Technische Universität München

Fakultät für Mathematik
Lehrstuhl für Angewandte Geometrie und Diskrete Mathematik

Implementierung eines Modells für die Wartungsplanung in Verkehrsnetzen

Bachelorarbeit von Andreas Schlattl

Themensteller:	Prof. Dr. Raymond Hemmecke
Betreuer:	Dr. Michael Ritter
Abgabedatum:	11.06.2014

Hiermit erkläre ich, dass ich diese Arbeit selbstständig angefertigt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Garching, den 11.06.2014

Andreas Schlattl

Zusammenfassung

Diese Arbeit beschäftigt sich mit einem Modell zur optimalen Terminplanung von Wartungsarbeiten für Bauwerke in der Verkehrsinfrastruktur. Diese Modell wurde in einer Vorgängerarbeit [Ell12] aufgestellt und in Xpress-Mosel implementiert. Der dabei verwendete nichtlineare Solver konnte allerdings keine diskreten Probleme lösen, was dafür aber notwendig gewesen wäre. In der vorliegende Arbeit wurde das Modell in C++ implementiert, wobei ein Solver verwendet wurde, der nichtlineare, gemischt-ganzzahlige Probleme lösen kann. Das erstellte Optimierungsprogramm wird erklärt und an Beispielen getestet.

Abstract

This thesis deals with a model for optimal scheduling for maintenance of traffic infrastructure buildings. The model was formed in a previous thesis, [Ell12], where it was implemented in Xpress-Mosel. The nonlinear solver that was used there wasn't able to solve discrete problems. The present thesis is concerned with the implementation of the model in C++, while a nonlinear, mixed-integer solver is used. The developed optimization program is explained and tested by using some examples.

Inhaltsverzeichnis

1 Einleitung **1**

2 Theoretische Grundlagen **3**
2.1 Lösungsverfahren in der Optimierung 3
2.1.1 Innere-Punkte-Verfahren 3
2.1.2 Column Generation in der linearen Optimierung 4
2.1.3 Branch-and-Bound für binäre Probleme 6
2.1.4 Branch-and-Price 8
2.2 Verkehrsmodell 9
2.2.1 Mathematische Beschreibung des Verkehrsmodells 9
2.2.2 Systemoptimum 10
2.2.3 Nutzeroptimum 11
2.2.4 Beispiel von Pigou und Paradoxon von Braess 11
2.2.5 Zusammenhang zwischen Systemoptimum und Nutzeroptimum . . 13
2.2.6 Verkehrsflussproblem 14

3 Modellierung des Wartungszeitproblems **17**
3.1 Beschreibung der Parameter 17
3.2 Berechnung der Fahrzeit für eine Sperrkombination 20
3.3 Ermittlung einer Anfangssperrkombination 21
3.4 Wartungszeitproblem 22
3.4.1 Formulierung als Optimierungsproblem 22
3.4.2 Restrictet Masterproblem 24
3.4.3 Pricing-Problem 25

4 Implementierung und Tests **29**
4.1 Generierung der nichtlinearen Unterprobleme 29
4.1.1 Vorbereitung der Generierung 30
4.1.2 Generierung des Verkehrsflussproblems 33
4.1.3 Generierung des Pricing-Problems 36
4.2 Funktionsweise des Optimierungsprogramms 38
4.3 Handhabung der Software 39
4.4 Tests an Beispielen 40
4.4.1 Test am ersten Beispiel 40
4.4.2 Test am zweiten Beispiel 42

5 Zusammenfassung und Fazit **45**

Abbildungsverzeichnis **47**

Literaturverzeichnis **49**

1 Einleitung

Welcher Autofahrer kennt das nicht: Der Winter ist endlich vorbei, es geht keine Gefahr mehr von eisglatten Fahrbahnen aus, das aggressive Streusalz verschwindet von den Straßen und es ist wieder länger hell am Abend. Die warme Jahreszeit könnte so schön sein für Autofahrer, wenn es da nicht ein ganz bestimmtes Problem gäbe: Baustellen. Jedes Jahr sorgen sie für Staus und Umleitungen. Viele Autofahrer verbringen mehr Zeit in ihrem Fahrzeug als ihnen lieb ist. Dabei sind diese Wartungsmaßnahmen notwendig, um sicherzustellen, dass die Verkehrsinfrastruktur auch in Zukunft noch für den Verkehr benutzbar ist. Werden wichtige Bauarbeiten versäumt, kann es zu Vollsperrungen kommen, was möglicherweise erhebliche Auswirkungen auf den Straßenverkehr nach sich zieht. Infrastrukturbauwerke sollten also stets rechtzeitig gewartet werden. Es stellt sich vielmehr die Frage, wie man die Wartungsmaßnahmen so aufeinander abstimmen kann, dass negative Auswirkungen auf den Verkehr möglichst gering sind, oder gar nicht erst auftreten. Wie sollten diese Bauarbeiten örtlich und zeitlich verteilt werden, damit der, zumindest teilweise, umgeleitete Straßenverkehr zu keiner oder möglichst wenig Überlastung auf anderen Strecken führt?
Die vorliegende Arbeit beschäftigt sich mit dieser Frage. In der Diplomarbeit von Susanne Ellßel wurde bereits ein Modell aufgestellt, mit dem diese Frage beantwortet werden soll [Ell12]. Dabei geht es darum, einen Zeitplan für Wartungsmaßnahmen aufzustellen, der für möglichst wenig negative Auswirkungen auf den Verkehr sorgt. Beim Aufstellen eines Wartungszeitplans müssen dabei gewisse Dinge beachtet werden. So müssen alle Infrastrukturbauwerke bis zu einem bestimmten Zeitpunkt repariert sein, da sie ansonsten möglicherweise nicht mehr für den Straßenverkehr benutzbar sind. Außerdem muss darauf geachtet werden, dass das Budget und die Ressourcen, also Arbeitskräfte und Baumaschinen, begrenzt sind. Das Modell wurde in dieser Arbeit bereits in Xpress-Mosel implementiert. Dabei wurde ein nichtlinearer Solver verwendet, der nicht mit gemischt-ganzzahligen Problemen umgehen kann, was für manche Unterproblem notwendig wäre. Das Ziel der vorliegende Arbeit ist es, das Modell in C++ zu implementieren. Dabei wurde ein Optimierungsprogramm geschrieben, das auf die nichtlinearen Solver Ipopt und `BONMIN` zurückgreift, wobei letzterer auch gemischt-ganzzahlige Probleme lösen kann. Zuerst erklärt diese Arbeit einige der notwendigen theoretischen Grundlagen. Dann wird im dritten Kapitel das zu implementierende Modell erklärt und im vierten Kapitel geht es um die Implementierung selbst. Dabei wird zunächst die Software erklärt und dann einige Testbeispiele. Im letzten Kapitel gibt es noch eine kurze Zusammenfassung der Arbeit. Außerdem liegt der Arbeit eine CD bei, auf der sich unter anderem die erstellten Quellcodes und eine pdf-Version der vorliegenden Arbeit befinden. Näheres dazu steht in der Datei `liesmich.txt`.

