

# Rolling Stock Assignment

Marco Nef  
Prof. Dr. Peter Widmayer

Semesterarbeit WS2000/2001  
D-INFK, Institut für Theoretische Informatik  
Eidgenössische Technische Hochschule Zürich





# Abstract

Das *Rolling Stock Assignment* ist die Problemstellung der Zuteilung von Rollmaterial auf die durch einen vorgegebenen Fahrplan definierten Fahrten eines Bahnsystems. In dieser Semesterarbeit werden schrittweise Algorithmen für immer komplexere Instanzen dieses in NP enthaltenen Problems hergeleitet und implementiert. Zuerst wird für eine einfache Bahnlinie eine Lösung gesucht, welche die minimale Anzahl Wagen benötigt. Später werden eine Kostenfunktion und Zwischendepots zur Bahnlinie hinzugefügt. Die Lösung des *Rolling Stock Assignment* wird dabei auf die Suche nach dem kürzesten Weg in einem Graphen reduziert.

# Danksagung

An dieser Stelle möchte ich Prof. Dr. Peter Widmayer herzlichst für die Betreuung dieser Semesterarbeit danken. Er hat mich in Gesprächen gefordert, meine Aussagen und theoretischen Herleitungen zu begründen und auch zu hinterfragen.

Einen grossen Dank auch an Nicole Tobler und Marc Bütikofer für das Korrekturlesen und die vielen daraus resultierenden Vorschläge zur Rechtschreibung und Verständlichkeit dieser Semesterarbeit.

Marco Nef, Zürich, den 7. Februar 2001



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Problemstellung . . . . .	1
1.2	Übersicht über die kommenden Kapitel . . . . .	2
<b>2</b>	<b>Einfache Bahnlinie</b>	<b>3</b>
2.1	Situation . . . . .	3
2.2	Fahrplangestaltung . . . . .	3
2.3	Varianten für das Rangieren . . . . .	4
2.4	Theoretische Überlegungen . . . . .	6
<b>3</b>	<b>Erster Algorithmus</b>	<b>9</b>
3.1	Annahmen . . . . .	9
3.2	Herleitung . . . . .	10
3.3	Implementierung . . . . .	11
3.4	Bemerkungen . . . . .	13
<b>4</b>	<b>Kostenrechnung</b>	<b>15</b>
4.1	Modellierung von Kosten . . . . .	15
4.2	Neue Fragestellungen . . . . .	16
4.3	Graphischer Ansatz . . . . .	16
4.4	Implementierung . . . . .	18
4.4.1	Datenstruktur . . . . .	18
4.4.2	Aufbau des Graphen . . . . .	19
4.4.3	Berechnung der Kosten . . . . .	22
4.4.4	Algorithmus von Dijkstra . . . . .	23
4.5	Bemerkungen . . . . .	24
<b>5</b>	<b>Zwischendepots</b>	<b>27</b>
5.1	Neue Situation . . . . .	27
5.2	Implementierung . . . . .	27
5.3	Verallgemeinerung auf $d$ Depots . . . . .	28

<b>6 Zusammenfassung</b>	<b>31</b>
6.1 Rückblick . . . . .	31
6.2 Ausblick . . . . .	31
<b>7 Quelltexte der Algorithmen</b>	<b>33</b>
<b>Literaturverzeichnis</b>	<b>35</b>
<b>Index</b>	<b>36</b>

# 1

## Einleitung

### 1.1 Problemstellung

Im Rahmen der Fahrplangestaltung für eine moderne Eisenbahn muss sich der Planer nicht nur die Frage stellen, wann die Züge fahren sollen, sondern er muss auch entscheiden, wie die einzelnen Zugskombinationen zusammenzustellen sind und wieviele Wagen an den Zug angehängt werden sollen.

Dieses sogenannte *Rolling Stock Assignment* hängt stark von der Passagiernachfrage ab, denn um Kosten zu sparen soll die Anzahl Wagen immer möglichst der Nachfrage entsprechen. Jeder zusätzlich mitgeführte Wagen benötigt Energie, was direkte Kosten erzeugt, die nicht durch Billettpreise oder andere Einnahmen<sup>1</sup> gedeckt sind. Auf der anderen Seite werden natürlich für jede Änderung der Wagenzahl Bahnarbeiter benötigt, deren Löhne wiederum Kosten verursachen.

Es gibt auch viele weitere Kostenarten, die nicht so einfach beziffert werden können. Ein Beispiel dafür ist der Imageverlust durch unzufriedene Passagiere, welcher eintritt, sobald zu kurze Züge verkehren und die Passagiere stehen müssen oder nicht einmal mitfahren können. Stehengelassene Personen werden der Bahn sofort den Rücken kehren und auf andere Verkehrsmittel umsteigen. Man hat es also mit einem Tradeoff zwischen Zufriedenstellung der Passagiere auf der einen Seite und der Rentabilität auf der anderen Seite zu tun.

Den Bahngesellschaften stehen dank regelmässigen Erhebungen der Passagierzahlen in den Zügen sehr genaue Daten über die Nachfrage an Transportleistung zur Verfügung. Diese Daten decken das gesamte Bahnnetz zu allen Tageszeiten und Wochentagen ab und können somit zur Optimierung der Zuglängen eingesetzt werden. Nichtsdestotrotz darf nicht vergessen werden, dass eine solche Nachfrageverteilung eine Zufallsvariable ist. Man wird

---

<sup>1</sup> Man muss sich dieselben Überlegungen auch für Güterzüge machen.

also immer auf der sicheren Seite sein, wenn man nicht zu eng kalkuliert, also zum Beispiel einen zusätzlichen Wagen an den Zug hängt, verbunden mit dem Risiko, dass dann gleich mehrere Wagen leer bleiben, weil an einem Tag zufälligerweise viel weniger Passagiere an den Bahnhöfen stehen.

Wie bereits an dieser Stelle sichtbar wird, handelt es sich beim *Rolling Stock Assignment* keineswegs um ein triviales Problem. Ganz im Gegenteil muss man schon bald erkennen, dass es nicht möglich ist, alle Kostenfaktoren und sonstigen Aspekte zu erfassen. Um so wichtiger ist es, dass die vorliegende Arbeit unter grossen Verallgemeinerungen begonnen wird. Der Leser wird mit fortlaufenden Kapiteln bemerken, wie schnell der Rechenaufwand für ein *Rolling Stock Assignment* unter Einbeziehung von nur wenigen Eigenschaften ansteigt.

## 1.2 Übersicht über die kommenden Kapitel

Ausgegangen wird von einer Bahnlinie, die nur aus je einem Bahnhof an den beiden Enden besteht. Diese Linie wird den Leser unter stetiger Erweiterung bis zum Ende der Arbeit begleiten. Das nachfolgende Kapitel versucht, die Eigenschaften einer Bahnlinie und eines *Rolling Stock Assignment* aufzuzeigen. Das dritte Kapitel wird einen ersten Algorithmus präsentieren, der sich darauf beschränkt, überflüssige Fahrten zu vermeiden.

Im vierten Kapitel werden dann Kosten eingeführt und das Ziel wird dann sein, möglichst ökonomisch zu fahren. Nachdem das fünfte Kapitel noch Depots entlang der Linie eingeführt haben wird, welche auch ein Rangieren entlang der Bahnlinie erlauben, werden im letzten Kapitel mögliche Erweiterungen der vorliegenden Überlegungen aufgezeigt werden.

# 2

## Einfache Bahnlinie

### 2.1 Situation

Gegeben ist eine Bahnlinie mit zwei Bahnhöfen  $A$  und  $B$ , welche die Bahnlinie begrenzen (Abbildung 2.1). Zur Verfügung stehen eine einzelne Lokomotive sowie eine Anzahl Wagen. Gegeben sind ausserdem eine Nachfragefunktion  $N_i(t)$ , welche für jeden Zeitpunkt  $t$  und Bahnhof  $i, i \in \{A, B\}$ , die Anzahl zu befördernder Personen angibt, sowie der Fahrplan, der bestimmt, wann die Züge fahren. Es kann ohne Einschränkung der Allgemeinheit angenommen werden, dass die Nachfragefunktion nur diskrete Werte für die erwarteten Passagierzahlen zu den Zeitpunkten der Abfahrten des Zuges von beiden Bahnhöfen liefert. Die Funktion wäre damit durch eine Wertetabelle beschreibbar.



Abbildung 2.1: Einfache Bahnlinie

### 2.2 Fahrplangestaltung

Zu Beginn ein paar Gedanken zur Fahrplangestaltung. Die Fragestellung ist dort ähnlich wie beim *Rolling Stock Assignment*. Das Problem, welches bei letzterem entsteht, wenn zuwenige Wagen fahren<sup>1</sup>, entspricht beim Erstellen eines Fahrplans dem Problem, welches entsteht, wenn zuwenige Züge

---

<sup>1</sup> Siehe Einleitung.

fahren. In beiden Fällen wird der Erfolg der betroffenen Bahngesellschaft längerfristig zu wünschen übrig lassen. Eine vertiefte Bearbeitung dieser Fragestellung bleibt in dieser Arbeit aus, es werden jedoch an dieser Stelle ein paar Gedanken dazu geäußert.

