Sächsisches Landesgymnasium Sankt Afra zu Meißen
Besondere Lernleistung im Fach Informatik

Darstellung von Algorithmen und Datenstrukturen für die Entwicklung und effiziente Implementierung eines Routingalgorithmus für Verkehrsnetze

Jonas Witt

eingereicht am 14. Januar 2005

betreut von

Dr. Dieter Sosna (Universität Leipzig, Institut für Informatik)
Ralf Böttcher (Fachlehrer Informatik am SLG St. Afra)

Inhaltsverzeichnis

1. Einleitung

1.1. Gegenstand und Ziel der Arbeit

Im Rahmen der Arbeit zur BeLL habe ich mich seit einem Jahr mit der Entwicklung eines Fahrplanauskunftssystems für das Leipziger Straßenbahnnetz beschäftigt. Dabei trat an vielen Stellen das Problem der ungenügenden Ausführungsgeschwindigkeit meines Programms auf, dessen Lösung auf der Softwareseite mich besonders interessierte.

Um ein Problem mit Hilfe der Informatik praktisch zu lösen, sind zwei Schritte notwendig. Der erste Schritt umfasst die Formulierung des Problems der realen Welt als mathematisches Modell. Dabei werden Tätigkeiten als Algorithmen und reale Objekte durch mathematische Objekte beschrieben. Die mathematischen Objekte sind nur abstrakt beschrieben, eine Implementierung erfolgt in diesem Schritt noch nicht; so arbeitet ein Algorithmus z.B. mit dem Modell einer Menge von Objekten, die später als Array, Liste oder Baum implementiert werden können. Die Operationen auf jeder dieser Implementierungen sind identisch (ein Objekt einfügen, ein Objekt entfernen, prüfen ob ein Objekt enthalten ist, prüfen ob die Menge leer ist), deshalb kann ein Algorithmus im ersten Schritt unabhängig von einer bestimmten Implementierung formuliert werden. Im zweiten Schritt wird der im ersten Schritt formulierte Algorithmus implementiert. Dabei werden dann geeignete Implementierungen der verwendeten Datenstrukturen ausgewählt.

Diese Arbeit beschreibt, wie sowohl bei der Formulierung des Algorithmus als auch bei der Auswahl der Datenstruktur bedeutende Geschwindigkeitsgewinne erzielt werden können. Bei der Anfertigung der Arbeit erhielt ich wertvolle Unterstützung von meinen Betreuern, bei denen ich mich auf diesem Wege bedanken möchte.

1.2. Notationen

Zur Notation der Algorithmen wird ein Pseudo-Code verwendet, der zur besseren Lesbarkeit auf die Darstellung implementationsspezifischer Details verzichtet. Zuweisungen geschehen mittels $\leftarrow$, $A \leftarrow B$ bedeutet also eine Zuweisung von B nach A (Wie auch in [1], Abschnitt 1.1 verwendet). Operationen mit Mengen werden mit den in der Mathematik üblichen Symbolen dargestellt.

Für die Behandlung und den Vergleich von Algorithmen muss man beschreiben, wieviel Zeit der Algorithmus benötigt, um eine bestimmte Menge an Arbeit zu verrichten (also die Laufzeit in Abhängigkeit von der Eingabegröße). Dies geschieht in der Informatik durch Landau-Symbole ("Groß-O-Notation"), die auch hier verwendet werden. Eine Definition findet sich in [4].

2. Algorithmen

Die Fragestellung "Wie komme ich am schnellsten von A nach B?" wird in der Informatik *Routing* bzw. genauer *single source shortest path problem* genannt und ist das Problem des Findens einer Folge von Kanten in einem Graphen[1], die von einem Knoten A zu einem (anderen) Knoten B führen, so dass die Summe der Gewichte der Kanten möglichst klein ist.[2]

Für die Lösung dieses Problems existieren drei bekannte Algorithmen:

Dijkstra-Algorithmus Löst das *shortest path problem* für Graphen mit *nicht-negativen Kantengewichten.*

Bellman-Ford-Algorithmus Löst das *shortest path problem* auch für Graphen mit *negativen Kantengewichten*, wenn keine Zyklen mit negativen Gewichten (also geschlossene Folgen von Kanten, für die die Summe der Gewichte negativ ist) auftreten.

A*-Algorithmus Löst das *shortest path problem* unter Verwendung eines heuristischen Ansatzes. Das heißt, dass er zur Auswahl des nächsten Knotens zusätzlich eine Schätzfunktion verwendet, die für jeden Knoten K angibt, wie nah K *etwa* am Zielknoten ist. Je genauer diese Schätzfunktion ist, desto schneller ist der Algorithmus.

Neben dem single source shortest path problem, welches die kürzeste Verbindung von *einem* Knoten zu einem oder allen anderen ermittelt, existiert auch das *all-pairs shortest path problem*, welches die kürzeste Verbindung von *allen* Knoten zu *allen anderen* Knoten ermittelt. [5]

[1] Ein Graph hat die Form eines Netzwerkes, in dem Knoten von Kanten verbunden werden. Siehe auch Abschnitt 3.3 auf Seite 19.

[2] Zum Zweck der besseren Unterscheidbarkeit werden für Knoten oder Linien große (K), für Kanten kleine kursive (v) und für Mengen große Kalligraphiebuchstaben ($\mathcal{M}$) verwendet.

Da hier ein Verkehrsnetz betrachtet werden soll, in dem das Gewicht einer Kante die für das Befahren der Kante benötigte Zeit darstellt, ist eine Beachtung von negativen Kantengewichten nicht notwendig. Ebenso wird eine Benutzeranfrage immer einen bestimmten Startpunkt und einen bestimmten Zielpunkt beinhalten, so dass lediglich das single source shortest path problem zu betrachten ist.

2.1. Der Dijkstra-Algorithmus

Der Dijkstra-Algorithmus (siehe Pseudo-Code in Algorithmus 1) ermittelt in einem Graphen die kürzeste Verbindung von einem Startpunkt A zu einem Zielpunkt B. Er speichert für jeden Knoten K dessen Gesamtentfernung $d(K)$ zum Startpunkt A sowie den Knoten $p(K)$, von dem aus er besucht wurde. Damit kann nach der Termination des Algorithmus durch schrittweises Rückwärtsverfolgen der jeweiligen Vorgängerknoten $p(K)$ von B aus der beste Pfad durch den Graphen abgelesen werden. Anfangs ist $d(A) = 0$ und $d(K_i) = \infty$ für alle anderen Knoten K_i (dies repräsentiert, dass bis jetzt kein Weg dahin existiert). Generell kann $d(K)$ für jeden Knoten K bestimmt werden, indem zu $d(L)$ eines besuchten Nachbarknotens L das Kantengewicht zwischen beiden ($v(K, L)$) addiert wird. In jedem Fall muss überprüft werden, ob die neue Entfernung $d(L) + v(K, L)$ geringer ist als die alte Entfernung $d(K)$ oder nicht. Im ersten Fall wird $d(K) \leftarrow d(L) + v(K, L)$ und $p(K) \leftarrow L$, im zweiten Fall geschieht keine Änderung, da diese Kante zu keiner Verbesserung führt.

Des Weiteren unterscheidet der Algorithmus zwei Mengen von Knoten: Die der schon besuchten Knoten ($\mathcal{S}$) und die der noch nicht besuchten Knoten ($\mathcal{Q}$). Anfangs befinden sich alle Knoten in der Menge $\mathcal{Q}$. In jedem Schritt wird der Knoten K_{min} aus $\mathcal{Q}$ ausgewählt, für den $d(K)$ minimal ist (siehe Zeile 9 in Algorithmus 1; Betrachtung dieser Funktion im nächsten Abschnitt). K wird aus $\mathcal{Q}$ nach $\mathcal{S}$ verschoben und für alle Knoten L_i, die durch eine Kante $v(K_{min}, L_i)$von K_{min} aus zu erreichen sind, wird $d(K_{min}) + v(K_{min}, L_i)$ berechnet und $d(L_i)$ dementsprechend aktualisiert, wenn $d(K_{min}) + v(K_{min}, L_i) < d(L_i)$ (s.o.).