2 Theoretische Grundlagen

2.1 Lösungsverfahren in der Optimierung

2.1.1 Innere-Punkte-Verfahren

Gegeben sei folgendes, nichtlineares Optimierungsproblem, mit konvexen Funktionen $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$ und $h : \mathbb{R}^n \to \mathbb{R}^p$:

$$\begin{aligned} &\min f(x) \\ &x \in X = \{x \in \mathbb{R}^n,\ g(x) \leq 0,\ h(x) = 0\} \end{aligned} \tag{2.1}$$

Neben den Penalty-Verfahren, bei denen zur Zielfunktion, außerhalb des zulässigen Bereichs Strafterme addiert werden, und den SQP-Verfahren, bei denen quadratische Unterprobleme gelöst werden, können auch sogenannte Innere-Punkte-Verfahren zur Lösung von (2.1) verwendet werden. Die Softwarebibliothek IPOPT (**I**nterior-**P**oints-**OPT**imizer), welche häufig zur Lösung von stetigen, nichtlinearen Problemen verwendet wird, arbeitet mit Innere-Punkte-Verfahren. Die grobe Idee hinter diesen Verfahren ist es, ähnlich wie bei Penalty-Verfahren, Strafterme zur Zielfunktion hinzu zu addieren. Ein wesentlicher Unterschied zu Penalty-Verfahren besteht jedoch darin, dass die Strafterme nicht erst außerhalb des zulässigen Bereichs wirken, sondern im Inneren bereits dafür sorgen, dass x dem Rand des durch g festgelegten Bereichs nicht zu nahe kommt. Die durch h bestimmten Gleichungsnebenbedingungen erzeugen keine Strafterme. Stattdessen können diese Nebenbedingungen durch Penalty-Terme behandelt werden, die erst außerhalb des zulässigen Bereichs wirken.

Vorgehensweise

Das ursprüngliche Problem (2.1) wird etwas umgeformt, wobei die neue Zielfunktion als (logarithmische) Barriere-Funktion bezeichnet wird. Der Logarithmus sorgt dafür, dass Punkte, die nahe an den Rand des durch $g(x) \leq 0$ bestimmten Bereichs gehen, Werte erhalten, die sehr groß sind, bzw. sogar gegen unendlich konvergieren. Außerdem kommen nur noch Gleichungsrestriktionen vor. Die Ungleichungsnebenbedingungen werden dabei mit Hilfe von Schlupfvariablen $s(x) \in \mathbb{R}^m$ in Gleichungsnebenbedingungen umgewandelt. Um die Strafterme der Barriere-Funktion zu gewichten wird ein Parameter $\psi \geq 0$ verwendet. Das Barriere-Problem lautet:

$$\begin{aligned} &\min \left(f(x) - \psi \sum_{i=1}^{m} \ln(s_i(x)) \right) \\ &g(x) - s(x) = 0 \\ &h(x) = 0 \end{aligned} \tag{2.2}$$

Auf dieses Problem kann, zum Beispiel ein SQP-Verfahren angewandt werden. Nachdem das Barriere-Problem gelöst ist, wird der Parameter ψ verringert und die gefundene Lösung dient als Startpunkt, um (2.2) erneut zu lösen. Dies wird so lange wiederholt, bis die Lösung die gewünschte Genauigkeit erreicht hat. Dazu kann man die Norm des Gradienten der Zielfunktion ermitteln. Ist diese besonders klein, bzw. unter einer bestimmten Schranke, so ist anzunehmen, dass keine bedeutenden Verbesserungen der Lösung mehr möglich sind.

2.1.2 Column Generation in der linearen Optimierung

In diesem Abschnitt wird ein Verfahren vorgestellt, um das folgende Optimierungsroblem, mit $m,\ n \in \mathbb{N},\ A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^n$, sowie $x,\ c \in \mathbb{R}^n$ zu lösen.

$$\begin{aligned} \min c^T x \\ Ax = b \\ x \geq 0 \end{aligned} \tag{2.3}$$

Man kann davon ausgehen, dass A vollen Zeilenrang hat, da es andernfalls redundante Nebenbedingungen gäbe oder der zulässige Bereich leer wäre. Um das Verfahren einfacher zu beschreiben, wird davon ausgegangen, dass der zulässige Bereich beschränkt ist. Außerdem sei angemerkt, dass sich jedes lineare Optimierungsroblem wie (2.3) darstellen lässt. Wie man Optimierungsprobleme entsprechend umwandelt wird in [Ulb12] erklärt. Die obige Darstellung wurde gewählt, da sich damit das Verfahren der Column Generation teilweise ähnlich wie das revidierte Simplex-Verfahren erklären lässt.
Column Generation wird meistens auf Probleme angewendet, bei denen es viele Variablen gibt, aber vergleichsweise wenige Gleichungsnebenbedingung. Die Anzahl der Variablen, deren Wert in der Lösung ungleich null ist, entspricht genau dem Wert der Gleichungsnebenbedingung, nämlich m. Alle anderen Variablen sind in der Lösung gleich null. Dies nutzt man beim Column Generation-Verfahren aus, indem man nur über einen Teil der Variablen optimiert. Dieses verkleinerte Optimierungsproblem nennt man **Restricted Masterproblem** (RMP), während das ursprüngliche Problem einfach **Masterproblem** genannt wird. Da zu Beginn die Lösung des RMP meist nicht mit der Lösung des Masterproblems gleich ist, müssen gegebenenfalls weitere Variablen hinzugefügt werden. Um zu entscheiden, welche Variablen zum RMP hinzugefügt werden sollten, wird ein weiteres Optimierungsproblem gelöst, das sogenannte Pricing-Problem. Sobald man keine Variable mehr findet, die zur Verbesserung des Zielfunktionswerts beitragen kann, ist die aktuelle Lösung des RMP auch für das Masterproblem optimal.

Vorgehensweise

Zu Beginn wird eine Startlösung benötigt. Damit ist ein, für (2.3) zulässiger Punkt $x_{\text{Start}} \in \mathbb{R}^n$ gemeint, der bis zu m Komponenten ungleich null enthält. Die Menge