Man kann sich verschiedene Varianten überlegen, wie entschieden werden soll, wann ein Zug fährt, wie also der Zugverkehr gesteuert wird:

1. Der Zug fährt ununterbrochen hin und zurück, ohne dass man an der Kombination etwas ändert. Die Kombination wird so zusammengestellt, dass die maximale Nachfrage bewältigt werden kann.
  - ⊕ Die Spitzenzeiten werden optimal abgedeckt, denn der Zug zieht genau die maximal benötigte Anzahl Wagen.
  - ⊖ In Randzeiten wird zu oft gefahren respektive die Auslastung der Wagen ist zu klein.
2. Der Zug fährt, sobald er voll ist oder einen geforderten Füllgrad erreicht hat.
  - ⊕ Das Rollmaterial wird optimal ausgenutzt, denn es fahren nur volle Züge.
  - ⊖ Die Entscheidung wird lokal gefällt, weshalb es möglich ist, dass an der anderen Station Passagiere lange Zeit warten müssen.
3. Der Zug fährt, sobald er voll ist respektive einen geforderten Füllgrad erreicht hat, oder damit er zu einem Zeitpunkt die andere Station erreicht, wo er dort einen genügenden Füllgrad erreichen kann.
  - ⊕ Das Rollmaterial wird in dem Sinne optimal ausgenutzt, dass der Zug immer entweder ausgelastet fährt oder aber die nachfolgende Fahrt ausgelastet sein wird.
  - ⊕ Es können Kosten gespart werden, weil der Zug in Randzeiten seltener fährt.

Im Folgenden wird angenommen, dass der Fahrplan gegeben ist. Die ökonomische Optimierung wird darin bestehen, dass die Wagen möglichst effizient eingesetzt werden.

### 2.3 Varianten für das Rangieren

Wenn man nun zulässt, dass die Zugskompositionen zwischen zwei Fahrten geändert werden, so muss man erkennen, dass einerseits durch das Rangieren die Zuglänge an die Nachfrage angepasst werden kann, was Kosten spart, andererseits dadurch aber auch neue Kostenträger entstehen:

- Für die Rangierarbeiten werden zusätzliche Arbeiter benötigt
- Die Rangierarbeiten selber benötigen eine gewisse Zeit, während welcher der Zug nicht fahren kann, also nicht produktiv eingesetzt werden kann.

Es gibt wiederum verschiedene Varianten von Rangieren, wobei hier solche präsentiert werden, welche in einem Aspekt ein Extrem darstellen. Es sind zwischen jeglichen der aufgeführten Möglichkeiten feine Abstufungen vorstellbar.

### 1. Kein Rangieren

Dies bedeutet, dass eine fixe Anzahl Wagen an der Lokomotive angehängt werden. Es gibt verschiedene Möglichkeiten, zu entscheiden, wie gross diese Zahl sein soll:

- a) Die Anzahl Wagen entspricht derjenigen Kapazität, welche benötigt wird, um die durchschnittliche Nachfrage über den gesamten Tag zu befriedigen. Damit ist es möglich, alle Passagiere (irgendwann) ans Ziel zu bringen. Der grosse Nachteil daran ist, dass es während Stosszeiten zu Wartezeiten und somit zu Unzufriedenheit der Kunden kommt.
- b) Die Anzahl Wagen entspricht derjenigen Kapazität, welche zur Bewältigung der grössten Nachfrage des Fahrplans<sup>2</sup> benötigt wird. Hierbei sei angenommen, dass die Lokomotive beliebig viele Wagen ziehen kann. Der Nachteil ist, dass die Kapazität ausserhalb der Stosszeiten viel zu gross ist.

### 2. Mit Rangieren

Die Frage ist nun, zu welchem Zeitpunkt welche Kombinationen zusammengestellt werden sollen.

- a) Vor jeder Fahrt wird genau die benötigte Anzahl Wagen angehängt. Dies hat den Vorteil, dass die Kapazität immer der Nachfrage entspricht. Auf der anderen Seite werden sehr viele Wagen benötigt, weil zum Beispiel am Morgen lange Züge von *A* nach *B* fahren, aber nur kurze Züge zurück. Am Abend ist dann die Situation gerade umgekehrt. Ebenfalls werden grosse Depots<sup>3</sup> an beiden Bahnhöfen benötigt.

---

<sup>2</sup> In dieser Arbeit bezieht sich ein Fahrplan auf eine Lokomotive, was bedeuten kann, dass der Zeitrahmen nicht einem Tag entspricht. Dies kann sich der Leser einfach veranschaulichen, wenn er sich den Orient Express vorstellt, der für die Reise von Moskau nach Peking wesentlich länger als einen Tag braucht. Im internationalen Zugverkehr sind solche Situationen üblich.

<sup>3</sup> In Depots werden die nicht benutzten Wagen zwischengelagert, bis sie wieder an einen Zug angehängt werden. Umgekehrt ist ein Depot notwendig, damit an einem Bahnhof Wagen zurückgelassen werden können.

- b) Der Tag wird in Blöcke mit ähnlicher Nachfrage unterteilt. Die Zugkompositionen werden jeweils zu Beginn eines solchen Blocks zusammengestellt und dann für die Zeit des Blocks nicht mehr verändert. Der Vorteil dabei ist, dass die Kapazität zu jedem Zeitpunkt der Nachfrage innerhalb gewisser Schranken gerecht wird. Eine wesentlich zu grosse Kapazität auf den Rückfahrten gibt es nur zu den Stosszeiten. Während dem Rest der Fahrplandauer kann die Nachfrage als in beiden Richtungen etwa gleich gross angenommen werden, weshalb dann keine wesentlichen Überkapazitäten entstehen.

## 2.4 Theoretische Überlegungen

Im Folgenden werden en bloc einige theoretische Überlegungen angestellt, welche auf die Entwicklung des ersten Algorithmus im Kapitel 3 hinführen und dieselbe vereinfachen.

**Definition 1** *Ein Leistungsverlust besteht, wenn weniger Passagiere transportiert werden können, oder wenn die Passagiere das Ziel später erreichen.*

**Theorem 1** *Eine Strecke mit zwei identischen Zügen und Taktfahrplan<sup>4</sup> kann ohne Leistungsverlust durch eine Strecke mit nur einem Zug ersetzt werden, falls eine doppelte Fahrgeschwindigkeit möglich ist.*

**Beweis:** Aus Abbildung 2.2 oben und unten folgt, dass der einzelne Zug jede Einzelfahrt der beiden anderen Züge durch zwei Fahrten doppelter Geschwindigkeit ersetzen kann. Dadurch bleibt die Anzahl Abfahrten und Ankünfte an beiden Endstationen konstant, was bedeutet, dass kein Leistungsverlust auftritt.

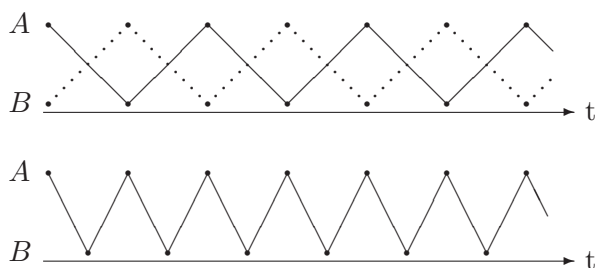


Abbildung 2.2: Oben: Strecke mit zwei Zügen, normale Geschwindigkeit; Unten: Strecke mit nur einem Zug, doppelte Geschwindigkeit

□

<sup>4</sup> Auf einem Streckennetz mit Taktfahrplan fährt von jedem Bahnhof in gleichmässigen Abständen ein Zug in eine bestimmte Richtung ab respektive erreicht den Bahnhof.

**Theorem 2** Wenn der einzelne Zug anstatt mit doppelter Geschwindigkeit zu fahren eine doppelt so grosse Anzahl Wagen zieht, tritt ein Leistungsverlust auf.

**Beweis:** Aus Abbildung 2.2 oben folgt durch Weglassen eines Zuges, dass jeder Bahnhof nur noch halb so oft angefahren wird, respektive der Takt halbiert wird. Dadurch verschiebt sich die Abfahrt und damit auch die Ankunft eines Teils der Passagiere, was einem Leistungsverlust entspricht.  $\square$

**Definition 2** Die Nachfrage  $N_i(t)$ ,  $i = 1 \dots n$ ,  $n \in \mathbb{N}$ , ist die Anzahl Personen, welche zum Zeitpunkt  $t$  vom Bahnhof  $S_i$  ( $S_1 = A, S_n = B$ ) weiterfahren wollen, also genau diejenigen Passagiere, welche sich bei der Ankunft in  $S_i$  bereits im Zug befunden hatten, aber nicht ausgestiegen sind, plus diejenigen Personen, welche in  $S_i$  den Zug bestiegen haben.

**Definition 3** Die Nachfrage  $N(t)$  ist das maximale  $N_i(t)$ ,  $i = 1 \dots n - 1$ , das der Zug, der  $A$  zum Zeitpunkt  $t$  verlässt, auf der Fahrt nach  $B$  antrifft. Analog für die Gegenrichtung von  $B$  nach  $A$  mit  $i = n \dots 2$ .

**Anmerkung:**  $N(t)$  definiert mit dem Zeitpunkt  $t$  implizit auch die Fahrtrichtung, denn  $t$  sagt aus, wann der Zug einen Endbahnhof verlässt. Zusammen mit dem (gegebenen) Fahrplan ist somit auch der Bahnhof gegeben. Durch  $\vec{N}(t)$  respektive  $\overleftarrow{N}(t)$  kann zusätzlich die Fahrtrichtung angegeben werden, was aber nur der Lesbarkeit dient.