Wenn $d(B) < \infty$ (B erreichbar ist) und $d(K_{min}) > d(B)$ (B mit positiven Kantengewichten nicht mehr schneller erreicht werden kann) oder $\mathcal{Q} = \{\}$ (keine Knoten mehr zu besuchen sind) gilt, terminiert der Algorithmus. Es existiert kein Weg von A nach B, wenn $B \in \mathcal{Q}$ und $d(k_{min}) = \infty$ (kein weiterer Knoten in $\mathcal{Q}$

besucht werden kann).

Algorithmus 1: Der Dijkstra-Algorithmus
Eingabe: Ein Graph $G(\mathcal{V}, \mathcal{E})$, 2 Knoten A und B aus $\mathcal{V}$.
Ausgabe: Der beste Pfad zwischen A und B.

```
DIJKSTRA(G(V,E), A, B)
(1)   foreach V_i ∈ V {
(2)     d(V_i) ← ∞;
(3)     p(V_i) ← null;
(4)   }
(5)   d(A) ← 0;
(6)   S ← {};
(7)   Q ← V;
(8)   while Q ≠ {} and min(d(x), x ∈ Q) < d(B) {
(9)     K ← min(x, x ∈ Q, d(x));
(10)    Q ← Q\K;
(11)    S ← S ∪ K;
(12)    foreach E(K, L_i) ∈ E {
(13)      if d(K) + v_E(K, L_i) < d(L_i) {
(14)        d(L_i) ← d(K) + v_E(K, L_i);
(15)        p(L_i) ← K;
(16)      }
(17)    }
(18)  }
```

2.1.1. Optimierung

Die Auswahl des nächsten Knotens (Algorithmus 1, Zeile 9) kann unterschiedlich implementiert werden. Zum einen kann durch schrittweises Durchsuchen der nächstliegende Knoten (für den $d(K)$ minimal ist) aus den noch nicht besuchten ausgewählt werden. Diese Methode muss jedoch zur Ermittlung des nächsten Knotens die gesamte Menge $\mathcal{Q}$ durchlaufen, sie hat also eine lineare Laufzeit ($O(N)$).

Speziell für die Aufgabe des schnellen Zurückgebens des kleinsten/größten Elements einer Menge ist die Datenstruktur *Heap* entwickelt worden. Ein Heap organisiert die Elemente einer Menge so, dass er ein bestimmtes Element (häufig das kleinste/größte) schnell zurückgeben kann. Eine einfache Implementierung ist der binäre Heap, der auf einem binären Baum (siehe Abschnitt 3.2.1) basiert. Aufgrund der Eigenschaften des binären Baums werden für das Einfügen

und Entnehmen eines Elements jeweils $O(logN)$ benötigt. Bei der oben beschriebenen linearen Methode muss keine Einfügeoperation durchgeführt werden, da durch die Menge $\mathcal{Q}$ der nicht besuchten Knoten bereits die zu durchsuchende Menge gegeben ist. Wenn, wie im Falle eines Heaps aber jeder Knoten, nachdem die Entfernung zu ihm berechnet wurde, in die Datenstruktur eingefügt werden muss, muss dies hinter Zeile 13 (Algorithmus 1) geschehen.

Letztendlich kann so mit einem binären Heap eine Steigerung von $O(N)$ auf $O(logN)$ (jeweils für Einfügen und Entnehmen) erreicht werden, in Abschnitt 4.3.1 wird dies mit Messwerten belegt. Zur Implementierung des binären Heaps sei auf den Quelltext (Klasse `BinaryHeap.java`) verwiesen; eine genauere Beschreibung sowie Verweise auf andere Arten von Heaps findet sich unter anderem unter [9].

2.1.2. Laufzeit

Die Hauptschleife des Algorithmus (Zeile 8) wird im ungünstigsten Fall $\mid \mathcal{Q} \mid$ mal durchlaufen. Darin wird der nächste Knoten ausgewählt und zu allen Nachbarknoten eine Verbindung aufgebaut. Die Auswahl des nächsten Knotens ist abhängig von der Implementierung des Heaps, die Anzahl der Nachbarknoten ist immer linear abhängig von $\mid \mathcal{E} \mid$. Wenn für das Ermitteln des nächsten Knotens ein lineares Durchsuchen von $\mathcal{Q}$ benutzt wird, läuft der Vorgang der Knotenauswahl in $O(\mathcal{V})$. Wird ein binärer Heap verwendet, sinkt diese Zeit auf $O(\log \mathcal{V})$, die allerdings zwei mal anfällt, da die Knoten auch in den Heap eingefügt werden müssen.

Die Gesamtlaufzeit ist mit linearem Durchsuchen $\mathcal{V}(\mathcal{V} + \mathcal{E})$, was $O(\mathcal{V}^2)$ entspricht, da laut Definition der Groß-O-Funktion lediglich die signifikantesten Teile angegeben werden. Mit binärem Heap ist die benötigte Zeit $O(V(\log V + E))$.

2.2. Modifikation des Dijkstra-Algorithmus

Der Dijkstra-Algorithmus kann in der bekannten und eben vorgestellten Form nicht für das Routing in Straßenbahnnetzen benutzt werden, da zur kompletten Modellierung (der erste Schritt des in Abschnitt 1.1 beschriebenen Prozesses) eines Straßenbahnnetzes zusätzlich zwei Aspekte berücksichtigt werden müssen.

- Die Wartezeit auf die nächste Bahn ist dynamisch. D.h., dass eine Kante im Graphen G nicht mehr nur ein statisches Gewicht, sondern zusätzlich einen dynamischen Anteil hat, der die Wartezeit an der Haltestelle darstellt.
- In einem Straßenbahnnetz verkehren Linien. Bei der Auswahl des nächsten Knotens muss der Algorithmus also Knoten bevorzugen, die ohne Wechsel der Bahn zu erreichen sind, da hier keine Wartezeit anfällt. Als elementarer Bestandteil des Graphen sind diese Linien auch in der Definition des Graphen enthalten: Statt auf dem normalen Graphen $G(\mathcal{V}, \mathcal{E})$ muss der veränderte Algorithmus auf $G(\mathcal{V}, \mathcal{E}, \mathcal{L})$ operieren.

2.2.1. Dynamische Kantengewichte

Das Gewicht der Kante $E(M, N)$ zwischen zwei Knoten M und N setzt sich aus zwei Teilen zusammen:

- aus einem statischen Teil, der bereits im normalen Dijkstra-Algorithmus vorhanden war und direkt am Graphen abgelesen werden kann ($v_{fest}(M, N)$)
- aus einem dynamischen Teil, da eine Linie eine Kante nur zu diskreten Zeiten befährt.

Der dynamische Teil ist die Zeitspanne von der aktuellen Zeit t bis eine Linie die gewünschte Kante befährt ($v_{dyn}(M, N, L, t)$).

2.2.2. Linien

Die Bestimmung von $v_{dyn}(M, N, t)$ beinhaltet die Bestimmung der Linie L, die von t an $E(M, N)$ als nächstes befährt ($minline(M, N, t)$, siehe Zeile 14 in Algorithmus 2). Dazu muss für jede Linie, die diese Kante befährt, berechnet werden, wann sie die Kante als nächstes befährt. Die Laufzeit dieser Operation ist von der Anzahl der Linien abhängig.

In den Netzdaten des Straßenbahnnetzes ist der Fahrplan einer Linie dadurch gegeben, dass die Zeiten, zu denen eine Straßenbahn an der Endhaltestelle losfährt, definiert sind. Durch den gegebenen Verlauf der Linie kann für jede Station durch Aufsummieren der statischen Kantengewichte von der Endhaltestelle bis

zur Haltestelle bestimmt werden, wie lange die Straßenbahn von der Endhaltestelle bis zu dieser Haltestelle braucht.

Der dynamische Anteil $v_{dyn}(M, N, t)$ kann bestimmt werden, indem die Entfernung s zur Endhaltestelle von der aktuellen Zeit t_0 abgezogen wird. Dann wird in der Liste der Startzeiten nach der nächsten Abfahrtszeit t_a nach $t_0 - s$ geschaut. Die Abfahrtszeit der Straßenbahn an der Haltestelle ist so $t_1 = t_a + s$. Damit ist $v_{dyn}(M, N, t) = t_1 - t_0$.