$B \subset \{1, \ldots, n\}$ beinhaltet genau die Indizes dieser Komponenten. Wichtig ist auch, dass die Menge der Spalten von A, die zu den Indizes aus B gehören linear unabhängig ist. Die Matrix A_B, die sich daraus ergibt, ist eine Basismatrix des $\mathbb{R}^m$, weswegen B auch als **Basis** bezeichnet wird. Um eine Startlösung zu finden, wird meistens ein Hilfsproblem vorab gelöst. Die zu den Indizes von B gehörenden Komponenten von x_{Start}, bzw. c werden in x_B, bzw. c_B zusammengefasst. Für x_B gilt $A_B x_B = b$ und damit $x_B = A_B^{-1} b$. Später, im Pricing-Problem, spielen die sogenannten **reduzierten Kosten** eine wichtige Rolle. Darunter versteht man einen Vektor $\bar{c}$ der Länge $n - |B|$. Jede Komponente c_j beschreibt, um wie viel sich der Zielfunktionswert erhöht, wenn sich der Wert einer Nichtbasisvariable, $x_j,\ j \in \{1, \ldots, n\} \setminus B$ um eins erhöht. Wichtig ist vor allem das Vorzeichen von $\bar{c}_j$, da sich nur bei einem negativen $\bar{c}_j$ der Zielfunktionswert verbessern kann. Wird der Wert von x_j erhöht, so muss zu $A_B x_B$ der Vektor $A_j x_j$ hinzu addiert werden. A_j ist dabei die j-te Spalte von A. Damit die Gleichheitsnebenbedingung weiterhin erfüllt ist, wird $A_B^{-1} A_j x_j$ von x_B subtrahiert. Oder anders ausgedrückt:

$$b = A_B x_B = A_B \left(x_B - A_B^{-1} A_j x_j \right) + A_j x_j$$

$\bar{c}_j$ ist die Differenz zwischen dem neuen Zielfunktionswert z_{neu}, mit entsprechend veränderten Werten für x und dem alten Zielfunktionswert z_{alt}. z_{neu} ist die Summe aus $c_j \cdot x_j$, also dem, was das neue x_j zum Zielfunktionswert beiträgt und dem, was die alten, in ihrem Wert veränderten Basisvariablen beitragen. Man erhält:

$$\begin{aligned} \bar{c}_j &= z_{\text{neu}} - z_{\text{alt}} \\ &= c_j x_j + c_B^T \left(x_B - A_B^{-1} A_j x_j \right) - c_B^T x_B \\ &= x_j \cdot \left(c_j - c_B^T A_B^{-1} A_j \right) \end{aligned}$$

Setzt man nun $x_j = 1$, so erhält man $\bar{c}_j$, wie es oben beschrieben wurde:

$$\bar{c}_j = c_j - c_B^T A_B^{-1} A_j \tag{2.4}$$

Es gibt auch noch die Möglichkeit, die reduzierten Kosten mit den dualen Variablen zu ermitteln. Diese Methode wird in dieser Arbeit auch angewandt. Das zum Minimierungsproblem (2.3) duale Problem hat folgende Form:

$$\begin{aligned} &\max b^T y \\ &A^T y \leq c \end{aligned} \tag{2.5}$$

Zu jeder Variable des ursprünglichen Problems gehört eine Nebenbedingung des dualen Problems. Aus diesem Grund gibt es zu jeder Basisvariablen von (2.3) eine Basisnebenbedingung im dualen Problem. Diese Basisnebenbedingungen müssen mit Gleichheit erfüllen sein, so dass gilt:

$$A_B^T y_B = c_B$$

Daraus folgt:

$$y_B^T = c_B^T A_B^{-1}$$

In (2.4) eingesetzt erhält man für die reduzierten Kosten:

$$\bar{c}_j = c_j - y_B^T A_j \tag{2.6}$$

Das Besondere an der Column Generation ist, dass nicht alle reduzierten Kosten berechnet werden. Dies wäre, wegen der Größe des Problems zu aufwendig, oder in manchen Fällen gar nicht möglich. Stattdessen wird ein weiteres Optimierungsproblem formuliert, um die aktuell bestmögliche Kostensenkung zu finden. Dabei handelt es sich um das oben erwähnte Pricing-Problem:

$$\min_j \left\{ c_j - y_B^T A_j \ : \ j \in \{1, \ldots, n\} \setminus B \right\} \tag{2.7}$$

Das Pricing-Problem hat außerdem Nebenbedingungen, die verhindern sollen, dass Variablen in das RMP aufgenommen werden, die für das Problem (2.3) ungeeignet sind. Die Nebenbedingungen des Masterproblems oder entsprechend angepasste Varianten können dafür verwendet werden. Zusätzliche Spalten werden nur dann ins RMP aufgenommen, wenn die dazugehörigen reduzierten Kosten negativ sind. Es können dabei auch mehrere Spalten aufgenommen werden. Wenn die Lösung des Pricing-Problems größer oder gleich null ist, gibt es keine Variablen mehr, die zur Verbesserung des Zielfunktionswerts beitragen können und somit ist die Lösung des Masterproblems auch die Lösung des ursprünglichen Problems. Damit die Anwendung der Column Generation überhaupt Sinn macht, ist es notwendig, dass sich das Pricing-Problem effizient lösen lässt.
Hat man neue Variablen zum RMP hinzugefügt, geht es in den nächsten Iterationsschritt. Dort wird das neue RMP gelöst, die Basis B angepasst, die Dualvariablen y_B gesucht und schließlich das neue Pricing-Problem gelöst. Die Iteration bricht ab, wenn die Lösung des Pricing-Problems größer oder gleich null ist. Die Lösung des aktuellen RMPs wird ausgegeben. Wie ein Columnn Generation Algorithmus grob aussehen könnte wird in Abb. 2.1 veranschaulicht.

2.1.3 Branch-and-Bound für binäre Probleme

Betrachtet wird folgendes Optimierungsproblem:

$$\begin{gathered} \min f(x) \\ x \in X = \{x \in \mathbb{R}^n, \ g(x) \leq 0, \ h(x) = 0\} \\ x \in \mathbb{Z}^n, \text{ oder auch } x \in \{0,1\}^n \end{gathered} \tag{2.8}$$

Gilt in Optimierungsproblemen noch zusätzlich die Bedingung $x \in \mathbb{Z}^n$, bzw. $x \in \{0,1\}^n$, so ist es erforderlich, die Lösungsansätze an die neue Situation anzupassen. Das Branch-and-Bound-Verfahren ermöglicht es, eine ganzzahlige Lösung für Optimierungsprobleme

```
begin
   Lösung_gefunden := FALSE
   suche Startlösung x_B und Startbasis B
   stelle das RMP mit den zu B gehörenden Variablen auf
   while Lösung_gefunden == FALSE do
      x_B := Lösung des RMP
      passe B an x_B an
      suche die zum RMP dualen Variablen y_B
      suche Lösung des Pricing-Problems c*
      if c* < 0 then füge Variable(n) minimal reduzierter Kosten zum RMP hinzu
         else Lösung_gefunden := TRUE
   end
   return x_B
end
```

Abbildung 2.1: So könnte ein Columnn Generation Algorithmus grob aussehen.

zu finden. Daneben können auch Schnittebenenverfahren verwendet werden. Außerdem gibt es noch das Branch-and-Cut-Verfahren, was im wesentlichen eine Mischung aus Branch-and-Bound und Schnittebenenverfahren ist.