**Theorem 3** Wenn eine Bahnlinie nur an den beiden Endbahnhöfen  $A$  und  $B$  (siehe Abbildung 2.3) ein Depot besitzt, so muss nur die Nachfrage  $N(t)$  für die Berechnung des Rolling Stock Assignment berücksichtigt werden.



Abbildung 2.3: Bahnlinie mit  $n$  Stationen

**Beweis:** Trivial, denn da sich in den Stationen  $S_2$  bis  $S_{n-1}$  keine Depots befinden, können dort auch keine Rangierarbeiten ausgeführt werden. Es muss also für die ganze Fahrt die Nachfrage  $N(t)$  verwendet werden.  $\square$



# 3

## Erster Algorithmus

### 3.1 Annahmen

Für den ersten, nun zu entwickelnden Algorithmus wird eine Reihe von Annahmen getroffen. Die vorläufig nicht berücksichtigten Elemente vereinfachen die Implementierung, fokussieren also den Blick auf das Wesentliche und fördern damit das Verständnis. Die meisten davon können aber mit wenig Aufwand nachträglich eingefügt werden.

- Die Gesamtzahl vorhandener Wagen ist beschränkt durch die grösste innerhalb der Fahrplanperiode auftretende Nachfrage.
- Alle Wagen sind identisch (betrifft vor allem die Sitzzahl respektive Kapazität).
- Die Lokomotive kann die gesamte vorhandene Menge an Wagen gleichzeitig ziehen.
- An den beiden Endbahnhöfen  $A$  und  $B$  (siehe Abbildung 2.1) existiert je ein Depot  $D_i$ ,  $i \in \{A, B\}$ , welches die vorhandene Anzahl Wagen aufnehmen kann.
- Die Lokomotive zieht bei jeder Fahrt mindestens einen Wagen.

Unter diesen Annahmen sind für die Berechnung eines *Rolling Stock Assignment* nur noch der Fahrplan, die Nachfragefunktion  $N(t)$  und die Kapazität  $c$  der Wagen als Parameter notwendig. Die benötigte Anzahl Wagen  $q_{tot}$  kann aus dem Maximum der Nachfrage und der Kapazität berechnet werden:

$$q_{tot} = \max_t \left\{ \left\lceil \frac{N(t)}{c} \right\rceil \right\}$$

Der herzuleitende Algorithmus verteilt diese  $q_{tot}$  Wagen optimal auf die einzelnen Zugfahrten. Als optimal wird hierbei bezeichnet, wenn es nicht möglich ist, den Fahrplan mit weniger Leerfahrten<sup>1</sup> als vom Algorithmus berechnet auszuführen.

### 3.2 Herleitung

Die Fragestellung ist jetzt also, wann in welchem Depot wieviele Wagen abgestellt werden sollen, um möglichst wenige Leerwagen mitzuführen, aber trotzdem jede Nachfragenspitze bewältigen zu können.

Wann immer sich ein Zug zu einem Zeitpunkt  $t$  in einem Bahnhof befindet, so gibt es drei Möglichkeiten, wie die Situation aussehen kann. Ohne Einschränkung der Allgemeinheit wird hier davon ausgegangen, dass der Zug zur Zeit  $t$  mit  $q(t)$  Wagen vom Bahnhof  $A$  nach  $B$  abfährt, wo er zur Zeit  $t + 1$  die Rückfahrt mit  $q(t + 1)$  Wagen antreten wird:

a)  $q(t) = q(t + 1)$

⇒ Der Zug benötigt für die Fahrt  $q(t) \leq q_{tot}$  Wagen

⇒ Er kann  $d_A(t) = q_{tot} - q(t) - d_B(t)$  Wagen in  $A$  stehen lassen und muss dann ...

a<sub>1</sub>) falls  $d_B(t) \neq 0$ : Die Situation in  $B$  neu beurteilen, weil dann eventuell aus dem Depot  $d_B(t)$  Wagen für die Fahrt zur Zeit  $t + 2$  mitgenommen werden müssen

a<sub>2</sub>) falls  $d_B(t) = 0$ : Die Situation erst beim nächsten Eintreffen in  $A$  neu beurteilen, denn im Depot  $d_B(t)$  befinden sich keine Wagen, was bedeutet, dass er für die Fahrt zur Zeit  $t + 2$  keine Wagen mitnehmen kann.

b)  $q(t) > q(t + 1)$

⇒ Der Zug benötigt für die Fahrt  $q(t)$  Wagen

⇒ In  $B$  muss die Situation für die folgenden Fahrten neu beurteilt werden.

c)  $q(t) < q(t + 1)$

⇒ Der Zug benötigt für die Rückfahrt von  $B$  nach  $A$  zur Zeit  $t + 1$   $q(t + 1)$  Wagen.

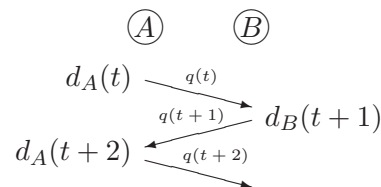
⇒ Für die Fahrt von  $A$  nach  $B$  sind also  $\max\{q(t), q(t + 1) - d_B(t)\}$  Wagen notwendig.

---

<sup>1</sup> Eine Leerfahrt liegt vor, wenn die Lokomotive einen Wagen zieht, welcher bei der erwarteten Nachfrage nicht benötigt wird.

⇒ **Theorem 4** Falls  $d_B(t) \neq 0$  oder  $q(t) > q(t+1) - d_B(t)$ , so muss die Situation in  $B$  neu beurteilt werden, ansonsten erst bei der Rückkehr nach  $A$ .

**Beweis:**



1.  $d_B(t) = 0$   
 ⇒ Weil  $q(t) \leq q(t+1)$ , müssen, wegen  $d_B(t) = 0$ , schon für die Fahrt von  $A$  nach  $B$   $q(t) = q(t+1)$  Wagen mitgeführt werden.
2.  $d_B(t) > 0$   
 ⇒ Weil im Bahnhof  $B$  noch Wagen vorhanden sind, müssen für die Fahrt dorthin  $q(t) \leq q(t+1)$  Wagen mitgezogen werden. Der Rest kann in  $B$  angehängt werden, was eine Neubeurteilung der Situation bedingt.
3.  $q(t) > q(t+1) - d_B(t)$   
 ⇒ Falls die Hin- und Retourfahrt ohne Neubeurteilung in  $B$  erfolgen, sind für die Fahrt zum Zeitpunkt  $t+2$  mit  $q(t+2)$  möglicherweise nicht genug Wagen in  $A$  vorhanden.  
 ⇒ Die Situation muss in  $B$  neu beurteilt werden.
4.  $q(t) \leq q(t+1) - d_B(t)$   
 ⇒  $\max\{q(t), q(t+1) - d_B(t)\} = q(t+1) - d_B(t)$   
 ⇒  $d_B(t+1) = 0$   
 ⇒ Nach der Retourfahrt werden alle Wagen in  $A$  sein.  
 ⇒ Es genügt, beim nächsten Eintreffen in  $A$  zur Zeit  $t+2$  die Situation neu zu beurteilen. □

**Satz 1** Es genügt, die Situation unter Betrachtung der nächsten zwei Fahrten einzuschätzen, um das Rolling Stock Assignment zu berechnen.

**Beweis:** Folgt aus der obigen Herleitung und aus Theorem 4. □

### 3.3 Implementierung

Die obige Herleitung führt direkt zum ersten Algorithmus. Da die Implementierung auf dem Internet verfügbar ist (siehe Kapitel 7), und auch aus Platzgründen, wird hier nur der Kern des Algorithmus aufgeführt. Das wird auch bei den nachfolgenden Algorithmen der Fall sein.

Die Parameter für den Algorithmus sind folgendermassen definiert:

`qTotal` Anzahl verfügbare Wagen.  
`c` Kapazität der Wagen.  
`N` `var N=new Array(hin,her, ...);`  
 Das Array enthält die Nachfrage als Anzahl Passagiere für die  $i$ -te Fahrt. Modulo 2 kann man die Fahrtrichtung berechnen.  
`d` `var d=new Array(Depot1, Depot2);`  
 Anzahl Wagen in den beiden Depots zu Beginn des Fahrplans.  
 Die Summe muss  $q_{tot}$  entsprechen.