Algorithmus 2: Modifizierter Dijkstra-Algorithmus
Eingabe: Ein Graph $G(\mathcal{V}, \mathcal{E}, \mathcal{L})$, 2 Knoten A und B aus $\mathcal{V}$, die Zeit t.
Ausgabe: Der beste Pfad zwischen A und B.
DIJKSTRA$(G(\mathcal{V}, \mathcal{E}, \mathcal{L}), A, B, t)$

```
(1)   foreach V_i ∈ V {
(2)     d(V_i) ← ∞
(3)     p(V_i) ← null
(4)     l(V_i) ← null
(5)   }
(6)   d(A) ← 0
(7)   S ← {}
(8)   Q ← V
(9)   while Q ≠ {} and min(d(x), x ∈ Q) < d(B) {
(10)    M ← heapmin(x, x ∈ Q, d(x))
(11)    Q ← Q\M
(12)    S ← S ∪ M
(13)    foreach E(M, N_i) ∈ E {
(14)      L_min ← minline(M, N_i, t)
(15)      if d(M) + v_dyn(M, N_i, L_min, t) < d(N_i) {
(16)        heapinsert(N_i)
(17)        d(N_i) ← d(M) + v_dyn(M, N_i, L_min, t)
(18)        p(N_i) ← M
(19)      }
(20)    }
(21)  }
```

2.2.3. Laufzeit

Zu der in Abschnitt 2.1.2 beschriebenen Laufzeit kommt in Algorithmus 2 die Zeit zum Ermitteln der günstigsten Linie für eine Kante hinzu. Die optimale Linie wird durch lineare Suche ermittelt, also beträgt die Laufzeit $O(\mathcal{L})$.

Die Laufzeit des modifizierten Algorithmus (unter Verwendung von Adjazenzlisten und eines binären Heaps) auf dem Graphen $G(\mathcal{V}, \mathcal{E}, \mathcal{L})$ ist $O(\mathcal{V}(\log \mathcal{V} + \mathcal{E}\mathcal{L}))$.

3. Datenstrukturen

Für das zu entwickelnde Programm und in der Informatik allgemein spielt die Wahl der Datenstruktur zur Anordnung und Verknüpfung von Daten eine wesentliche Rolle. Verschiedene Datenstrukturen unterscheiden sich in der benötigten Zeit, die zum Ausführen elementarer Operationen auf den Daten notwendig ist, und dem zusätzlich benötigten Speicherplatz zur Verwaltung der Daten.

Im eben beschriebenen Algorithmus sind zwei Aufgaben zu erledigen:

- die Speicherung der Menge der Haltestellen
- die Speicherung des Verkehrsnetzes als Graph.

3.1. Mengen von Elementen

Eine Datenstruktur zur Speicherung von Elementen, die eine Menge von Objekten enthält, muss folgende Operationen durchführen können:

- auf einzelne Elemente zugreifen
- ein Element an einer Position einfügen
- ein Element löschen.

Des Weiteren sollte der zusätzliche Speicherplatzbedarf für die Organisation der Menge möglichst klein sein.

3.1.1. Arrays

Ein Array besteht aus einem zusammenhängenden Teil Speicher, dessen Größe

$$N \cdot GroesseElement$$

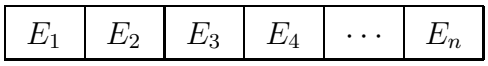

Abbildung 3.1.: Ein Array

ist (Abbildung 3.1). Die Größe (Anzahl der Elemente) eines Arrays ist konstant und muss bei der Initialisierung festgelegt werden, da der Speicher zusammenhängend alloziert[1] werden muss und im Nachhinein (durch die Speicherverwaltung aller Systeme) nicht vergrößert werden kann. Dadurch entsteht in bestimmten Fällen eine große Unflexibilität.

Der Zugriff auf ein Element eines Arrays benötigt eine konstant kleine Zeit ($O(1)$), da die Position des Elements im Speicher durch

$$Position = ArrayStart + (Index \cdot ElementGroesse)$$

gegeben ist. Zum Einfügen eines Elements ist ebenso eine konstante Zeit nötig, wenn die Position bekannt ist und sich kein Element an der Position befindet. Wenn sich bereits ein Element an dieser Stelle des Arrays befindet, müssen alle dahinter liegenden Elemente eine Position nach hinten verschoben werden, was abhängig von der Position einen Aufwand zwischen $O(1)$ und $O(N)$ bedeutet. Beim Verschieben nach hinten muss wieder beachtet werden, dass die Größe des Arrays fest ist. Ein Einfügen mit Verschieben ist also nur möglich, wenn am Ende des Arrays noch Platz ist.

Der Vorteil eines Arrays ist die vergleichsweise kompakte und sparsame Speicherung der Elemente. Für ein Array von z.B. 100 Integer-Zahlen wird lediglich das 100-fache der Größe eines Integers benötigt, da für die Navigation im Array keine weitere Information außer der Größe der Elemente und der Startposition im Speicher benötigt wird.

3.1.2. Verkettete Listen

Das Einfügen eines Elements in eine Menge kann ggf. erheblich beschleunigt werden, wenn als Datenstruktur eine *verkettete Liste* verwendet wird. Vor allem fällt zusätzlich das Problem der festen Größe (wie bei einem Array) weg. Eine verkettete Liste besteht aus separat allozierten Objekten der Struktur *(Wert, Zeiger)*.

[1] Allokation: Reservierung von Speicherplatz für ein Element

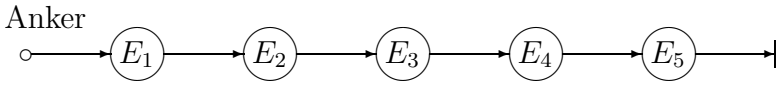

Abbildung 3.2.: Eine verkettete Liste

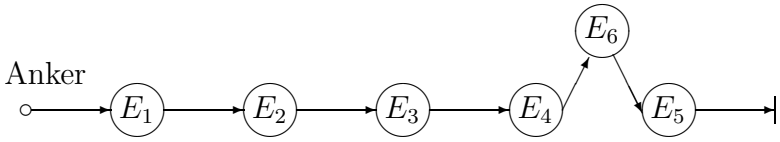

Abbildung 3.3.: Einfügen in eine verkettete Liste

Der Zeiger des Objekts zeigt auf die Speicherposition des nächsten Elements (Abbildung 3.2). Ein Zugriff auf jedes Element der Liste ist möglich durch Halten des Verweises auf das erste Element (Anker) und folgen der Verknüpfungen.

Der Aufwand zum Zugriff auf ein Element auf Basis von dessen Position in der Liste ist $O(Position)$. Dies stellt eine Verschlechterung im Vergleich zum Array dar, doch bietet die verkettete Liste Flexibilität, die über die eines Arrays weit hinaus geht. Das Einfügen eines Elements in die Liste ist (wenn die Speicherposition des Elements, hinter dem das neue Element eingefügt werden soll bekannt ist - dazu muss unter Umständen in oben genannter Art und Weise das Element zuerst durch Verfolgen der Verknüpfungen gefunden werden) nun ein Vorgang, der eine konstante Zeit benötigt (Abbildung 3.3). Der Verweis des vorangehenden Elements wird auf die Position des neuen Elements geändert und der Verweis des neuen Elements wird auf die Position des darauffolgenden Elements geändert. Mit vorangehender Suche benötigt die Einfügeoperation jedoch nach wie vor $O(N)$. Wenn auf die Elemente der verketteten Liste sequentiell zugegriffen werden soll, fällt dieser Nachteil jedoch weg.

Durch die separate Allokation der Elemente ist es im Gegensatz zum Array nun möglich, die Liste beliebig zu vergrößern. Wenn in der Liste Elemente vorrangig am Ende eingefügt werden, kann neben dem Anker, der auf das erste Element der Liste verweist, auch ein Schwanzzeiger auf das letzte Element gehalten werden. Mit dessen Hilfe sind Einfügeoperationen am Ende der Liste sehr schnell ($O(1)$) möglich, da der Zeiger nicht erst vom Anfang der Liste gesucht werden muss. Nach jedem Einfügen muss der Schwanzzeiger aktualisiert werden. Performantes

Rückwärtsablaufen der Liste ist durch Halten von zwei Verknüpfungen in jedem Element möglich; nicht nur auf das jeweils folgende Element, sondern auch auf das vorangehende. Diese Liste wird *doppelt verkettet* genannt.

