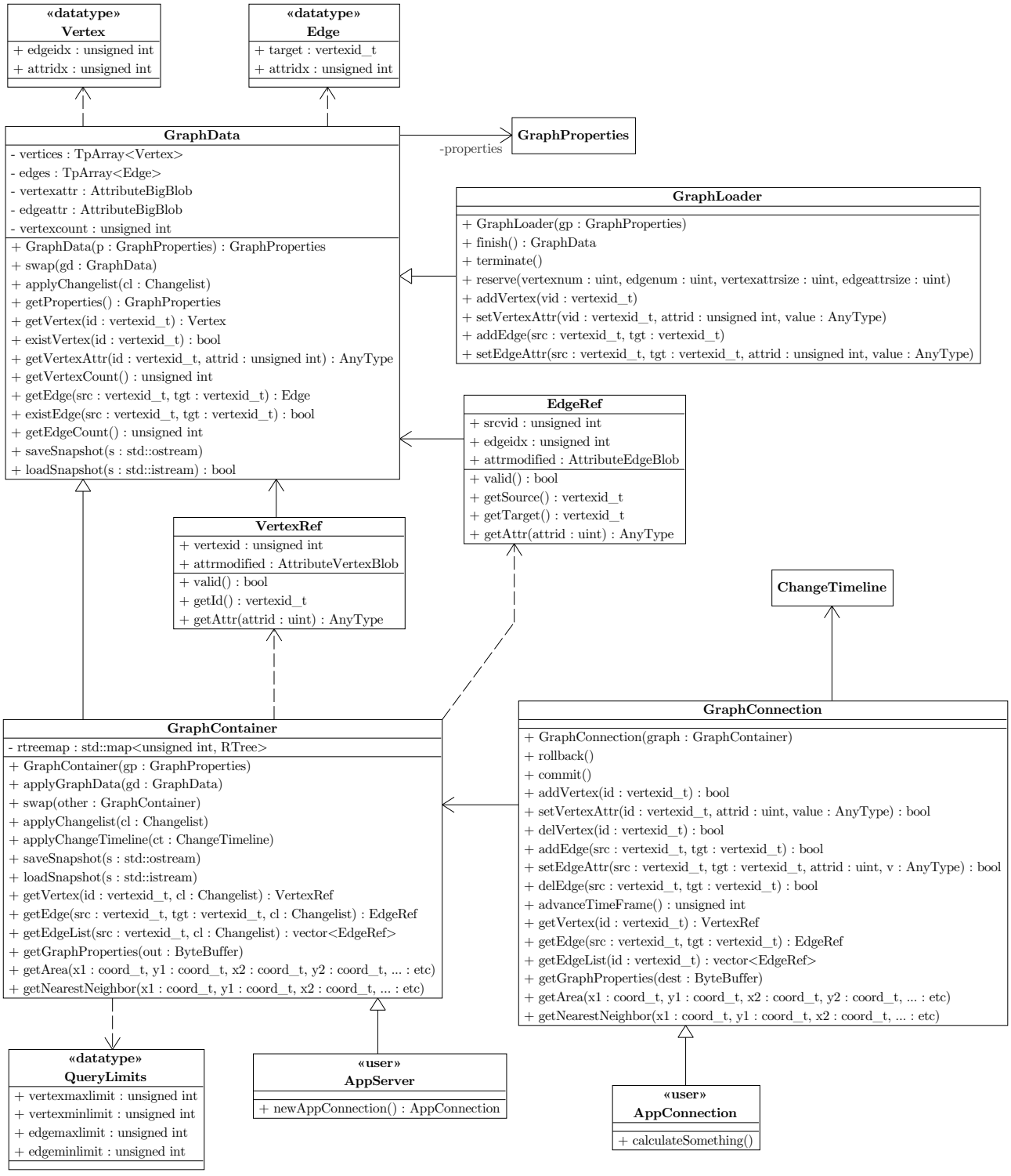


Abbildung 32: UML Diagramm der Datenhaltungsklassen

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Cor01] T. Cormen, C. Leiserson and R. Rivest: *Introduction to Algorithms*, second edition, MIT Press, 2001

[Spirit] Boost.Spirit C++ Parser Library: http://spirit.sourceforge.net/

[Gae98] V. Gaede and O. Günther, *Multidimensional access methods*, ACM Computing Surveys, vol 30, no. 2, pages 170-231, June 1998

[Gut84] A. Guttman: *R-Trees: A Dynamic Index Structure for Spatial Searching*. In Proc. ACM SIGMOD, pages 47-57, June 1984

[Beck90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger: *The R*-tree: An efficient and robust access method for points and rectangles*. In Proc. ACM SIGMOD, pages 322-331, 1990.

[Sam90] H. Samet, *The design and analysis of spatial data structures*. Addison-Wesley, 1990.

[Sel87] T. Sellis, N. Roussopoulos and C. Faloutsos: *The R^+-tree: A dynamic index for multi-dimensional objects*. In Proc. of VLDB, pages 3-11, 1987.

[Niev84] J. Nievergelt, H. Hinterberger and K. C. Sevcik: *The grid file: An adaptable symmetric multikey file structure*. ACM Transactions on Database Systems; ACM CR 8411-0931, 1984.

[FigF] Fig Format 3.2: http://www.xfig.org/userman/fig-format.html

[Rig01] P. Rigaux, M. Scholl, A. Voisard: *Spatial Databases: With Application to GIS*. Morgan Kaufmann Publishers, 2001.

[PgSQL] PostgreSQL Source: http://www.postgresql.org/

[RTree1] Marios Hadjieleftheriou: *Spatial Index Library*. http://u-foria.org/marioh/spatialindex/

[RTree2] Ondrej Pavlata: *Mg R-tree Library*. http://www.volny.cz/r-tree/

[Leut97] S. Leutenegger, M. López and J. Edgington: *STR: A Simple and Efficient Algorithm for R-Tree Packing*, Proc. 13th Int'l Conf. on Data Eng, 1997