Damit sieht der Algorithmus folgendermassen aus:

```

1  var n=N.length();      // Anzahl Fahrten.
2  var doCheck=true;     // Situation anfangs beurteilen.
3  var q=new Array(2);   // Nächste zwei Fahrten.
4  var t;                // Zeitzähler.
5
6  for(t=0;t<n;t++)
7  { if(doCheck)
8    { q[t%2]=Math.ceil(N[t]/c);
9      q[(t+1)%2]=Math.ceil(N[(t+1)%n]/c);
10     if(q[t%2]==q[(t+1)%2])doCheck=(d[(t+1)%2]!=0)
11     else if(q[t%2]>q[(t+1)%2])doCheck=true
12     else
13     { q[t%2]=Math.max(q[t%2],q[(t+1)%2]-d[(t+1)%2]);
14       doCheck=(q[t%2]>q[(t+1)%2]-d[(t+1)%2]);
15       doCheck=doCheck|| (d[(t+1)%2]!=0);
16     }
17     d[t%2]=qTotal-d[(t+1)%2]-q[t%2];
18   }
19   else doCheck=true;
20 }
```

### 3.4 Bemerkungen

- a) **Theorem 5** *Der vorgestellte Algorithmus liefert eine optimale Lösung, das heisst, es kann nicht mit weniger Wagen gefahren werden (bei gegebener Startkonfiguration).*

**Beweis:**

1.  $q(t) = \left\lceil \frac{N(t)}{c} \right\rceil$   
 $\Rightarrow$  trivial, denn mit weniger Wagen wäre die Nachfrage nicht befriedigt.
2.  $q(t) > \left\lceil \frac{N(t)}{c} \right\rceil$   
 $\Rightarrow \left\lceil \frac{N(t+1)}{c} \right\rceil > \left\lceil \frac{N(t)}{c} \right\rceil + d_B(t)$   
 $\Rightarrow d_B(t+1) = 0$   
 $\Rightarrow \text{AnzahlLeerwagen} = \left\lceil \frac{N(t+1)}{c} \right\rceil - \left\lceil \frac{N(t)}{c} \right\rceil - d_B(t)$   
 $\Rightarrow$  Jeder dieser Leerwagen wird bei der Rückfahrt benötigt.  
 $\Rightarrow$  Kein Wagen wird überflüssigerweise mitgezogen.  
 $\Rightarrow$  Algorithmus liefert optimale Lösung.  $\square$

- b) **Theorem 6** *Der von einer beliebigen gültigen Ausgangslage durch den Algorithmus erreichte Endzustand stellt eine optimale Startkonfiguration für einen periodischen Fahrplan dar.*

**Beweis:** Der Algorithmus betrachtet für die Berechnung der letzten Fahrt auch die erste Fahrt.

- $\Rightarrow$  Falls der Algorithmus die optimale Lösung liefert, so kann der Endzustand als optimaler Zwischenzustand vor der nächsten Fahrt (entspricht der ersten Fahrt) des periodischen Fahrplans betrachtet werden.
- $\Rightarrow$  Theorem 5 zeigt, dass diese Voraussetzung erfüllt ist.
- $\Rightarrow$  Endzustand ist optimale Startkonfiguration.  $\square$

Um eine optimale Startkonfiguration zu erhalten, kann also der Algorithmus auf eine beliebige gültige Startkonfiguration angewandt werden. Der erhaltene Endzustand kann danach als optimale Startkonfiguration benutzt werden.

- c) Der Algorithmus kann auch für Züge mit fest zusammengestellten Kompositionen<sup>2</sup>, bestehende aus mehreren Wagen, verwendet werden, wenn als „Wagenkapazität“  $c$  die Kapazität einer solchen Komposition genommen wird.

---

<sup>2</sup> z.B. TGV, Cisalpino, ICN, S-Bahn Zürich.

- d) Unter der Annahme, dass sich nur bei  $A$  ein Depot befindet, wird der Algorithmus sogar noch vereinfacht:

$$\Rightarrow q(t) = \max \left\{ \left\lceil \frac{N(t)}{c} \right\rceil, \left\lceil \frac{N(t+1)}{c} \right\rceil \right\} \forall t = k \cdot 2, k \in \mathbb{N}$$

Die Zugkapazität muss also immer dem Maximum der Nachfrage für die Hin- und Rückfahrt genügen.

# 4

## Kostenrechnung

### 4.1 Modellierung von Kosten

Als erste Erweiterung des ursprünglichen Modells werden in diesem Kapitel Kostenträger eingeführt. Wie bereits in der Einleitung besprochen gibt es eine ganze Menge von Kostenträgern, die berücksichtigt werden können.

Für den Rest der Arbeit werden die Kosten als Funktion  $k(i, t, \Delta t, f, l, r)$  der im folgenden beschriebenen Parameter modelliert, wobei in der Auflistung jeweils in Klammern exemplarisch angegeben ist, wie die Parameter einen Einfluss auf die Kosten für den Bahnbetrieb haben können. Die Kostenfunktion wird als gegeben angenommen.

- Bahnhof  $i$ , von wo der Zug abfährt (Lohngefälle Stadt/Land).
- Zeitpunkt  $t$  der Abfahrt (Lohnunterschiede Tag/Nacht).
- Zeitdauer  $\Delta t$  der mit der gewählten Kombination ausgeführten Fahrten (Amortisation der Rangierkosten).
- Anzahl  $f$  der mitgezogenen Wagen (Energiekosten).
- Anzahl  $l$  der Leerwagen, kummuliert über die Anzahl Fahrten mit der Zugkombination (Nicht genutzte Kapazität während allen Fahrten mit der Kombination).
- Anzahl  $r$  der zu Beginn der Fahrt rangierten Wagen (Lohnkosten).

Da diese Funktion alle wesentlichen Ursachen für entstehende Kosten als Parameter erhält, ist es damit auch möglich, die grosse Mehrheit der erwähnten Kostenträger zu monetarisieren. Die Kostenträger können auch beliebig komplex miteinander verknüpft werden (zum Beispiel Lohnkosten in der Stadt und während der Nacht, wenn mindestens drei Wagen rangiert werden).

## 4.2 Neue Fragestellungen

Mit der Einführung von Kosten entstehen eine Reihe neuer Fragestellungen, welche betrachtet werden können. Beispiele sind ...

- Lohnt es sich, nach jeder Fahrt die Zugkombination zu ändern?
- Wie lange soll eine Zugkombination konstant bleiben?
- Wieviele Wagen müssen mindestens ab-/angehängt werden können, damit sich das Rangieren lohnt?
- In welchem Verhältnis stehen die Rangierkosten zu den Kosten für eine Leerfahrt?
- Welche Kostenträger verursachen den Hauptteil der Gesamtkosten?

Diese Fragestellungen zeigen deutlich den Zusammenhang mit der Betriebswirtschaft, schliesslich geht es darum, den Bahnbetrieb möglichst kostengünstig zu halten.

## 4.3 Graphischer Ansatz

Wenn man nicht mehr nur garantieren will, dass genügend Wagen vorhanden sind, um die Nachfrage zu bewältigen, sondern gleichzeitig auch die Betriebskosten minimieren will, so kann die Situation auftreten, dass eine Zugkombination während mehreren Fahrten konstant bleibt. Deshalb genügt es nicht mehr, nur zwei Fahrten vorzuschauen. Der im vorangehenden Kapitel besprochene Ansatz kann somit nicht weiterverwendet werden.

Der neue Ansatz ist, einen Graphen aufzubauen, dessen Knoten  $K$  für jeden Zeitpunkt  $t$  die möglichen Zustände  $(d_A, d_B)$  der beiden Depots nach Abfahrt des Zuges repräsentieren:

$$\left( \begin{array}{c|c} a & b \end{array} \right) \quad \begin{array}{l} a = \text{Anzahl Wagen im Depot A} \\ b = \text{Anzahl Wagen im Depot B} \end{array}$$

Von jedem Knoten aus gehen Verbindungen zu allen Knoten der späteren Zeitpunkte, die einen Depotzustand repräsentieren, welcher vom ausgehenden Knoten her mit der minimal benötigten Anzahl Wagen erreicht wird (minimale Verbindung). Angenommen wird hierbei, dass nicht alleine durch das Mitführen eines Wagens Geld verdient werden kann, was plausibel ist, denn die Lokomotive benötigt für jeden zusätzlich gezogenen Wagen auch mehr Energie.<sup>1</sup>

<sup>1</sup> Wobei angenommen wird, dass kein Perpetum Mobile existiert...

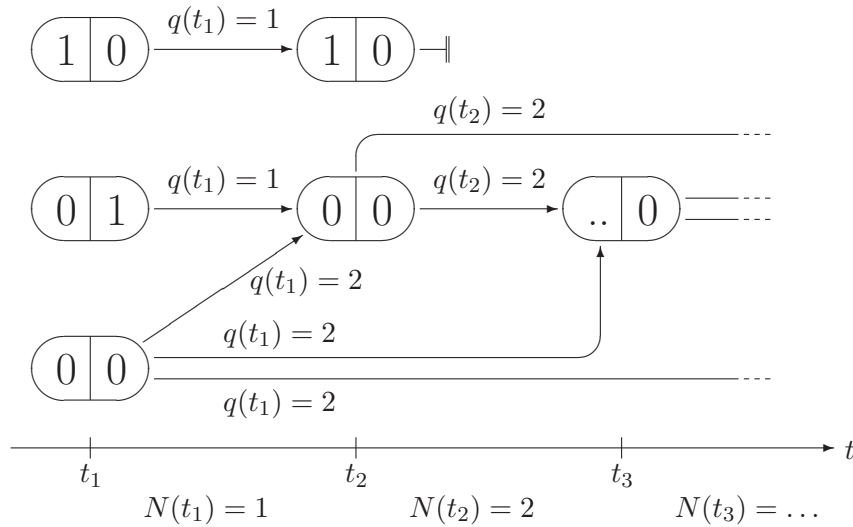


Abbildung 4.1: Knoten der Zeitpunkte  $t_i$ ,  $i = 1 \dots 3$ , mit allen möglichen Verbindungen bei gegebenen Nachfragen  $N(t_i)$  sowie der Anzahl verwendeter Wagen  $q(t_i)$ ,  $q_{tot} = 2$ .