Nachteil einer verketteten Liste ist die Uneffizienz der Speichernutzung, die besonders für kleine Elementgrößen schwer wiegt. Im schlechtesten Fall, einer Elementgröße von einem Byte, wird auf üblichen Systemen das vierfache der Elementgröße für einen Verweis auf eine Speicheradresse verwendet. Mit zunehmender Elementgröße relativiert sich dieses Problem jedoch.

3.1.3. Suche

Für das Auffinden eines Elements auf Grund einer Eigenschaft in einer unsortierten Menge muss die gesamte Menge durchsucht werden, der Aufwand ist also direkt linear proportional zur Anzahl der Elemente ($O(N)$). Wenn das Kriterium nur auf ein Element der Menge zutrifft, kann nach Auffinden dieses Elements die Suche abgebrochen werden; dies passiert im günstigsten Fall beim ersten Element der Liste, im schlechtesten Fall am Ende der Liste ($O(N)$).

Da diese schrittweise Suche lediglich iterativ von einem Element zu nächsten springt, existiert kein Laufzeitunterschied zwischen Array und verketteter Liste. Im Array kann der Index inkrementiert werden und das nächste Element ist in konstanter Zeit verfügbar. In der Liste wird der Verkettung gefolgt, damit ist das nächste Element auch in konstanter Zeit verfügbar.

Um die Laufzeit der Suche zu verringern, müssen der Menge weitere Eigenschaften gegeben werden, damit im Suchprozess aufgrund dieser Eigenschaften Annahmen über große Teile der Menge getroffen werden können, ohne dass diese Teile betrachtet werden müssen. Wenn die Menge z.B. sortiert ist, kann ich annehmen, dass alle Elemente hinter dem aktuellen Element größer (oder kleiner, abhängig von der Sortierung) als dieses sind. Damit fällt das Absuchen dieses Teils weg, wenn das gesuchte Element kleiner als das aktuelle ist.

Zum Suchen eines Elements in einer sortierten Menge mittels Binärsuche wird das Element in der Mitte der Menge (an Position $\frac{1}{2}N$) ausgewählt, und mit dem gesuchten Element verglichen. Ist das gesuchte Element kleiner als das Element in der Mitte der Menge, wird das Element an Position $\frac{1}{4}N$ ausgewählt. Ist das gesuchte Element größer als das Element in der Mitte der Menge, wird äquivalent

das Element an Position $\frac{3}{4}N$ zum weiteren Vergleich ausgewählt. So wird weiter verfahren, bis das Element gefunden wurde. Da sich bei jedem Schritt die zu durchsuchende Menge halbiert, beträgt die zur Suche benötigte Zeit $O(\log N)$, was besonders für große Mengen einen erheblichen Fortschritt gegenüber den $O(N)$ einer kompletten Durchsuchung bedeutet.

Für die Suche in einer sortierten Menge ist es aber nötig, Elemente schnell auf Basis ihrer Position in der Menge anzusprechen - eine Operation, die nur im Array, nicht aber in der Liste in konstanter, kurzer Zeit möglich ist. Im Array wiederum ist das Einfügen eines Elements aus oben beschriebenen Gründen schwer bis unmöglich, da die Größe des Arrays vorher bestimmt werden muss. Eine befriedigende Lösung für eine Suche kann also weder durch ein sortiertes Array noch durch eine sortierte, verkettete Liste gegeben werden.

3.2. Bäume

Ein Baum besteht aus Elementen der Struktur *(Wert, Zeiger 1, Zeiger 2, ...)*. Jedes Element (*Elternknoten*) enthält neben den eigentlichen Daten jeweils mehrere Zeiger auf weitere Elemente, die *Kinderknoten*. In einem Baum hat genau ein Element keinen Elternknoten; dieses Element ist die *Wurzel* des Baums. Ein Knoten ohne Kinderknoten wird als *Blatt* bezeichnet.

3.2.1. Binäre Bäume

Im binären Baum enthält jedes Element *zwei* Zeiger (lat. bina - paarweise) auf Kinderknoten. Das Sortieren eines binären Baums ist möglich, indem für jeden Knoten der Kinderknoten mit dem kleinern Wert der linke Kinderknoten ist, der Wert des rechten Kinderknotens ist größer als der des linken. Wenn diese Regel für alle Knoten erfüllt ist, kann der sortierte Baum ähnlich einer sortierten Menge schnell durchsucht werden. Dazu wird das zu suchende Element mit dem Wurzelknoten verglichen. Ist es kleiner als der Wurzelknoten, wird mit dem linken Kindknoten des Wurzelknotens fortgefahren; ist es größer, fährt man mit dem rechten fort. So kann iterativ das gesuchte Element gefunden werden.

Die Effizienz dieser Suche ist stark von der Form des Baums abhängig. Wenn in einen binären Baum schon sortierte Element eingefügt werden (also immer der

rechte Kindknoten des letzten Knotens werden), nimmt der Baum die Form einer verketteten Liste an, da es in keinem Knoten einen linken Kindknoten gibt. Damit gelten auch die Nachteile einer verketteten Liste bezüglich der Suche ($O(N)$).

Wenn der Baum jedoch balanciert ist, kann die Suche wie mit der Binärsuche in maximal $O(logN)$ erfolgen. Aufgabe muss es also sein, nach dem Einfügen von Elementen die Ausgeglichenheit des Baumes wiederherzustellen, damit die Suche schnell erfolgen kann.

3.2.2. AVL-Bäume

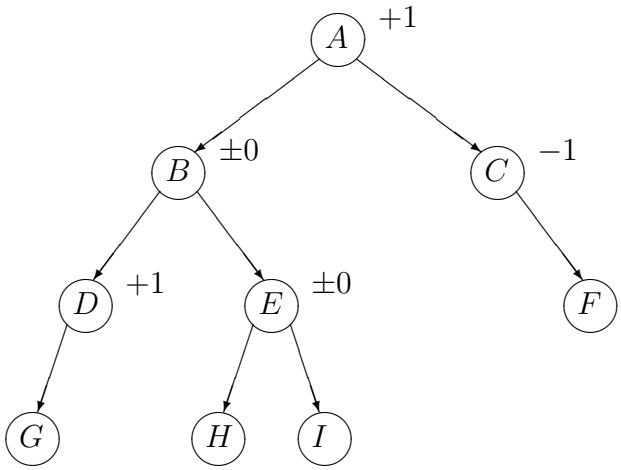

Abbildung 3.4.: Ein AVL-Baum mit balancierten Knoten

Ein AVL-Baum ist ein balancierter binärer Baum (siehe Abbildung 3.4). Er ist benannt nach Adelson-Velskii und Landis, die den AVL-Baum 1962 entwickelten (siehe [3], Seite 260). Ein binärer Baum ist balanciert, wenn alle seine Knoten balanciert sind. Ein Knoten ist balanciert, wenn sich die maximale Höhe seiner beiden Unterbäume (deren Wurzeln die beiden Kinderknoten sind) um nicht mehr als 1 unterscheidet.

Nach dem Einfügen oder Löschen eines Elements aus dem balancierten Baum kann es sein, dass diese Bedingung für einige Knoten verletzt ist. Ausgeglichen wird ein unbalancierter Knoten durch eine Rotation, die das Gleichgewicht wieder herstellt.

Rotation

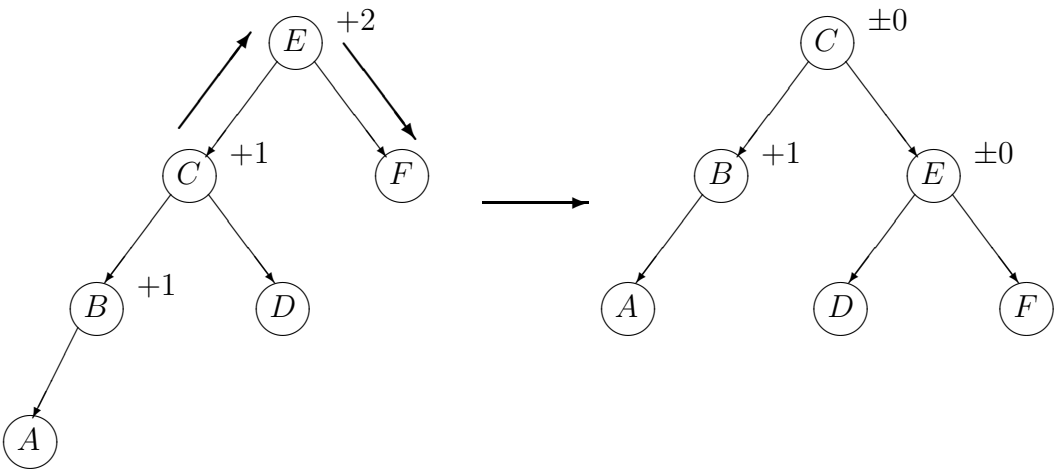

Abbildung 3.5.: Einfache Links-Rotation

Die einfache Rotation setzt dabei den linken Kindknoten B vom aktuellen Knoten A an die Position von A und lässt A den rechten Kindknoten von B sein. Dabei wird der rechte Unterbaum von B linker Unterbaum von A (siehe Abbildung 3.5).