Vorgehensweise

Zu Beginn löst man die relaxierte Variante des Problems. Das heißt, dass die Ganzzahligkeitsbedinung vorerst ignoriert wird. Im binären Fall bedeutet das, dass die Bedingung $x \in \{0,1\}^n$ durch $x \in [0,1]^n$ ersetzt wird. Ist die gefundene Lösung $\underline{x}$ ganzzahlig, so ist das Problem gelöst. Ist dies nicht der Fall, so wird $f(\underline{x}) = \underline{z}$ als untere Schranke für den Zielfunktionswert gespeichert, da eine ganzzahlige Lösung nicht besser sein kann. Außerdem wird die obere Schranke für den Zielfunktionswert $\bar{z}$ auf ∞ gesetzt, da noch keine ganzzahlige Lösung gefunden wurde, mit der man eine bessere obere Schranke gegeben hätte. Dann wählt man eine nicht ganzzahlige Komponente $\underline{x}_i$ von $\underline{x}$ aus und teilt das Problem in zwei relaxierte Teilprobleme auf, auch **Branching** genannt. Eins der beiden neuen Probleme hat die neue Nebenbedingung $x_i \leq \lfloor \underline{x}_i \rfloor$, und im Anderen gilt $x_i \geq \lceil \underline{x}_i \rceil$. Für den binären Fall gilt sogar $x_i = 0$ bzw. $x_i = 1$. Da in diesem Fall eine Komponente konstant ist, kann x auch entsprechend verkürzt werden. Gegebenenfalls müssen Zielfunktion und Nebenbedingungen daran angepasst werden. Wenn es mehrere Variablen gibt, die nicht ganzzahlig sind, so muss mit einem geeigneten Verfahren entschieden werden, nach welcher Variable man verzweigt. Falls in den Unterproblemen ebenfalls keine ganzzahligen Lösungen gefunden werden, können auch diese wiederum entsprechend aufgeteilt werden, und so weiter. Das Verfahren lässt sich somit als Baum darstellen. In jedem Knoten steht ein Teilproblem des vorigen Knotens. Der Zielfunktionswert der Lösung eines Elternproblems ist dabei eine untere Schranke für die beiden

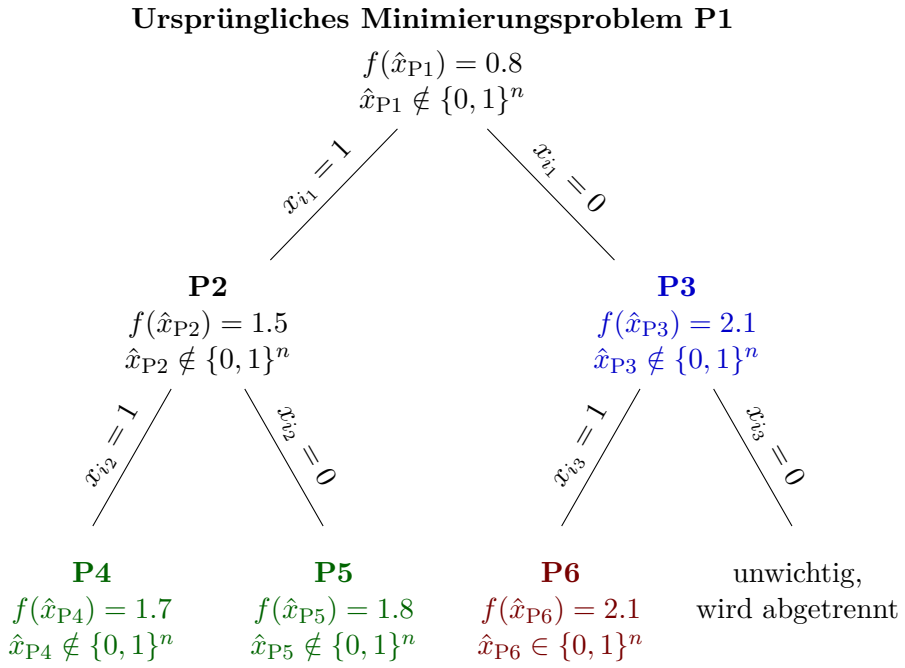

Abbildung 2.2: Ein Beispiel für einen Branch-and-Bound-Baum. Eine binäre Lösung $\hat{x}_{P6}$ wurde am roten Knoten, als Lösung von **P6** gefunden. Da die Lösungen $\hat{x}_{P6}$ und $\hat{x}_{P3}$ gleich gut sind, ist auch ein Optimum für den ganzen, von **P3** ausgehenden Ast gefunden. Ab **P3** und **P6** müssen somit keine weiteren Verzweigungen betrachtet werden. Es bleibt zu überprüfen, ob von den grünen Knoten aus noch eine ganzzahlige Lösung gefunden werden kann, die besser als $\hat{x}_{P6}$ ist.

Kinderprobleme. Falls in einem Knoten eine ganzzahlige Lösung gefunden wird, die einen besseren Wert als $\tilde{z}$ liefert, so wird $\tilde{z}$ damit ersetzt. Andere Knoten, deren relaxierte Lösungen nicht besser als $\tilde{z}$ sind, müssen dann nicht mehr betrachtet werden und können aus dem Problem entfernt werden. Auf diese Weise kann man sich auch einige weitere Teilprobleme sparen. Dieses Abschneiden von Zweigen wird auch **Bounding** genannt. Der optimale Zielfunktionswert ist größer oder gleich $\underline{z}$ und kleiner oder gleich $\bar{z}$.
Satt alle Knoten abzuarbeiten, kann man Rechenaufwand einsparen, indem man das Verfahren abbricht, wenn $\bar{z}$ nicht zu stark von $\underline{z}$ abweicht. Das heißt, dass für ein vorher fest gewähltes ε gilt: $\varepsilon \geq \frac{\bar{z}-\underline{z}}{\underline{z}}$.

2.1.4 Branch-and-Price

Um große, lineare, ganzzahlige Probleme zu lösen eignet sich eine Kombination aus Branch-and-Bound-Verfahren und Column Generation, das sogenannte Branch-and-Price-Verfahren. Ein Column Generation-Verfahren wird dabei in einen Branch-and-Bound-

Algorithmus eingebettet. Für den Pricing-Schritt sollte ein exaktes Lösungsverfahren verwendet werden, da es ansonsten möglich ist, dass Zweige entfernt werden, durch die eine bessere Zielfunktion erreicht werden könnte.
Außerdem sollte es für den Branching-Schritt eine gute Regel geben, nach der verzweigt wird. Dies ist allerdings schwierig, da es viele Dinge zu beachten gibt [DSD84]. Dabei sollte zum Beispiel verhindert werden, dass eine Variable in den verschiedenen Teilproblemen mehrmals zum RMP hinzugefügt wird. Deswegen werden meistens generierte Variablen vom Elternproblem an die Kinderprobleme weitergegeben. Zusätzlich sollte darauf geachtet werden, dass so verzweigt wird, dass die Struktur des Pricing-Problems erhalten bleibt.

2.2 Verkehrsmodell

Das Verkehrsmodell orientiert sich an [Rou05] und natürlich an der Vorgängerarbeit [Ell12].