Die Verbindungen werden anschliessend mit der Kostenfunktion  $k(\dots)$  gewichtet, um schliesslich mit dem Algorithmus von Dijkstra den kürzesten Weg vom Beginn des gegebenen Fahrplans bis zum Ende desselben zu suchen. Es bleibt zu zeigen, dass dieser Weg identisch mit dem optimalen *Rolling Stock Assignment* ist.

Damit nicht zuviele Knoten entstehen, also der Graph nicht zu gross wird, werden anfangs nur für den Zeitpunkt  $t_1$  alle möglichen Knoten erstellt. Für jeden dieser Knoten werden alle möglichen minimalen Verbindungen hin zu späteren Zeitpunkten  $t_i$ ,  $i > 1$  erstellt. Sollte der Zielknoten einer solchen Verbindung noch nicht vorhanden sein, so wird er erstellt. Anschliessend wird inkrementell bis zur letzten Fahrt des Fahrplans der Schritt  $t_i \mapsto t_{i+1}$  gemacht, wobei jeweils für alle Knoten des Zeitpunkts  $t_i$  alle möglichen minimalen Verbindungen nach späteren Zeitpunkten erstellt werden.

**Beispiel:** In der Abbildung 4.1 stehen  $q_{tot} = 2$  Wagen zur Verfügung. Zum Zeitpunkt  $t_1$  gibt es somit drei mögliche Startknoten  $(1, 0)$ ,  $(0, 1)$  und  $(0, 0)$ , denn diese Knoten repräsentieren die Depotkonfiguration nach Abfahrt des Zuges, welcher mindestens einen Wagen ziehen muss.

Für die erste Fahrt werden  $N(t_1) = 1$  Wagen benötigt. Wenn also vom Knoten  $(1, 0)$  mit nur einem Wagen nach  $B$  gefahren wird und der zweite Wagen im Depot  $A$  zurück bleibt, so stehen für die zweite Fahrt mit  $N(t_2) = 2$

nicht genügend Wagen zur Verfügung, was diesen Pfad im Graphen zu einer Sackgasse macht.

Vom Knoten  $(0, 1)$  aus geht nur gerade eine Verbindung mit  $q(t_1) = 1$  zum Knoten  $(0, 0)$  des Zeitpunkts  $t_2$ , denn für die zweite Fahrt werden  $N(t_2) = 2$  Wagen benötigt, was ein Rangieren am Bahnhof  $B$  erforderlich macht, um den sich dort im Depot befindlichen Wagen anzuhängen.

Ganz anders sieht es beim Knoten  $(0, 0)$  aus. Hier befinden sich anfangs beide Wagen im Bahnhof  $A$ . Weil für die zweite Fahrt beide Wagen gebraucht werden, muss auch bereits die erste Fahrt mit  $q(t_1) = 2$  Wagen ausgeführt werden, um den zweiten Wagen nach  $B$  zu bringen. Dies führt zwar zu einer Leerfahrt, ist aber unerlässlich. Von diesem Knoten aus können ausserdem Verbindungen zu allen späteren Zeitpunkten erstellt werden, was bedeutet, dass von Beginn weg bis zum Zeitpunkt des verbundenen Knotens alle verfügbaren Wagen von der Lokomotive gezogen werden.

## 4.4 Implementierung

Die Parameter für den Algorithmus sind auf dieselbe Weise wie beim ersten Algorithmus im Kapitel 3 definiert, allerdings werden nicht mehr alle davon benötigt. So berechnet der Algorithmus  $q_{tot}$  selber und die Bestimmung der Startkonfiguration der Depots ist Teil des Optimierungsvorganges.

```
c    Kapazität der Wagen.
N    var N=new Array(hin,her, ...);
     Das Array enthält die Nachfrage als Anzahl Passagiere für die
      $i$ -te Fahrt. Modulo 2 kann man die Fahrtrichtung berechnen.
```

### 4.4.1 Datenstruktur

Der Graph, welcher für den Algorithmus aufgebaut wird und im vorangehenden Kapitel beschrieben wurde, wird aus Objekten (*JavaScript*-Notation) zusammengesetzt. Es gibt drei verschiedene Objekt-Typen: *Zeitpunkt*, *Knoten* und *Verbindung*. In den Zeitpunkt-Objekten wird neben der Nachfrage und der minimal benötigten Anzahl Wagen ein Array von Knoten-Objekten gespeichert. Diese Knoten repräsentieren die oben beschriebenen Knoten und enthalten selbst ein Array mit Verbindungs-Objekten, welche je eine Verbindung zu einem Knoten eines späteren Zeitpunkts darstellen. Im Knoten-Objekt sind noch Verwaltungsdaten für den Dijkstra-Algorithmus (kursiv herausgehoben) enthalten.

```

1  function Zeitpunkt(Nachfrage)
2  { this.N=Nachfrage;           // Nachfrage.
3    this.min=Math.ceil(iNachfrage/c); // Min. Anz. Wagen.
4    this.Knoten=new Array();    // Knoten.
5  }
6
7  function Knoten(t,dA,dB)
8  { this.d=new Array(dA,dB);    // Depotzustände.
9    this.Verbindungen=new Array(); // Verbindungen.
10   this.Kosten=-1;
11   this.Vorgaenger=null;
12   this.istInRand=false;
13 }
14
15 function Verbindung(kNach,Q,R,L)
16 { this.nach=kNach;           // Zielknoten.
17   this.q=Q;                  // Fahrende Wagen.
18   this.r=R;                  // Rangierte Wagen.
19   this.l=L;                  // Leere Wagen.
20   this.Kosten=-1;           // Kosten der Ver-
21 }                             // bindung.

```

#### 4.4.2 Aufbau des Graphen

Um den Algorithmus von Dijkstra ausführen zu können, muss der Graph aus Abbildung 4.1 einen eindeutigen Start- und Endknoten haben. Er muss deshalb durch die zwei Knoten `kStart` und `kZiel` erweitert werden, welche wie in Abbildung 4.2 jeweils mit allen Knoten am entsprechenden Ende des Graphen verbunden sind. Ihre Depotzustände werden nicht ausgewertet und

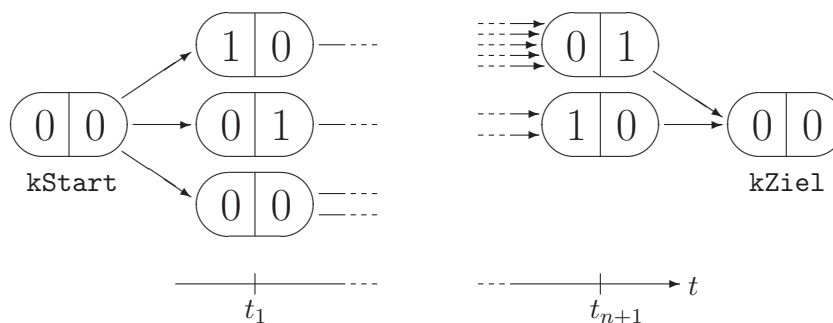


Abbildung 4.2: Graph ergänzt durch Start- und Endknoten

deshalb einfach auf  $(0,0)$  gesetzt. Man beachte, dass zum Zeitpunkt  $t_{n+1}$ <sup>2</sup> nicht unbedingt alle möglichen Depot-Konfigurationen als Knoten vorhanden sein müssen, denn es werden nur die möglichen minimalen Verbindungen und deren Zielknoten in den Graph eingefügt. Auch wird weder innerhalb des Graphen noch am Ende jemals einer der Knoten  $(q_{tot},0)$  oder  $(0,q_{tot})$  vorhanden sein, denn es muss, es sei nochmals wiederholt, bei jeder Fahrt mindestens ein Wagen am Zug angehängt sein.

```

1  var Zeitpunkte=new Array();           // Zeitpunkt-Objekte.
2  var kStart=new Knoten(-1,0,0);        // Startknoten.
3  var kZiel=new Knoten(-1,0,0);        // Zielknoten.
4  var n=N.length;                       // Anzahl Fahrten.
5  var qTotal=0;                          // Anzahl Wagen.
6  var i,j,q,r,t,v;
7  var min,leer;
8  var d=new Array(2);
9
10 function NeuerKnoten(t,dA,dB)         // Erstellt einen neuen
11 { var nK=new Knoten(t,dA,dB);         // Knoten und fügt ihn
12   Zeitpunkte[t].Knoten.push(nK);     // zum Zeitpunkt t in
13   return nK;                          // Knotenliste ein.
14 }
15
16 function GibKnoten(t,dA,dB)           // Gibt Knoten (dA,dB)
17 { var i,K;                             // zum Zeitpunkt t.
18   for(i=0;i<Zeitpunkte[t].Knoten.length;i++)
19   { K=Zeitpunkte[t].Knoten[i];
20     if(K.d[0]==dA&&K.d[1]==dB)return K;
21   }
22   return NeuerKnoten(t,dA,dB);        // Nicht vorhandenen
23 }                                       // Knoten erstellen.
24
25 function GibMinimumVektor(t)          // Gibt einen Vektor [0..n-t]
26 { var i;                                // mit der für die Fahrt
27   var m=new Array();                   // vom Zeitpunkt t nach t+i
28   m[0]=Zeitpunkte[t].min;             // minimal notwendigen An-
29   for(i=1;i<n-t;i++)                   // zahl Wagen zurück.
30   { m[i]=Math.max(m[i-1],Zeitpunkte[t+i].min);
31   }
32   return m;
33 }
34