Die doppelte Rotation involviert drei Knoten: Den aktuellen Knoten A, den linken Kindknoten B von A und den rechten Kindknoten C von B. C rutscht an Stelle von A, A wird der rechte Kindknoten von C, B der linke. Der linke Unterbaum von C wird rechter Unterbaum von B und der rechte Unterbaum von C wird linker Unterbaum von A (siehe Abbildung 3.6). [8]

Natürlich können beide Rotationen auch spiegelverkehrt durchgeführt werden.

3.3. Graphen

Ein Graph $G(\mathcal{V}, \mathcal{E})$ besteht aus der Menge seiner Knoten $\mathcal{V}$ und der seiner Kanten $\mathcal{E}$, wobei jede Kante ein Paar $(A, B) \in \mathcal{V} \times \mathcal{V}$ ist.

Ohne Angabe der Richtung einer Kante ist der Graph *ungerichtet*, und jede Kante kann bidirektional verwendet werden. Im *gerichteten* Graphen kann jede Kante nur in der angegebenen Richtung benutzt werden. Zusätzlich kann jede Kante ein Gewicht haben, dann spricht man von einem *gewichteten* Graphen.

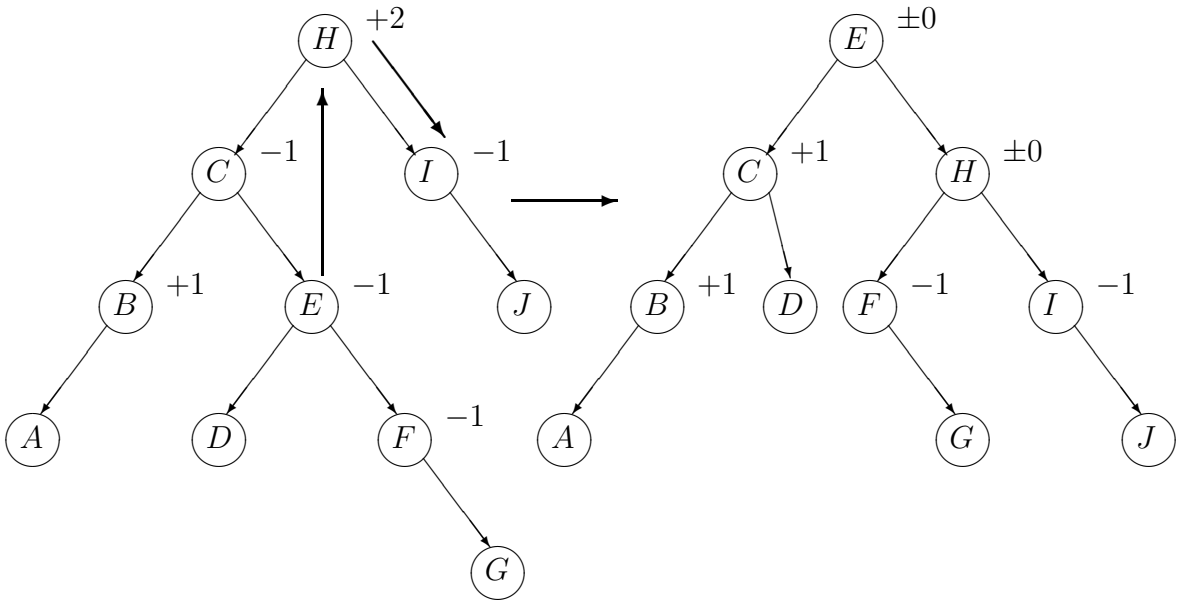

Abbildung 3.6.: Doppelte Links-Rechts-Rotation

Graphen sind komplexer als Bäume, denn von jedem Knoten kann eine Verbindung (Kante) zu jedem anderen führen. Dadurch lassen sich viele Probleme als Frage an einen bestimmten Graphen einfach darstellen. So kann zum Beispiel das Internet als Graph dargestellt werden: eine Seite (Knoten) verlinkt auf eine andere Seite, wenn eine Kante zwischen beiden existiert. Ein endlicher Automat kann auch durch einen Graphen dargestellt werden.

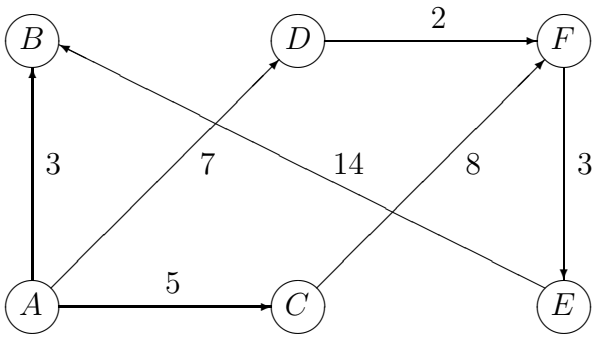

Abbildung 3.7.: Ein gerichteter, gewichteter Graph

Der Graph in Abbildung 3.7 ist sowohl gerichtet als auch gewichtet. Die Menge der Knoten $\mathcal{V}$ ist

$$\mathcal{V} = \{A, B, C, D, E, F\}$$

und die Menge der Kanten $\mathcal{E}$ ist

$$\mathcal{E} = \{\overrightarrow{AB}, \overrightarrow{AC}, \overrightarrow{AD}, \overrightarrow{DF}, \overrightarrow{CF}, \overrightarrow{FE}, \overrightarrow{EB}\}$$

3.3.1. Darstellung

Es gibt mehrere Möglichkeiten, einen Graphen rechnerintern darzustellen, die sich in Zugriffsgeschwindigkeit und Speicherplatzbedarf in Abhängigkeit von $\mathcal{V}$ und $\mathcal{E}$ unterscheiden:

Inzidenzmatrix Diese Darstellung benutzt ein Array A der Größe $E \cdot V$. Wenn die Kante E den Knoten V berührt, ist $A(E, V)$ 1, sonst 0.

Inzidenzliste Diese Darstellung benutzt eine Liste der Kanten, wobei für jede Kante beide Endknoten gespeichert werden. Die Menge der Knoten ist die Summe aller Knoten, die als Endknoten auftauchen. Es ist also nicht möglich, einen Knoten ohne anliegende Kante darzustellen.

Adjazenzmatrix Diese Darstellung benutzt ein Array A der Größe $V \cdot V$. Wenn zwei Knoten K und L durch eine Kante verbunden sind, ist $A(K, L)$ 1, sonst 0. Im gewichteten Graphen enthält das Feld das Gewicht der Kante.

Bei einer geringen Anzahl von Kanten belegt die Adjazenzmatrix unnötig viel Platz mit Nullen, anderseits bietet sie einen schnellen Zugriff auf eine Kante.

Adjazenzliste Diese Darstellung benutzt eine Liste aller Knoten. Jeder Knoten enthält eine Liste der durch eine Kante mit diesem Knoten verbundenen Knoten sowie u.U. das Gewicht dieser Kante.

Zum Zugriff auf eine Kante muss diese Liste durchsucht werden, was aber mit den in Abschnitt 3.2.1 beschriebenen Möglichkeiten beschleunigt werden kann.