2.2.1 Mathematische Beschreibung des Verkehrsmodells

Das Verkehrsmodell, das dieser Arbeit zugrunde liegt, besteht aus drei Teilen: dem **Netzwerk**, der **Nachfragematrix** und den **Wirkungsmodellen**. Das Netzwerk repräsentiert das Verkehrsnetz, also die Straßen und beschreibt auch wie diese miteinander zusammenhängen. Die Nachfragematrix gibt an, wie viel Verkehr zwischen einzelnen Teilen des Verkehrsnetzes fließt. Mit den Wirkungsmodellen wird der Verkehrsfluss analysiert und bewertet. Da es in dieser Arbeit darum geht, die Fahrzeit der Verkehrsteilnehmer zu minimieren, ist nur das Benutzermodell als Wirkungsmodell von Interesse.
Die Verkehrsnachfrage muss auf die einzelnen Straßen im Netzwerk verteilt bzw. zugewiesen werden. Eine solche Zuweisung wird als **Umlegung** bezeichnet. Mit einer Umlegung kann das **Systemoptimum** oder das **Nutzeroptimum** erreicht werden. Beide Begriffe werden in diesem Kapitel noch erklärt. Wird eine Baumaßnahme auf einer Straße durchgeführt, so kann dies Auswirkungen auf andere, vor allem benachbarte, Straßen haben, oder sogar auf das gesamte Verkehrsnetz. Treten also Baumaßnahmen auf, ist es erforderlich, eine neue Umlegung zu berechnen.
Um das Netzwerk mathematisch zu beschreiben wird ein **gerichteter Graph** G, mit **Knotenmenge** V und **Kantenmenge** $E \subset V \times V$, verwendet. Eine Einbahnstraße wird dabei durch genau eine Kante $(v, w) \in E$ repräsentiert und eine Zweirichtungsstraße durch genau zwei entgegengesetzte Kanten $(v, w),\ (w, v) \in E$. Jede Kante $e \in E$ hat außerdem noch einige spezifische Attribute, wie zum Beispiel, die **Kapazität** $p_e \in \mathbb{N}$, die **Fahrzeit im unbelasteten Netz** $t_{0,e} \in \mathbb{N}$ oder die **Capacity-Constraint-Funktion** (CR-Funktion). Diese berechnet die Fahrzeit auf einer Kante e, in Abhängigkeit des **aktuellen Flusses** f_e und der beiden Konstanten p_e und $t_{0,e}$. f_e ist größer oder gleich null. Außerdem kann f_e als Anzahl der auf e fahrenden Fahrzeuge aufgefasst werden, wobei

für f_e nicht-ganzzahlige Werte zugelassen werden können, um das Modell zu vereinfachen. Die CR-Funktion wird folgendermaßen [PTV10] beschrieben:

$$c_{t_{0,e},p_e}(f_e) = t_{0,e} \cdot \left(1 + \left(\frac{f_e}{p_e}\right)^2\right) \tag{2.9}$$

Für $f_e \in \mathbb{R}^+$ und $p_e \neq 0$ ist diese Funktion konvex und streng monoton wachsend.
Mit der CR-Funktion kann die Fahrzeit auf einer Kante e in Abhängigkeit vom Fluss f_e berechnet werden. Die Gesamtfahrzeit aller Verkehrsteilnehmer auf e wird mit der Funktion $h_{t_{0,e},p_e}(\cdot)$ berechnet. Man erhält $h_{t_{0,e},p_e}(\cdot)$ indem man die CR-Funktion mit f_e, also der Anzahl der Fahrzeuge auf e multipliziert. Somit ergibt sich:

$$h_{t_{0,e},p_e}(f_e) = f_e \cdot c_{t_{0,e},p_e}(f_e) \tag{2.10}$$

Die Gesamtfahrzeit H aller Verkehrsteilnehmer im gesamten Netzwerk ist die Summe aller Gesamtfahrzeiten der einzelnen Kanten:

$$H(f) = \sum_{e \in E} h_{t_{0,e},p_e}(f_e) = \sum_{e \in E} f_e \cdot c_{t_{0,e},p_e}(f_e) \tag{2.11}$$

Um die Verkehrsnachfrage zu modellieren, wird das Verkehrsnetz in **Bezirke** unterteilt, wobei R die **Menge der Bezirke** beschreibt. Jeder Knoten aus V wird mindestens einem Bezirk zugeordnet. Die Nachfragematrix D gibt die Verkehrsnachfrage bzw. den Verkehrsfluss zwischen den einzelnen Bezirken an. Jedes Paar $\{i, j\} \in R \times R$ wird als *commodity* bezeichnet. Der Eintrag d_{ij} in D steht für die **Verkehrsnachfrage** von Bezirk i nach Bezirk j, gibt also an, wie viele Fahrzeuge von i nach j fahren. Jeder Bezirk hat mindestens einen Knoten, einen sogenannten **Anbindungsknoten**, der den Bezirk mit dem Verkehrsnetz verbindet. Es ist auch möglich, dass sich mehrere Bezirke einen Anbindungsknoten teilen. In diesem Fall kann die Verkehrsnachfrage zwischen diesen Bezirken trivialerweise über den gemeinsamen Knoten abgewickelt werden und wird deshalb für den Gesamtfluss nicht berücksichtigt. $L(r)$ bezeichnet die Menge der Anbindungsknoten des Bezirks r, dabei gilt: $\emptyset \neq L(r) \subset V \ \forall r \in R$.

2.2.2 Systemoptimum

Der Begriff Systemoptimum bezeichnet den zeitminimalen Fluss im Verkehrsnetz. Damit ist $\tilde{f} = \arg\min_f(H(f))$ gemeint, also der Fluss, der die Summe aller Fahrzeiten im Verkehrsnetz minimiert. Hierbei ist zu beachten, dass das Systemoptimum über die Kanten definiert ist, da zur Berechnung von $H(f)$ über alle Kanten summiert wird. Da $H(f)$ stetig ist und über die abgeschlossene und beschränkte Menge der zulässigen Flüsse minimiert wird, gibt es immer ein Systemoptimum, falls die Menge der Flüsse zulässig ist. Um dieses Minimum zu erreichen kann es notwendig sein, dass manche

Verkehrsteilnehmer eine Route wählen müssen, die mehr Zeit in Anspruch nimmt als ein anderer Weg, der zur Verfügung steht. Dieser Umstand wird in diesem Kapitel noch durch zwei Beispiele anschaulich erklärt. Ein **optimaler Fluss** ist in dieser Arbeit immer das Systemoptimum.