```

<sup>2</sup> Der Zeitpunkt  $t_{n+1}$  repräsentiert das Ende des Fahrplans mit  $n$  Fahrten.

```

35 function GibLeerVektor(t,m) // Gibt einen Vektor [0..n-t]
36 { var i; // mit der kumulierten An-
37   var l=new Array(); // zahl Leerwagen bei einer
38   l[0]=0; // Fahrt vom Zeitpunkt t bis
39   for(i=1;i<n-t;i++) // zum Zeitpunkt t+i.
40   { l[i]=l[i-1]+(m[i]-m[i-1])*i+(m[i]-Zeitpunkte[t+i].min);
41   }
42   return l;
43 }
44
45 function ErstelleVerbindung(Von,Nach,d,q,r,l)
46 { var K=GibKnoten(Nach,d[0],d[1]);
47   var v=new Verbindung(K,Nach-Von,q,r,l);
48   return v;
49 }
50
51 for(t=0;t<n;t++) // Parameter in Zeit-
52 { Zeitpunkte[t]=new Zeitpunkt(N[t]); // punkte übernehmen.
53   if(Zeitpunkte[t].min>qTotal)qTotal=Zeitpunkte[t].min;
54 }
55 Zeitpunkte[n]=new Zeitpunkt(0); // Ziel der letzten Fahrt.
56
57 // Alle möglichen Depotzustände für Zeitpunkt t=1 erstel-
58 // len und mit Startknoten verbinden.
59 for(i=Zeitpunkte[0].min;i<=qTotal;i++)
60 { for(j=0;j<=qTotal-i;j++)NeuerKnoten(0,j,qTotal-i-j);
61 }
62 for(i=0;i<Zeitpunkte[0].Knoten.length;i++)
63 { v=new Verbindung(Zeitpunkte[0].Knoten[i],0,0,0,0);
64   kStart.Verbindungen.push(v);
65 }
66
67 for(t=0;t<n;t++)
68 { min=GibMinimumVektor(t);
69   leer=GibLeerVektor(t,min);
70   for(i=0;i<Zeitpunkte[t].Knoten.length;i++)
71   { for(j=1;j<n-t+1;j++)
72     { q=min[j-1];
73       if(q<=qTotal-Zeitpunkte[t].Knoten[i].d[(t+1)%2])
74       { if(t%2==0)
75         { d[1]=Zeitpunkte[t].Knoten[i].d[1];
76           d[0]=qTotal-d[1]-q;
77         }
78         else

```

```

79         { d[0]=Zeitpunkte[t].Knoten[i].d[0];
80           d[1]=qTotal-d[0]-q;
81         }
82         r=Math.abs(Zeitpunkte[t].Knoten[i].d[t%2]-d[t%2]);
83         v=ErstelleVerbindung(t,t+j,d,q,r,leer[j-1]);
84         Zeitpunkte[t].Knoten[i].Verbindungen.push(v);
85     }
86     else break;
87 } } }
88
89 // Alle Endknoten mit dem Zielknoten verbinden.
90 for(i=0;i<Zeitpunkte[n].Knoten.length;i++)
91 { v=new Verbindung(kZiel,0,0,0,0);
92   Zeitpunkte[n].Knoten[i].Verbindungen.push(v);
93 }

```

#### 4.4.3 Berechnung der Kosten

Bevor der Algorithmus zum Auffinden des kürzesten Weges zwischen dem Start- und dem Zielknoten gestartet werden kann, müssen noch die Kosten der Verbindungen berechnet werden. Es gibt drei mögliche Zeitpunkte, die Kostenfunktion  $k(\dots)$  auf die Verbindungen anzuwenden:

1. Während dem Aufbau des Graphen.
  - ⇒ Bei dieser Methode ist es nicht möglich, den Graphen für mehrere aufeinanderfolgende Berechnungen mit unterschiedlichen Kostenfunktionen wiederzuverwerten, ausser es wird zusätzlich eine der folgenden Möglichkeiten implementiert.<sup>3</sup>
2. Als zusätzliche Schleife zwischen dem Aufbau des Graphen und dem Algorithmus von Dijkstra.
  - ⇒ Diese Möglichkeit ist wiederverwendbar für eine Neuberechnung, benötigt aber einen Aufwand in der Grösse der Anzahl vorhandener Verbindungen, was sich auf die Rechenzeit auswirkt.
3. Der nachfolgend beschriebene Algorithmus von Dijkstra besucht jede Verbindung genau einmal, weshalb deren Kosten bei diesem Schritt vor dem Zugriff berechnet werden können.
  - ⇒ Weil auch bei jeder Neuberechnung der Algorithmus auszuführen ist, entsteht bei dieser Methode kein zusätzlicher Aufwand.
  - ⇒ Diese Methode ist vorzuziehen.

<sup>3</sup> Die auf dem Internet verfügbare Implementation funktioniert so (siehe Kapitel 7).

#### 4.4.4 Algorithmus von Dijkstra

Der Algorithmus von Dijkstra für die Suche des kürzesten Weges zwischen zwei Knoten eines Graphen wurde nach der Beschreibung in [1] implementiert. Dabei werden die Knoten des Graphen in drei Bereiche eingeteilt: Knoten mit bekanntem kürzestem Weg, Randbereich und noch nicht betrachtete Knoten.

Zu Beginn befindet sich der Startknoten `kStart` im Randbereich. Die zentrale Schleife wird solange abgearbeitet, bis sich kein Knoten mehr im Rand befindet. Es wird in jedem Durchlauf ein Knoten aus dem Rand genommen und bearbeitet. Für die Knoten, nach welchen eine Verbindung vom aus dem Rand entfernten Knoten ausgeht, werden die Weg-Kosten über den Knoten berechnet. Das ist die Summe der Kosten, um den aus dem Rand genommenen Knoten zu erreichen, und der Kosten für die betrachtete Verbindung. Sind diese Kosten kleiner als die aktuellen Kosten des verbundenen Knotens oder befindet sich dieser noch nicht im Rand, so werden ihm die Kosten zugewiesen und er wird in den Randbereich eingefügt.

```

1  var Rand=new Array(); // Randbereich während Abarbeitung.
2  var Weg=new Array(); // Vom Algorithmus gefundener Weg.
3  var i,K;
4
5  function GibMinKnoten() // Entfernt den Knoten mit minima-
6  { var i,minK; // len Kosten aus dem Randbereich
7    var j=0; // und gibt ihn zurück.
8    var n=Rand.length;
9    for(i=0;i<n;i++)if(Rand[i].Kosten<Rand[j].Kosten)j=i;
10   minK=Rand[j];
11   if(n>1&&j!=n-1)Rand.splice(j,1,Rand[n-1]);
12   Rand.pop();
13   minK.istInRand=false;
14   return minK;
15 }
16
17 kStart.Kosten=0; // Kürzester Weg beginnt beim
18 Rand.push(kStart); // künstlichen Startknoten.
19
20 while(Rand.length>0) // Alle Knoten im Rand bearbeiten
21 { K=GibMinKnoten(); // (Dijkstra-Algorithmus).
22   for(i=0;i<K.Verbindungen.length;i++)
23   { with(K.Verbindungen[i])
24     { if(nach.Vorgaenger!=null)
25       { if(nach.Kosten>K.Kosten+Kosten||nach.Kosten===-1)
26         { nach.Kosten=K.Kosten+Kosten;

```

```

27         nach.Vorgaenger=K;           // Weg ist billiger.
28         if(!nach.istInRand)         // Knoten in Rand ein-
29         { nach.istInRand=true;      // fügen, falls nicht
30           Rand.push(nach);         // bereits dort.
31     } } } } } } }
32
33     K=kZiel.Vorgaenger;              // Kürzesten Weg zusammenstellen
34     while(K.Vorgaenger!=null)       // → Beim Zielknoten beginnen,
35     { Weg.unshift(K);               // auf dem Weg zum Startknoten
36       K=K.Vorgaenger;              // die angetroffenen Knoten vorne
37     }                               // an das Resultat-Array anfügen.

```

## 4.5 Bemerkungen

**Theorem 7** Die erstellte Datenstruktur enthält die optimale Lösung als Pfad im Graphen.

**Beweis:**

1. Vom Startknoten aus gibt es eine Kante zu jeder Depot-Kombination  $(d_A, d_B)$ , welche die erste Fahrt minimal möglich macht.
  - ⇒ Optimale erste Fahrt ist enthalten.
2. Von jedem inneren Knoten werden Kanten zu allen nachfolgenden Zeitpunkten erstellt, welche jeweils die für diese Strecke minimale Anzahl Wagen mitführen.
  - ⇒ Optimaler Pfad ist enthalten. □

**Theorem 8** Der Algorithmus liefert immer eine Lösung, das heisst, es gibt immer einen Pfad, welcher den gesamten Graphen durchquert.