4. Implementierung

4.1. Server-/Client-Konzept

Da das zu entwickelnde Programm Informationen (darüber, wie der Nutzer am schnellsten von A nach B kommt) anbieten soll, ist die nutzerfreundliche Aufbereitung dieser Informationen sehr wichtig. Wenn das Programm allen (zumindest sehr vielen) Nutzern Zugang zu den ermittelten Informationen bieten will, muss es dafür verschiedene Interfaces anbieten. Folgende Nutzerszenarien sollten dabei unbedingt berücksichtigt werden:

Durchschnittsnutzer Er möchte die Informationen bequem an seinem Rechner oder einem ähnlich ausgestatteten öffentlichen Terminal abrufen können. Dazu bietet sich entweder ein Webinterface oder ein eigenständiges Programm an.

Mobiler Nutzer Er möchte Fahrplaninformationen auf seinem PDA oder Handy abrufen. Zu beachten sind dabei das sehr kleine Display des Geräts und die durch i.d.R. drahtlose Kommunikation bedingte geringe Bandbreite zum Abrufen der aufbereiteten Informationen. Dem kann mit einem Webinterface speziell für kleine Displays oder einer WAP-Seite entsprochen werden.

Behinderter/Eingeschränkter Nutzer Hier muss sehr nach der Art der Behinderung unterschieden werden. Ein Sehbehinderter benötigt ein Programm/Webinterface, welches über große Schrift verfügt oder die Informationen akustisch ausgeben kann. Körperlich Behinderte müssen von besonderen Eingabehilfen unterstützt werden, was jedoch nur teilweise im Bereich der Software realisiert werden kann.

Deutlich wird, dass diese unterschiedlichen Benutzer nicht optimal mit einem einzigen Benutzerinterface bedient werden können. Da aber alle Benutzerinter-

faces auf den Algorithmus zugreifen müssen, um die Informationen überhaupt aufbereiten zu können, liegt hier der Gedanke eines definierten Interfaces nahe. So kann die Funktionalität, die benötigt wird, um die Informationen zu ermitteln (Server), einfach von der Funktionalität der Aufbereitung der Informationen (Client) getrennt werden. Zur Entwicklung eines Benutzerinterfaces (Client) ist dann keine Kenntnis des Servers mehr nötig, lediglich die Beschreibung des Interfaces.

Desweiteren entsteht durch ein netzwerktransparentes Interface der Vorteil, dass das sehr ressourcenintensive Serverprogramm nicht auf dem selben Rechner laufen muss wie der Client. So ist es möglich, mit einem gut ausgestatteten Server viele Nutzer gleichzeitig zu bedienen, die einen Client auf einem älteren Rechner nutzen.

Die Definition des in dieser Arbeit enwickelten Interfaces befindet sich im Anhang, Abschnitt A.1.

4.2. Architektur

Das in Abbildung 4.1 dargestellte Architekturstruktogramm des Programms verdeutlicht mehrere Designentscheidungen:

Trennung von Funktionalität (1) Wie schon in Abschnitt 4.1 beschrieben, kann durch das netzwerktransparente Interface die Last auf mehrere Server verteilt werden sowie ein einheitliches Interface geschaffen werden. Für die Implementierung des Interfaces auf Server-Seite ist die Klasse `RequestServer` zuständig.

Parallele Verarbeitung (2) Da der Server mehrere Anfragen gleichzeitig beantworten können soll, ist es notwendig, jedes Routing in einem separaten Thread durchzuführen. Alle Threads können gemeinsam auf den Graphen (`Station`) und die Linien (`TramLine`) zugreifen, jedoch ist zur Speicherung von Laufzeiteigenschaften einer Haltestelle (wie z.B. der Vorgängerhaltestelle im Routing) eine Datenstruktur notwendig, die für jeden Thread und jede Haltestelle separat diese Daten speichert (`StateMap`).

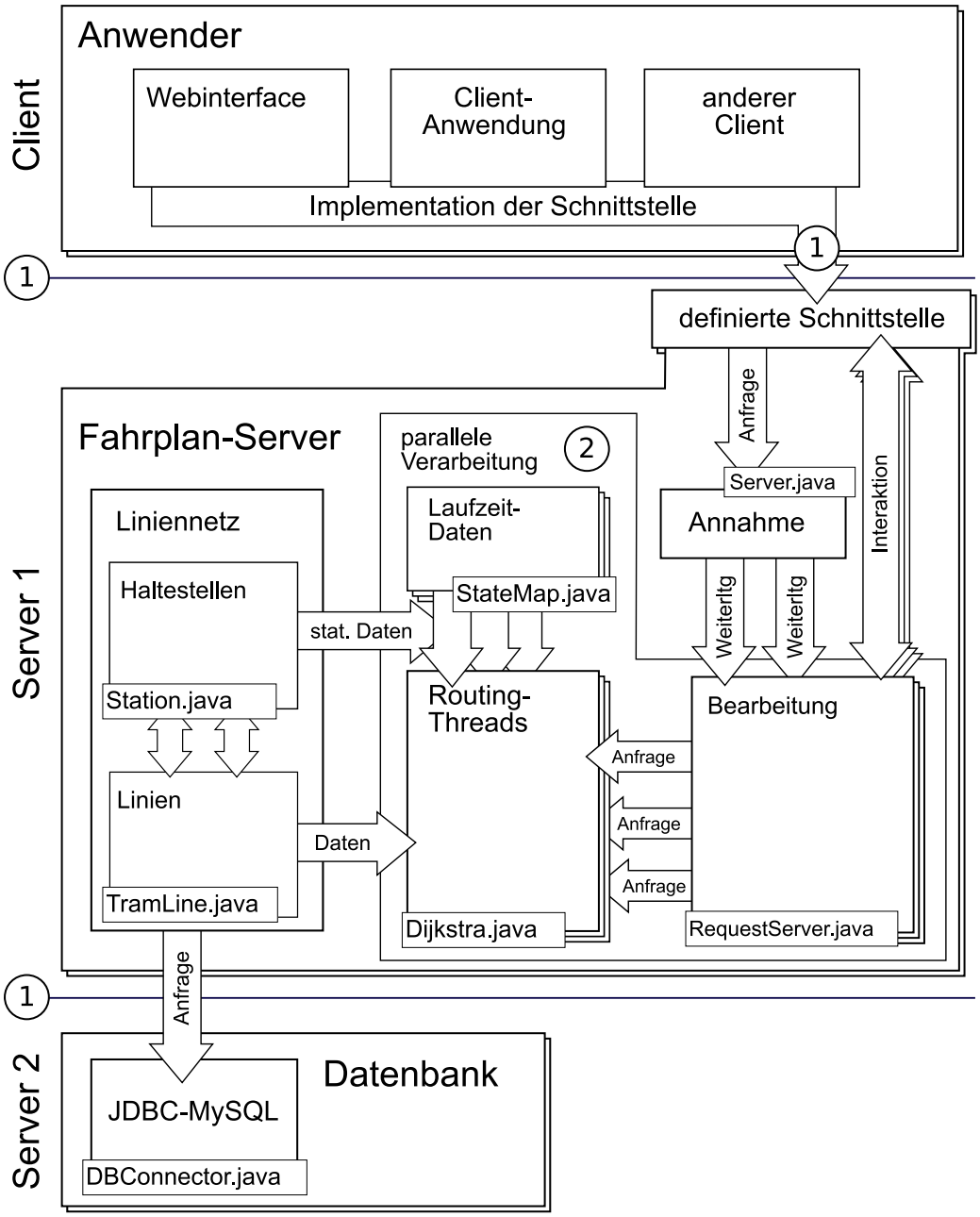

Abbildung 4.1.: Übersicht über die Architektur des Programms

4.3. Ergebnisse/Benchmarks

Um die Performance des Servers zu messen, wurde eine Testsuite erzeugt, die 20 Verbindungen zwischen völlig zufällig ausgewählten Haltestellen enthält. In jedem Testlauf wird die benötigte Zeit für jede Verbindung dieser Suite aufgezeichnet. Da die Suite immer die selben Verbindungen enthält, sind die Benchmarks untereinander vergleichbar.

4.3.1. Verwendung eines Heaps

Der Server verfügt über einen Konfigurationsschlüssel `Dijkstra.useHeap`, der entweder per `SET Dijkstra.useHeap`[1] oder durch Ändern der Konfigurationsdatei `server.xml` auf *true* oder *false* gesetzt werden kann. *true* bewirkt, dass der in Abschnitt 2.1.1 beschriebene Heap genutzt wird. In Abbildung 4.2 sind die Verbesserungen der Laufzeit dargestellt, die durch Verwendung des Heaps entstehen.