2.2.3 Nutzeroptimum

Mit Hilfe des ersten Wardrop'schen Prinzips wird das Nutzeroptimum wie folgt beschrieben:

> *„Jeder einzelne Verkehrsteilnehmer wählt seine Route derart, dass der Widerstand auf allen alternativen Routen letztlich gleich ist und jeder einseitige Wechsel auf eine andere Route die persönliche Fahrzeit erhöhen würde.“* aus [PTV11]

Sei E die Kantenmenge und V die Knotenmenge eines Graphen G. Ein Kantenzug ist eine Abfolge $(v_1, e_1, v_2, e_2, \ldots, e_a, v_a)$ von Knoten $v_1,\ v_2,\ \ldots,\ v_a \in V$ und Kanten $e_1 = (v_1, v_2),\ e_2 = (v_2, v_3),\ \ldots,\ e_a = (v_{a-1}, v_a)$. Eine Route ist ein kreisfreier Kantenzug. Anders als das Systemoptimum ist das Nutzeroptimum über Routen definiert. Die Anzahl der Routen ist oft deutlich größer als die Anzahl der Kanten. Um das Nutzeroptimum zu berechnen ist es also nachteilig alle Routen zu betrachten. In Abschnitt 2.2.5 wird erklärt, wie das Nutzeroptimum effizienter ermittelt werden kann. Man geht davon aus, dass jeder Verkehrsteilnehmer ein Egoist ist, der möglichst schnell am Ziel sein will. Deswegen wechselt kein Verkehrsteilnehmer seine Route, da sich, wie oben beschrieben, seine Fahrzeit dadurch erhöhen würde. Man erhält somit ein Gleichgewicht, weswegen die Umlegung, die das Nutzeroptimum zum Ziel hat auch **Gleichgewichtsumlegung** heißt. Dabei handelt es sich um ein Nash-Gleichgewicht, da kein Verkehrsteilnehmer einen Vorteil hat, wenn er seine Route ändert.

2.2.4 Beispiel von Pigou und Paradoxon von Braess

Im Folgenden werden zwei Beispiele behandelt, die den Unterschied zwischen Systemoptimum und Nutzeroptimum verdeutlichen sollen, wobei sich herausstellt, dass das Nutzeroptimum deutlich schlechter ist.

Beispiel von Pigou

In diesem Beispiel haben alle Verkehrsteilnehmer den Startort s und den Zielort t. Es gibt genau zwei verschiedene Möglichkeiten von s nach t zu gelangen. Bei der ersten Möglichkeit handelt es sich um eine relativ lange Straße, die dafür aber so gut ausgebaut ist, dass jeder Verkehrsteilnehmer dort stets gleiche Fahrzeit hat. Die zweite Möglichkeit ist eine sehr kurze aber schmale Straße. Die Fahrzeit auf dieser Straße steigt mit der Zahl der dortigen Verkehrsteilnehmer stark an. Die Verkehrsnachfrage von s nach t sei 100.

Da alle Verkehrsteilnehmer ihre persönliche Fahrzeit minimieren wollen, wählt jeder von

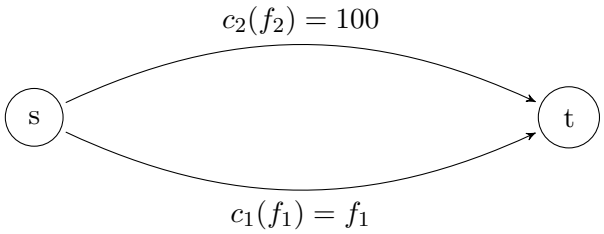

Abbildung 2.3: c_2 und c_1 beschreiben die Fahrzeiten in Abhängigkeit vom Verkehrsfluss auf den jeweiligen Kanten. Für die Gesamtfahrzeit gilt: $H = 100 \cdot f_1 + f_2^2$

ihnen die untere, enge Straße. Denn dort braucht man, auch bei viel Verkehrsaufkommen, nur höchstens so lang wie auf der oberen Straße. Wegen der dadurch entstehenden Überfüllung der unteren Straße beträgt die Fahrzeit aller Verkehrsteilnehmer unten 100 Zeiteinheiten. Außerdem ist das auf diese Weise das Nutzeroptimum erreicht und die Gesamtfahrzeit beträgt 10000 Zeiteinheiten.
Um das Systemoptimum zu erreichen müssten genau fünfzig Verkehrsteilnehmer oben fahren während der Rest unten bleibt. Die daraus resultierende Gesamtfahrzeit beträgt 7500 Zeiteinheiten und die durchschnittliche Fahrzeit somit 75 Zeiteinheiten. Insgesamt erhält man eine Verbesserung.
Das Beispiel von Pigou verdeutlicht, dass egoistisches Verhalten einzelner Verkehrsteilnehmer ein Gesamtergebnis liefern kann, das global betrachtet nicht optimal ist.

Paradoxon von Braess

Während Pigous Beispiel ein wenig überraschendes Ergebnis liefert, resultiert aus dem Paradoxon von Braess eine viel bemerkenswertere Erkenntnis: Das Einbinden zusätzlicher Straßen ins Verkehrsnetz kann die durchschnittliche Fahrzeit im Nutzeroptimum verschlechtern.

Auch in diesem Beispiel wollen alle Verkehrsteilnehmer von s nach t gelangen. Es stehen auch wieder genau zwei verschiedene Routen zur Verfügung, wobei diese jeweils aus einer kurzen, schmalen Straße, einem Zwischenknoten v, bzw. w und einer langen, gut ausgebauten Straße bestehen. Im Nutzeroptimum verteilen sich die Verkehrsteilnehmer gleich auf beide Strecken. Bei einer Verkehrsnachfrage von 100 von s nach t, sind alle Verkehrsteilnehmer durchschnittlich 150 Zeiteinheiten unterwegs.

Nun wird das Verkehrsnetz verändert: Eine neue Straße zwischen v und w wird eröffnet, so dass es möglich ist von v nach w in 0 Zeiteinheiten zu gelangen. Auf der neuen Route s→v→w→t ist die Fahrzeit geringer oder gleich der Fahrzeit auf den anderen beiden Routen. Deswegen benutzen, im Nutzeroptimum, alle Verkehrsteilnehmer die neue Strecke. Die durchschnittliche Fahrzeit beträgt damit 200 Zeiteinheiten und ist deutlich schlechter

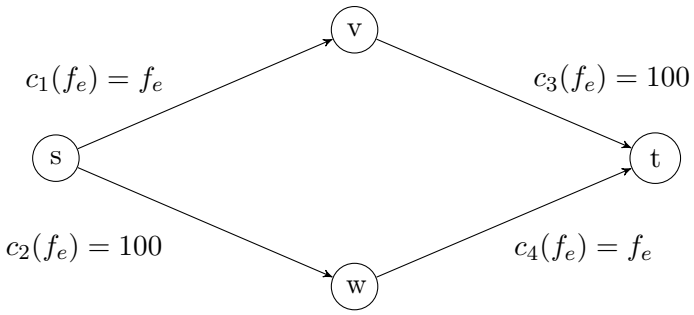

Abbildung 2.4: Hier sind Nutzeroptimum und Systemoptimum gleich.

als im nicht erweiterten Verkehrsnetz.