**Beweis:** Es wird immer eine Kante vom Zeitpunkt  $t_1$  zum Zeitpunkt  $t_n$  geben. Für diese Kante wird die gesamte Menge  $q_{tot}$  aller Wagen benötigt, denn es muss auch diejenige Fahrt bewältigt werden, welche diese Anzahl Wagen verlangt. □

**Theorem 9** Der Algorithmus findet die optimale Lösung des Rolling Stock Assignment.

**Beweis:** Die optimale Lösung entspricht demjenigen Weg durch den Graphen, welcher nach Anwendung der Kostenfunktion  $k(\dots)$  die geringsten Kosten aufweist. Der angewandte Algorithmus von Dijkstra findet eben diesen kürzesten Weg zwischen den zwei Endknoten (Beweis in [1]). □

---

**Beobachtung:** Wenn die Kosten für eine Leerfahrt viel grösser sind als die Kosten für das Rangieren, dann liefert der Algorithmus dieselbe Lösung wie der erste Algorithmus im Kapitel 3, denn es wird derjenige Weg durch den Graphen gewählt, welcher immer mit der minimalen Anzahl Wagen verkehrt.

**Beobachtung:** Es gibt Knoten, welche keine weiterführenden Verbindungen haben (z.B. der Knoten  $(1,0)$  zum Zeitpunkt  $t_2$  in Abbildung 4.1). Diese könnten als Optimierung für Dijkstra vorgängig gelöscht werden, wobei allerdings fragwürdig ist, ob sich dieser Aufwand gegenüber dem Mehraufwand für die Suche des kürzesten Weges auszahlt.



# 5

## Zwischendepots

### 5.1 Neue Situation

In diesem Kapitel wird die bisherige Bahnlinie von  $A$  nach  $B$  zuerst um ein Zwischendepot  $D$  erweitert, später als Verallgemeinerung auf  $d$  Depots.

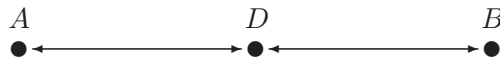
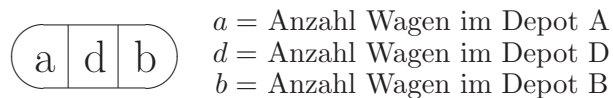


Abbildung 5.1: Bahnlinie mit Zwischendepot

Damit sind vorerst drei Depots vorhanden, je eines bei den beiden Endbahnhöfen  $A$  und  $B$  und das dritte in  $D$ . Der Zug fährt auch weiterhin immer die ganze Strecke von  $A$  nach  $B$  und wieder nach  $A$  zurück, er kann aber bei  $D$  je nach Nachfrage Wagen stehen lassen und später wieder mitnehmen.

### 5.2 Implementierung

Der Algorithmus aus dem letzten Kapitel kann für diese neue Situation grösstenteils übernommen werden, es sind nur wenige Anpassungen notwendig. Die augenfälligste Anpassung ist, dass jeder Knoten neu die Zustände von drei Depots darstellen muss:



In der Implementierung führt das zu einer Neudefinition des Knoten-Objektes, welches jetzt ein Array mit drei Depotzuständen enthalten muss:

```

1  function Knoten(t,dA,dD,dB)
2  { this.d=new Array(dA,dD,dB);          // Depotzustände.
  :
7  }
```

Eine zweite Änderung betrifft die Interpretation der Zeit  $t$  in den diversen Schleifen des besprochenen Algorithmus. Bisher wurde die Fahrtrichtung durch Modulo-Arithmetik als  $t \bmod 2$  bestimmt. Neu muss  $t$  folgendermassen interpretiert werden:

$$\begin{aligned}
 t \bmod 4 = 0 : & \quad A \rightarrow D \\
 = 1 : & \quad D \rightarrow B \\
 = 2 : & \quad D \leftarrow B \\
 = 3 : & \quad A \leftarrow D
 \end{aligned}$$

Die Zuweisungen der Depotzustände müssen im Algorithmus entsprechend angepasst werden.

Eine letzte Anpassung betrifft die Anzahl möglicher Startkonfigurationen  $\#_3(q_{tot})$  für drei Depots und  $q_{tot}$  Wagen. Die allgemeine Formel für  $\#_d(q_{tot})$  wird im Kapitel 5.3 hergeleitet werden, hier sei nur das Resultat für drei Depots gegeben:

$$\begin{aligned}
 \#_3(q_{tot}) &= \prod_{i=1}^{q_{tot}-1} \frac{i+3}{i} = \frac{1}{(q_{tot}-1)!} \cdot \frac{(q_{tot}+2)!}{6} \\
 &= \frac{1}{3} \cdot q_{tot} + \frac{1}{2} \cdot q_{tot}^2 + \frac{1}{6} \cdot q_{tot}^3
 \end{aligned}$$

### 5.3 Verallgemeinerung auf $d$ Depots

Ebenso wie der Algorithmus für zwei Depots auf drei Depots erweitert wurde, kann man auch beim allgemeinen Fall mit  $d$  Depots,  $d > 1$ , vorgehen. Das Knoten-Objekt muss dann die Zustände der  $d$  Depots repräsentieren, die Interpretation der Zeitpunkte kann analog zur obigen Erweiterung mit einer Fallunterscheidung für  $t \bmod (2 \cdot (d-1))$  eingeführt werden.

Der interessantere Aspekt und wesentliche Faktor für den Berechnungsaufwand ist die Anzahl Startknoten  $\#_d(q_{tot})$  für  $d$  Depots und  $q_{tot}$  Wagen im Graph. Die Fragestellung nach dieser Zahl entspricht folgendem kombinatorischem Problem (nach [2]):

*Wieviele verschiedene Möglichkeiten gibt es,  $q_{tot}$  Kugeln auf  $d + 1$  Körbe zu verteilen, wobei immer mindestens eine Kugel im ersten Korb sein muss?*

Die  $d + 1$  Körbe entsprechen den  $d$  Depots plus dem Zug, welcher ebenfalls als „Depot“ angesehen werden kann. Die erwähnte Kugel, welche immer im ersten Korb sein muss, entspricht demjenigen Wagen, welcher immer mindestens am Zug angehängt sein muss, wenn der Zug fährt. Die Anzahl möglicher Startknoten  $\#_d(q_{tot})$  ist damit gegeben durch:

$$\begin{aligned} \#_d(q_{tot}) &= \prod_{i=1}^{q_{tot}-1} \frac{i+d}{i} \quad \forall q_{tot} > 1, d > 0 \\ &= \frac{1}{(q_{tot}-1)!} \cdot \frac{(q_{tot}+d-1)!}{d!} \quad (5.1) \\ \#_d(1) &= 1 \end{aligned}$$

In Zahlen ausgedrückt ergibt das folgende Tabelle:

$\#_d(q_{tot})$	$d=1$	$d=2$	$d=3$	$d=4$	$d=5$
$q_{tot}=1$	1	<b>1</b>	1	1	1
$q_{tot}=2$	2	<b>3</b>	4	5	6
$q_{tot}=3$	3	<b>6</b>	10	15	21
$q_{tot}=4$	4	<b>10</b>	<b>20</b>	35	56
$q_{tot}=5$	5	15	35	70	126
$q_{tot}=6$	6	21	56	126	252

Die Beobachtung dieser Zahlenwerte führt zu einer Summenformel, welche anstatt der Gleichung 5.1 zur Berechnung der maximalen Anzahl Startknoten benutzt werden kann:

$$\begin{aligned} \#_1(q_{tot}) &= q_{tot} \\ \#_d(1) &= 1 \\ \#_d(q_{tot}) &= \sum_{i=1}^{q_{tot}} \#_{d-1}(i) \quad \forall q_{tot} > 1, d > 1 \end{aligned}$$

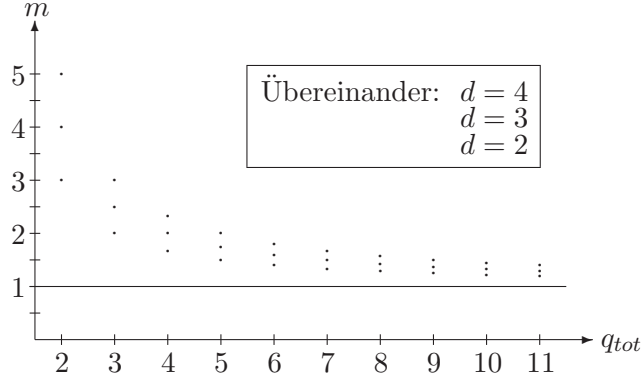


Abbildung 5.2: Vergleich des Wachstums der Anzahl möglicher Startknoten bei sich ändernder Anzahl Wagen oder Depots für  $d = 2 \dots 4$

**Beispiel:** Der Wert 20 für  $q_{tot} = 4$  und  $d = 3$  in der Tabelle entspricht der Summe der Werte für  $q_{tot} = 1 \dots 4$  und  $d = 2$ .

Ebenfalls interessant ist eine Abschätzung der Gesamtzahl  $\#_d^{tot}$  aller Knoten. Falls der Fahrplan  $n$  Fahrten vorsieht, so beträgt diese Zahl im schlimmsten Fall

$$\#_d^{tot}(n) = (n + 1) \cdot \#_d(n) + 2$$

Der Faktor beträgt  $(n + 1)$ , weil jede Fahrt respektive Verbindung von einem Knoten zu einem anderen geht, was bedingt, dass wie bei den Latten eines Zauns eine Abschlussmenge von Knoten vorhanden ist. Die zwei zusätzlichen Knoten sind **kStart** und **kZiel**.