Die durchschnittliche Laufzeit ohne Heap beträgt $379ms$, mit Heap $120ms$. Die Verwendung eines Heaps verbessert die Laufzeit in diesem Fall auf weniger als $\frac{1}{3}$.

4.3.2. Verwendung der Server-VM

Java erlaubt durch Angabe einer Kommandozeilenoption (`-server`) das Wechseln von der Standard-(Client-)VM auf die Server-VM. Diese führt während der Ausführung des Programms eine Analyse der häufig verwendeten Stellen im Bytecode (sog. HotSpots) durch und compiliert diese in die plattformabhängige Maschinensprache. Dadurch läuft das Programm am Anfang langsamer, da die HotSpots ermittelt werden. Der dann compilierte Code erhöht jedoch die Ausführungsgeschwindigkeit in darauffolgenden Durchläufen erheblich. Dieser Gewinn wird besonders bei lange laufenden Server-Anwendungen deutlich, die immer das selbe tun. [10]

Nach dem Starten des Servers (Abbildung 4.3) braucht die Server-VM besonders bei der ersten Verbindung länger als die Client-VM. Die durchschnittliche Zeit für die Server-VM ist hier $243ms$, für die Client-VM sind es nur $164ms$.

[1] Siehe Interfacedefinition in Abschnitt A.1

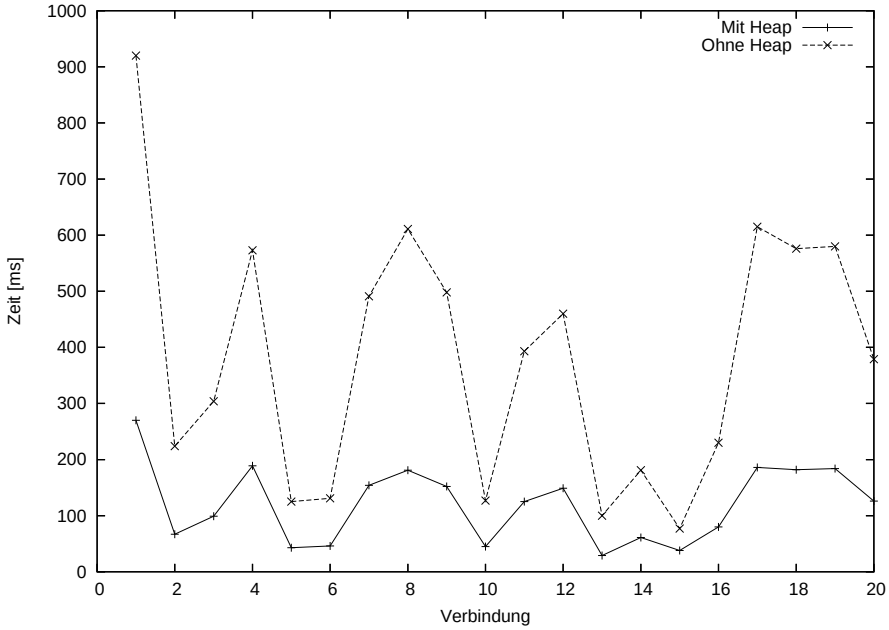

Abbildung 4.2.: Verbesserung der Laufzeit durch Verwendung eines binären Heaps

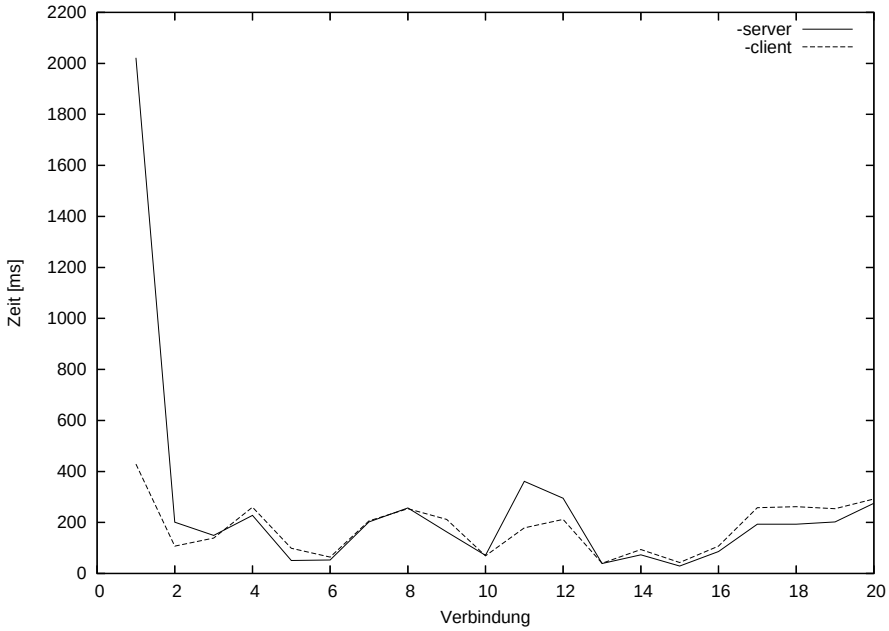

Abbildung 4.3.: Vergleich von Server- und Client-VM beim Start des Servers

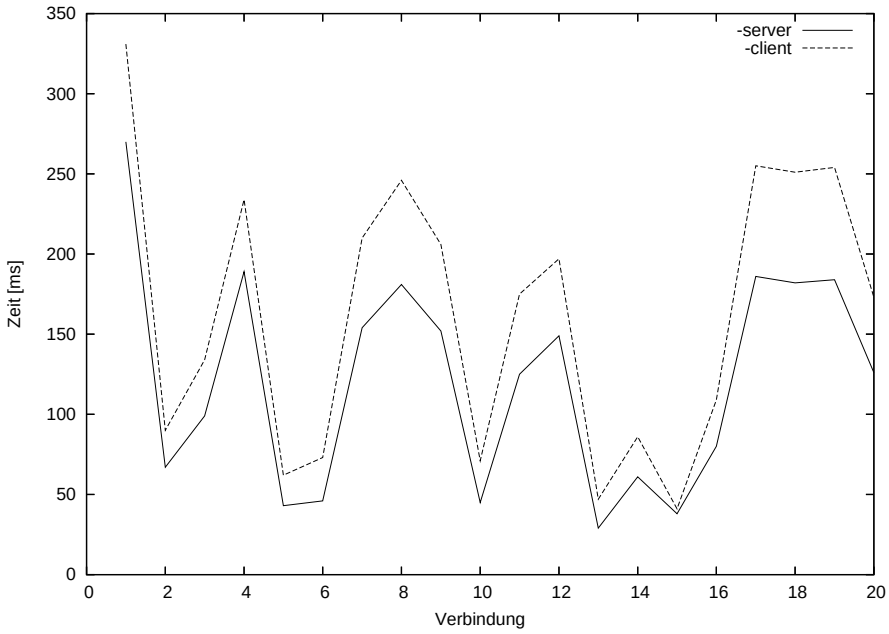

Abbildung 4.4.: Vergleich von Server- und Client-VM nach längerem Betrieb des Servers

Nachdem der Server eine größere Anzahl von Verbindungen berechnet hat, ist jedoch die Server-VM deutlich schneller (Abbildung 4.4). Die durchschnittliche Zeit für die Server-VM ist hier $120ms$, für die Client-VM sind es $153ms$. Besonders deutlich wird, dass die Client-VM ihre Performance nicht ändert, da sie den gesamten Bytecode interpretiert. Die Server-VM kann hier im optimierten Zustand immerhin eine Verbesserung um 25% gegenüber der Client-VM erreichen.

4.4. Clients

Für den Server habe ich einen graphischen Java-Client und ein webbasiertes PHP-Interface entwickelt (siehe CD). Diese beiden Clients stellen lediglich eine Demonstration der Flexibilität eines Interfaces dar und erheben keinen Anspruch auf Benutzerfreundlichkeit.

4.5. Daten

Die in der Arbeit verwendeten Fahrplandaten wurden mir im Herbst 2003 freundlicherweise von den Leipziger Verkehrsbetrieben zur Verfügung gestellt. Sie enthalten den Fahrplan für die 42. (Schulwoche) und 43. (Ferienwoche) Kalenderwoche des Jahres 2003.