2.2.5 Zusammenhang zwischen Systemoptimum und Nutzeroptimum

Mit (G, d, c) wird eine Instanz des Verkehrsflussproblems bezeichnet. G ist dabei ein gerichteter Graph mit jeweils endlicher Knotenmenge V und Kantenmenge E. $d \in \mathbb{N}_0^{|V|\times|V|}$ beschreibt die Verkehrsnachfrage zwischen den einzelnen Knoten. c ist ein Vektor mit $|E|$ Einträgen, von denen jeder die Fahrzeitfunktion $c_e : \mathbb{R}^+ \to \mathbb{R}^+$ der zugehörigen Kante beschreibt.
Ist $c(\cdot)$ differenzierbar, so gibt es eine sogenannte marginale Fahrzeitfunktion $c^*(\cdot)$, die wie folgt definiert ist:

$$c^*(f_e) = \frac{d}{df_e}\left(f_e \cdot c(f_e)\right) \tag{2.12}$$

Mit Hilfe dieser Definitionen lässt sich ein wichtiger Zusammenhang zwischen Systemoptimum und Nutzeroptimum herstellen:

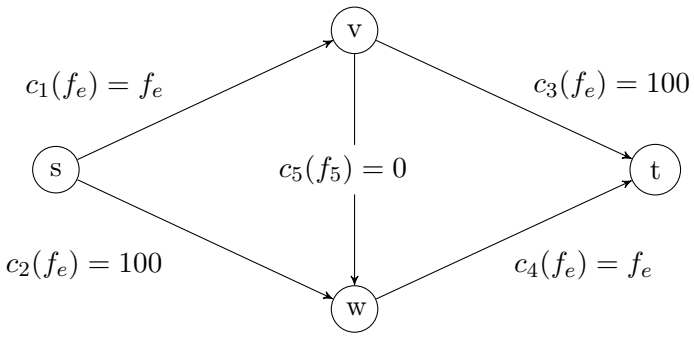

Abbildung 2.5: Eine Kante wurde hinzugefügt, was dazu führt, dass sich die Fahrzeit für alle Verkehrsteilnehmer erhöht.

Aus [Rou05], S. 27:
Sei (G, d, c) eine Instanz mit stetig differenzierbaren Fahrzeitfunktionen $c(\cdot)$, wobei sowohl $c(x)$, als auch $x \cdot c(x)$ konvex sind. Außerdem seien $c^*(\cdot)$ die zugehörigen marginalen Fahrzeitfunktionen. Dann gilt: Ein für (G, d, c) zulässiger Fluss f ist genau dann optimal, wenn f das Nutzeroptimum für (G, d, c^*) ist.

Unter Verwendung des obigen Satzes lässt sich nun auch das Nutzeroptimum einfach berechnen. Beweis und Herleitung ist in [Rou05] zu finden.

2.2.6 Verkehrsflussproblem

Ein wichtiger Bestandteil dieser Arbeit ist die Bestimmung des Nutzeroptimums. Um sich, zu diesem Zweck, den Satz aus dem vorigen Abschnitt zu Nutze zu machen, werden die Fahrzeitfunktionen c des Ausgangsproblems einfach als marginale Fahrzeitfunktionen betrachtet. Indem man für alle $e \in E$, $c_{t_{0,e},p_e}(f_e)$ nach f_e integriert und dann durch f_e dividiert. Man erhält dadurch die Funktion $\tilde{c}_{t_{0,e},p_e}(f_e)$. Das Nutzeroptimum des Ausgangsproblems lässt sich nun ermitteln, indem man $c(\cdot)$ durch $\tilde{c}(\cdot)$ ersetzt und dann das Systemoptimum davon bestimmt. $c_{t_{0,e},p_e}(f_e)$ wurde in Abschnitt 2.2.1 wie folgt definiert:

$$c_{t_{0,e},p_e}(f_e) = t_{0,e} \cdot \left(1 + \left(\frac{f_e}{p_e}\right)^2\right)$$

Für $\tilde{c}_{t_{0,e},p_e}(f_e)$ ergibt sich:

$$\tilde{c}_{t_{0,e},p_e}(f_e) = \frac{\int c_{t_{0,e},p_e}(f_e) df_e}{f_e} = t_{0,e} \cdot \left(1 + \frac{1}{3}\left(\frac{f_e}{p_e}\right)^2\right) \tag{2.13}$$

$\tilde{c}_{t_{0,e},p_e}(f_e)$ lässt sich zweimal ableiten und es gilt:

$$\tilde{c}''_{t_{0,e},p_e}(f_e) = \frac{2t_{0,e}}{3p_e^2} \geq 0$$

$\tilde{c}_{t_{0,e},p_e}(f_e)$ ist somit konvex und stetig differenzierbar.
Falls $f_e \in \mathbb{R}_0^+$, so gilt für $f_e \cdot \tilde{c}_{t_{0,e},p_e}(f_e)$:

$$\frac{d^2}{df_e^2}\left(f_e \cdot \tilde{c}_{t_{0,e},p_e}(f_e)\right) = f_e \cdot \frac{2t_{0,e}}{3p_e^2} \geq 0$$

$f_e \cdot \tilde{c}_{t_{0,e},p_e}(f_e)$ ist also konvex und stetig differenzierbar für $f_e \in \mathbb{R}_0^+$. Der Satz aus dem vorigen Abschnitt kann also angewandt werden. Für die Zielfunktion des Problems ist es noch notwendig, die Funktion $\tilde{h}_{t_{0,e},p_e}(f_e)$ zu definieren:

$$\tilde{h}_{t_{0,e},p_e}(f_e) = \tilde{c}_{t_{0,e},p_e}(f_e) \cdot f_e \tag{2.14}$$

Man erhält ein Optimierungsproblem, das die Gesamtfahrzeit des Systemoptimums von $(G, d, \tilde{c})$, bzw. des Nutzeroptimums von (G, d, c) ermittelt:

Verkehrsflussproblem:

$$\min \sum_{e \in E} \tilde{h}_{t_{0,e}, p_e}(f_e)$$

$$\begin{aligned}
f_e &= \sum_{r \in R} \sum_{\substack{r' \in R \\ r' \neq r}} f_e^{(r,r')} && \forall e \in E \\
f_e^{(r,r')} &\geq 0 && \forall e \in E \\
& && \forall r,\ r' \in R, \text{ mit } r \neq r' \\
\sum_{u:(u,v) \in E} f_{(u,v)}^{(r,r')} &= \sum_{u:(v,u) \in E} f_{(v,u)}^{(r,r')} && \forall r,\ r' \in R, \text{ mit } r \neq r' \\
& && v \in V \setminus (L(r) \cup L(r')) \\
\sum_{v \in L(r)} \left(\sum_{u:(v,u) \in E} f_{(v,u)}^{(r,r')} - \sum_{u:(u,v) \in E} f_{(v,u)}^{(r,r')} \right) &= d_{r,r'} && \forall r,\ r' \in R, \text{ mit } r \neq r'
\end{aligned}$$

In der ersten Nebenbedingungsklasse wird der Fluss jeder Kante in Flüsse zwischen den einzelnen Quelle-Ziel-Paaren aufgeteilt, wobei für $f_e^{(r,r')}$ r der Startbezirk und r' der Zielbezirk ist. Die Bedingung $r' \neq r$ besagt hier und in den folgenden Nebenbedingungsklassen, dass der Verkehrsfluss innerhalb eines Bezirks nicht betrachtet werden muss. Die zweite Bedingung besagt, dass kein Fluss negativ sein darf. Andernfalls müsste man davon ausgehen, dass auch Geisterfahrer unterwegs sind. Die dritte Nebenbedingungsklasse beschreibt die Flusserhaltung. Das bedeutet, dass der Fluss zwischen einem Quelle-Ziel-Paar (r, r'), der in einen Knoten hinein fließt auch vollständig wieder hinaus fließen muss, es sei denn es handelt sich um einen Anbindungsknoten von r oder r'. Die vierte Nebenbedingungsklasse sorgt dafür, dass die in der Nachfragematrix angegebene Verkehrsnachfrage auch erfüllt wird.