Bei realen Daten wird dieser schlimmste Fall allerdings niemals erreicht werden, denn normalerweise wird ein wesentlicher Anteil der Wagen in Betrieb sein, was die Anzahl Knoten massiv reduziert.

Es kann noch betrachtet werden, wie stark die Anzahl Startkombinationen bei einer Änderung der Anzahl Wagen wächst. Dieses Wachstum ist in Abbildung 5.2 für  $d = 2 \dots 4$  folgendermassen dargestellt:

$$\#_d(q_{tot}) = m \cdot \#_d(q_{tot} - 1)$$

Man sieht leicht, dass dieses Wachstum nicht polynomiell ist. Das *Rolling Stock Assignment* ist denn auch nach [3] und [4] in NP enthalten.

# 6

## Zusammenfassung

### 6.1 Rückblick

In dieser Semesterarbeit werden drei Algorithmen für verschiedene Komplexitätsstufen des *Rolling Stock Assignment* präsentiert. Schon zu Beginn der Arbeit war klar, dass es sich dabei um ein schwieriges Problem handelt. Aus diesem Grund wurde mit einfachsten Überlegungen begonnen, was sogar noch zu einem ersten Algorithmus mit linearer Laufzeit führte.

Bereits die erste Erweiterung um die Kostenfunktion machte dann aber klar, wie schnell man mit einer Fragestellung in die Klasse der nicht-polynomiellen Probleme rutschen kann, auch wenn man nur eine „Kleinigkeit“ hinzufügt. Der zur Optimierung der Kosten gewählte graphische Ansatz konnte dann aber auch die Erweiterung um Zwischendepots mit wenigen Anpassungen bewältigen. Zusätzlich wurde gezeigt, dass damit eine Verallgemeinerung auf  $d$  Depots ebenso einfach möglich ist.

Die Implementierungen der vorgestellten Algorithmen bewiesen schliesslich, dass auch ein nicht-polynomielles Problem sogar mit einer interpretierten Programmiersprache bei realistischem Datenumfang in weniger als einer Sekunde berechnet werden kann. Das lässt die Hoffnung zu, dass auch die im Kapitel 6.2 beschriebenen Erweiterungen mit einer Implementierung in einer Hochsprache in nützlicher Zeit berechnet werden können. Der gewählte graphische Ansatz dürfte auch dort erfolgversprechend sein.

### 6.2 Ausblick

Zusätzlich zu den in dieser Arbeit betrachteten Aspekten gibt es noch zahlreiche mögliche Erweiterungen des *Rolling Stock Assignment* Problemkreises, welche aber den Rahmen einer Semesterarbeit sprengen würden.

Weil Bahngesellschaften meist mehrere Bahnlinien besitzen (wie in Abbildung 6.1), wäre zum Beispiel naheliegend, das Rollmaterial je nach Nach-

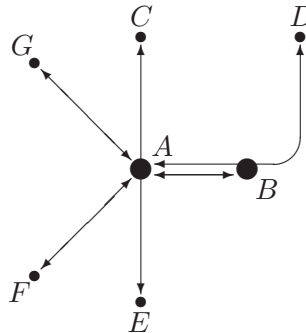


Abbildung 6.1: Bahnnetz

frage zwischen den einzelnen Linien des Bahnnetzes zu tauschen. Dabei muss aber sogleich die Frage gestellt werden, ob sich diese zusätzliche Komplexität auszahlt. Die Nachfragespitzen treten bei einem Langzeit-Fahrplan normalerweise auf dem gesamten Netz zum selben Zeitpunkt auf, also zum Beispiel durch den Pendlerverkehr bedingt am Morgen und am Abend. Es werden also zur selben Zeit auf dem gesamten Streckennetz viele Wagen benötigt und anschliessend überall wieder in die Depots gestellt.

Interessanter kann der Ansatz der Vernetzung aller Bahnlinien bei der Online-Planung sein, wenn kurzfristig eine Nachfrageänderung eintritt, zum Beispiel zu Ferienbeginn oder bei Streckenunterbrüchen, und darauf schnell reagiert werden muss. Diese ausserordentlichen Passagieraufkommen werden erfahrungsgemäss mit Extrazügen abgewickelt. Dafür müssen genügend Wagen vorhanden sein, weshalb solche unter Umständen ausser Plan von anderen Bahnhöfen respektive deren Depots herbeigeschafft werden müssen. Wenn die Planung das gesamte Netz berücksichtigt, so können nicht gebrauchte Wagen aufgefunden und für den Extrazug bereitgestellt werden. Zu einem späteren Zeitpunkt werden diese herbeigeholten Wagen nach Fahrplan am anderen Bahnhof aber wieder gebraucht, was zeigt, dass eine solche Online-Planung nicht nur das ganze Bahnnetz berücksichtigt, sondern sich auch auf dasselbe auswirkt.

Noch viel aufwendiger werden die Berechnungen, wenn jeder Wagen und jede Lokomotive der Bahngesellschaft eine Identität bekommt und somit gesondert behandelt werden kann. Dadurch wird es möglich, Routinekontrollen des Rollmaterials, welche es erforderlich machen, einen bestimmten Wagen oder eine Lokomotive für einige Zeit aus dem Betrieb zu entfernen, in die Planung einzubeziehen.

Offensichtlich sind der Phantasie kaum Grenzen gesetzt. In [3] wird ein Überblick über verschiedenste Lösungsansätze für das *Rolling Stock Assignment* gegeben, welche für die oben beschriebenen stark erweiterten Problemstellungen Lineare Programmierung verwenden.

# 7

## Quelltexte der Algorithmen

Die Quellen der in dieser Arbeit beschriebenen Algorithmen sind auf dem Internet verfügbar. Es handelt sich dabei um Hypertext-Dokumente, welche den Programmcode in JavaScript<sup>1</sup> enthalten. Getestet wurde die Ausführbarkeit mit dem Microsoft Internet Explorer 5.5, dem Netscape Communicator 4.73 und mit Netscape 6. Mit älteren Browsern funktionieren die Algorithmen nur teilweise, liefern falsche Resultate oder funktionieren erst gar nicht.

Folgende Adresse führt zu den erwähnten Dokumenten:

<http://www.shima.ch/papers/>

Unter derselben Adresse ist ausserdem diese Arbeit als PDF-Dokument verfügbar.

Eine gute Einführung in JavaScript gibt [5].

---

<sup>1</sup> Die Implementierung wurde in *JavaScript* vorgenommen, weil diese Programmiersprache sehr eng mit *C* und *Java* verwandt ist, und weil die berechneten Daten innerhalb eines HTML-Dokumentes sehr einfach in den verwendeten Browser ausgegeben und somit auch komfortabel betrachtet und ausgedruckt werden können, was für die Demonstration eines Prototypen ideal ist.



# Literaturverzeichnis

- [1] Peter Widmayer und Thomas Ottmann.  
*Algorithmen und Datenstrukturen.*  
BI-Wissenschaftsverlag, 1993.
- [2] John A. Rice.  
*Mathematical Statistics and Data Analysis.*  
Duxbury Press, Belmont CA, 1995.
- [3] U. T. Zimmermann M. R. Bussieck, T. Winter.  
*Discrete optimization in public rail transport.*  
TU Braunschweig, Abteilung für Mathematische Optimierung, 1997.  
<http://www.math.tu-bs.de/mo/ismp.html>.
- [4] G. Gallo A.A. Bertossi, P. Carraresi.  
*On some matching problems arising in vehicle scheduling models.*  
Networks, 17: 271-281, 1987.
- [5] Stefan Münz.  
*SELFHTML - HTML-Dateien selbst erstellen.*  
<http://www.teamone.de/selfaktuell>.

# Index

- Algorithmen
  - Dijkstra, 23
  - Erster Algorithmus, 9
  - Kostenrechnung, 16
  - Quelltexte, 33
  - Zwischendepot, 27
- Bahnlinien
  - Einfache Bahnlinie, 3
  - Zwischendepot, 27
- Bahnnetz, 32
- Berechnungsaufwand, 29
- Depot
  - Knoten-Repräsentation, 16, 28
  - Verallgemeinerung, 28
  - Zwischendepot, 27
- Fahrplangestaltung, 3
- Fahrtrichtung, 28
- Graphischer Ansatz, 16
  - Datenstruktur, 18
- Internet, 33
- JavaScript, 33
- Knoten
  - Anzahl Startknoten, 28, 29
  - Gesamtzahl, 30
  - Wachstumsbetrachtung, 30
- Kosten
  - Kostenarten, 1
  - Kostenfunktion, 15
  - Kostenträger, 4
  - Modellierung, 15
- Leerfahrt, 10
- Lineare Programmierung, 32
- Nachfrage, 1, 7
- NP, 30
- Objekte
  - Knoten, 19, 28
  - Verbindung, 19
  - Zeitpunkt, 19
- Online-Planung, 32
- Pendlerverkehr, 32
- Problemstellung, 1
  - Weitere Lösungsansätze, 32
- Quelltexte, 33
- Rangieren, 4
  - Kostenträger, 4
  - Varianten, 5
- Rolling Stock Assignment, 1
- Startknoten, 29
- Startkonfiguration, 13, 28, 29
- Verbindungen, 16
- Zwischendepot, 27