Das Format der Daten ist in [11], Abschnitt 9 dargestellt.

A. Anhang

A.1. Interface

Das entwickelte Protokoll hat starke Ähnlichkeit mit beispielsweise dem SMTP[1]- oder HTTP-Protokoll[2]. Transportiert werden Befehle und Antworten über eine TCP-Verbindung. Dabei besteht jeder Befehl aus einer Zeile, mit dem Zeilenumbruch (*0x0d 0x0a*) wird dieser Befehl abgeschlossen. Argumente eines Befehls werden mit Leerzeichen (*0x20*) oder Tabulatoren (*0x09*) voneinander getrennt. Die Antwort des Servers kann aus mehreren Zeilen bestehen, sie wird durch eine Leerzeile (2 aufeinanderfolgende Zeilenumbrüche) beendet. Die erste Zeile der Serverantwort besteht aus einem Statusindikator, der angibt, ob der Befehl erfolgreich ausgeführt wurde (In Anlehnung an das HTTP-Protokoll steht 200 für eine erfolgreiche Ausführung und $4xx$ für einen Fehler), gefolgt von einer textuellen Repräsentation. Folgende Befehle stehen zur Verfügung:

AUTH <Benutzer> <Passwort> Dient der Authentifizierung des Benutzers. Ohne Argumente aufgerufen, gibt das Programm den aktuellen Status der Authentifizierung zurück. Andernfalls erwartet der Befehl zwei Argumente: Den Benutzernamen und das dazugehörige Passwort.

REQ LIST DAYTYPES Listet alle dem Server bekannten Tagestypen auf. Die Ausgabe erfolgt mit einem Typ pro Zeile, wobei in jeder Zeile durch Leerzeichen getrennt die interne ID des Tagestyps und sein Bezeichner ausgegeben werden.

REQ LIST MNEMONICS Listet alle Abkürzungen der Stationen auf.

REQ LIST STATIONS Listet alle einzelnen Stationen des Netzes auf.

[1] siehe RFC 821
[2] siehe RFC 2616

REQ LIST STATIONGROUPS Listet alle Stationen auf, jedoch werden zusammengehörige Stationen (z.B. alle Stationen des Hauptbahnhofs) als eine angezeigt. Die Ausgabe besteht aus dem Namen der Station und – getrennt von einem Leerzeichen – der ID der Station, mit einem vorangestellten #. Dies ist notwendig, da Namen von Stationen auf Leerzeichen enthalten können.

REQ CONN <Von> <Nach> <Tagestyp> <Zeit> Führt eine Fahrplananfrage durch. Argumente sind dabei die Start- und Zielhaltestelle (Angabe durch ihre ID), die ID des gewünschten Tagestyps und die Tageszeit in Sekunden. Der Server antwortet mit der optimalen Verbindung, wobei jede dabei besuchte Haltestelle in einer Zeile zurückgegeben wird. Jede Zeile enthält (durch Leerzeichen getrennt) die Abkürzung der Haltestelle, die Fahrzeit zur vorangehenden Haltestelle (im Falle der ersten Zeile die tatsächliche Startzeit in Sekunden), die aktuelle Straßenbahnlinie und den kompletten Namen der Haltestelle.

REQ INFO <ID/Kürzel> Gibt Informationen zur angegebenen Haltestelle aus. Die Angabe der Haltestelle kann entweder über die ID oder das Kürzel der Haltestelle erfolgen.

SET <Schlüssel> <Wert> Setzt den gegebenen Konfigurationsschlüssel auf den gegebenen Wert. Hierzu sind *operator*-Rechte notwendig.

RECONF Weist den Server an, seine Konfiguration neu einzulesen. Hierzu sind *operator*-Rechte notwendig.

WRITE <Datei> Schreibt den aktuellen Zustand des Servers in eine Datei, die später eingelesen werden kann. Hierzu sind *operator*-Rechte notwendig.

HALT Beendet den Server. Hierzu sind *operator*-Rechte notwendig.

INFO Gibt Informationen zum Server aus. Diese werden auch zu Beginn der Verbindung vom Server gesendet.

In Detailfragen soll der mit dieser Arbeit mitgelieferte Java-Server/-Client als Referenz angesehen werden.

A.2. Installation des Servers

Zur Inbetriebnahme müssen sich das Verzeichnis mit den Klassen des Programms sowie die in Tabelle A.1 aufgeführten Bibliotheken im *class path* befinden. Des Weiteren erwartet der Server eine Datei `logging.conf` im aktuellen Verzeichnis, die die Konfiguration der Statusausgaben des Servers beinhaltet. Die Datei `server.xml` (Pfad durch Java system property `bell.backend.Server.config` veränderbar) beinhaltet die Konfiguration des Servers, wie z.B. Informationen zum verwendenden Datenbankserver.

Tabelle A.1.: Benötigte Bibliotheken

Bibliothek	**Version**
Apache log4j [12]	1.2.8
MySQL Connector/J [13]	3.0.11

Der Einstiegspunkt des Programms ist die Klasse `Server`, also könnte der Programmaufruf z.B. so aussehen:

```
java -server -cp ./bin:./log4j-1.2.8.jar:\
  ./mysql-connector-java-3.0.11-stable-bin.jar \
  bell.backend.monkey.Server
```

A.3. Inhalt der CD

/Arbeit Eine Kopie dieser Arbeit im PDF-Format

/Datenbank Der Inhalt der Liniennetzdatenbank als SQL-Dump

/LVB-Netz Die Rohdaten des Liniennetzes sowie entsprechende Dokumentation des MDV

/Programme Der entwickelte Server sowie zwei Clients. TramPHP ist ein Webfrontend und benötigt einen PHP-fähigen Webserver, TramJava ist ein graphisches Interface und kann auf jeder Java-VM ausgeführt werden.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Donald E. Knuth: *The art of computer programming*, 1st Volume, 3rd Edition, Addison-Wesley, 1997

[2] Robert Sedgewick: *Algorithmen*, 2. Auflage, Addison-Wesley, 2002

[3] Thomas Ottmann, Peter Widmayer: *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag, 2000

[4] Wikipedia: *Big O notation* (eingesehen: 27. November 2004), `http://en.wikipedia.org/wiki/Big_O_notation`

[5] Wikipedia: *Shortest path problem* (eingesehen: 26. November 2004), `http://en.wikipedia.org/wiki/Shortest_path_problem`

[6] Wikipedia: *Dijkstra's algorithm* (eingesehen: 24. Januar 2004), `http://en.wikipedia.org/wiki/Dijkstra's_algorithm`

[7] Christian Raskob: *Kürzeste Wege in Graphen*, November 2002, `http://www.mcgods.de/fun/1904/Dijkstra-Bellford.html`

[8] Oliver Vornberger, Olaf Müller: *Algorithmen - Vorlesung im WS 2000/2001*, `http://www-lehre.informatik.uni-osnabrueck.de/~ainf/2000/skript/node60.html`

[9] Wikipedia: *Binary Heap*, eingesehen: 28. November 2004, `http://en.wikipedia.org/wiki/Binary_heap`

[10] Sun Microsystems, Inc.: *The Java HotSpotTM Server VM*, 1999, `http://java.sun.com/products/hotspot/docs/general/hs2.html`

[11] Verband Deutscher Verkehrsunternehmen, *ÖPNV Datenmodell 5.0, Standardschnittstelle Liniennetz/Fahrplan*
http://www.vdv.de/vorstellung/vdvorganisationen/grafik/Schri452_sds.pdf

[12] Apache Software Foundation: log4j Logging Services,
http://logging.apache.org/log4j/docs/index.html

[13] MySQL AB: Connector/J 3.0,
http://dev.mysql.com/downloads/connector/j/3.0.html

Erklärung zur selbstständigen Anfertigung der Arbeit

Hiermit erkläre ich, dass ich die Arbeit "Darstellung von Algorithmen und Datenstrukturen für die Entwicklung und effiziente Implementierung eines Routingalgorithmus für Verkehrsnetze" selbst verfasst habe.

Bei der Abfassung der Arbeit habe ich nur die angegebenen Hilfsmittel benutzt und wörtlich oder inhaltlich übernommene Stellen als solche gekennzeichnet.

Meißen, den 14.01.2005

Jonas Witt