3 Modellierung des Wartungszeitproblems

In diesem Kapitel wird das Wartungszeitmodell aus [Ell12] beschrieben.

3.1 Beschreibung der Parameter

Im Folgenden wird jeder Parameter, der im Modell verwendet wird, ausführlich beschrieben. Einige dieser Parameter wurden bereits in Abschnitt 2.2 eingeführt.

- $G(V, E)$ ist ein gerichteter Graph mit Knotenmenge V und Kantenmenge E, der das **Verkehrsnetz** des Modells darstellt. Dadurch, dass G gerichtet ist, können auch Einbahnstraßen erfasst werden. Außerdem kann man, bei Straßen mit zwei Fahrtrichtungen, Bauarbeiten modellieren, die nur eine Richtung betreffen.

- $V = \{1, 2, 3, \ldots\}$ steht für die endliche **Menge der Knoten** des Verkehrsnetzes. Die Knoten aus V verbinden die einzelnen Straßen des Netzwerks miteinander. Nur von einem Knoten aus kann man die Straße wechseln.

- $E \subseteq V \times V$ ist die **Menge der Kanten**. Die Kanten beschreiben die Straßen des Verkehrsnetzes, wobei eine Kante immer zwischen genau zwei Knoten und in genau eine Richtung verläuft. Die Kanten benutzen Infrastrukturbauwerke, wie zum Beispiel Brücken, Tunnel oder den Straßenbelag selbst.

- $p_e \in \mathbb{N}_0$ ist die **Kapazität der Kante** e, wenn diese gerade von keiner Wartungsmaßnahme eines Infrastrukturbauwerks betroffen ist. Finden Wartungsarbeiten eines zu e gehörenden Bauwerks statt, so verringert sich die Kapazität. In diesem Modell kann durch Bauarbeiten die Kapazität nie auf null reduziert werden.

- $\kappa \in \mathbb{Q} \cap]0, 1[$ steht für den **Anteil der Kapazität** einer Kante e, der bei einer Wartungsmaßnahme erhalten bleibt. Ist e gerade von Wartungsarbeiten betroffen, so ist die neue Kapazität gleich $\lfloor \kappa p_e \rfloor$. κ wird so gewählt, dass jede Straße, befahrbar ist, auch wenn dort gebaut wird. Zu beachten ist auch, dass ein κ einheitlich für alle Kanten gewählt wird. Es wäre auch möglich κ kantenspezifisch zu wählen. Dieser Ansatz wurde in diesem Modell allerdings nicht berücksichtigt.

- R ist die endliche **Menge der Bezirke**. Jeder Knoten aus V ist einem Bezirk aus R zugeordnet. Knoten, die reine Verkehrsknotenpunkte und deshalb keinem Bezirk zugeordnet sind, werden hier nicht modelliert. Umgekehrt hat jeder Bezirk mindestens einen Anbindungsknoten, in dem der Verkehr starten und enden kann. Hat ein Bezirk mehrere Anbindungsknoten, so kann der ein-, bzw. ausgehende Verkehr einen der Knoten wählen beliebig. Die Bezirke sind wichtig für die Modellierung der Verkehrsnachfrage.

- $L(r) \subseteq V$ ist die Menge der **Anbindungsknoten eines Bezirks** r. Für jeden Bezirk $r \in R$ gilt: $L(r) \neq \emptyset$, da jeder Bezirk an das Verkehrsnetz angebunden

sein muss. Es ist auch möglich, dass sich zwei Bezirke einen Anbindungsknoten teilen. In diesem Fall kann die Nachfrage zwischen den beiden Bezirken über den gemeinsamen Knoten abgewickelt werden, ohne dabei eine Kante zu benutzten. Aus diesem Grund hat der Fluss zwischen diesen beiden Bezirken auch keine Auswirkung auf den restlichen Verkehr.

- $D \in \mathbb{N}_0^{|R|\times|R|}$ ist die **Nachfragematrix** des Verkehrsmodells. Jeder Eintrag $d_{r,r'}$ steht dabei für die Verkehrsnachfrage von Bezirk r nach Bezirk r'. Die Diagonaleinträge sollten null sein, da der Verkehrsfluss innerhalb eines Bezirks in diesem Modell nicht betrachtet wird.

- $f_e \in \mathbb{Q}_0^+$ bezeichnet den **(Verkehrs-)Fluss** auf der Kante e. Dieser darf die aktuelle Kapazität von e nicht überschreiten. Außerdem setzt sich f_e aus allen Teilflüssen $f_e^{(r,r')}$, zwischen den Quelle-Ziel-Paaren (r, r') zusammen. Dabei gilt $(r, r') \in R \times R$ und $r \neq r'$. Wenn ein Teilfluss zwischen (r, r') die Kante e nicht benutzt, so ist $f_e^{(r,r')}$ gleich null. Der Vektor f, mit Länge $|E|$, enthält den Fluss jeder einzelnen Kante.

- $t_{0,e} \in \mathbb{Q}_0^+$ ist die **Fahrzeit auf einer unbelasteten Kante**. Eine Kante ist genau dann unbelastet, wenn kein Verkehr auf ihr unterwegs ist. Wenn genau ein Verkehrsteilnehmer über diese Kante fährt, so beträgt seine Fahrzeit genau $t_{0,e}$. Der tatsächliche Fahrzeit auf einer Kante e hängt vom Verkehrsfluss f_e darauf ab.

- $T = \{1, 2, 3, \ldots\}$ ist eine endliche Menge, die für den **betrachteten Zeitraum** steht, in dem alle Wartungsarbeiten stattfinden müssen. Eine Wartungsmaßnahme nimmt immer genau ein Jahr $t \in T$ in Anspruch.

- $B = \{b_1, b_2, b_3, \ldots\}$ ist die endliche Menge der Infrastrukturbauwerke. Jedes dieser Bauwerke muss im betrachteten Zeitraum T genau einmal gewartet werden.

- $E(b) \subseteq E$ bezeichnet für alle $b \in B$ die Menge aller **Kanten, die von einer Wartungsmaßnahme des Bauwerks b betroffen sind**. So können Bauarbeiten an einer Straße mit zwei Fahrtrichtungen dazu führen, dass die Kante, welche die Hinrichtung beschreibt genauso betroffen ist, wie die Kante, welche die Rückrichtung beschreibt. Außerdem sind alle Mengen $E(b) \subseteq E$ paarweise disjunkt. Dies ist allerdings ungünstig, wenn es um Wartungsmaßnahmen direkt an Verkehrsknotenpunkten geht, da eine Straße an zwei Verkehrsknotenpunkte angeschlossen sein kann. Es kann allerdings auch Kanten geben, die kein Bauwerk haben, das im betrachteten Zeitraum T gewartet werden muss. Diese Bauwerke sind alle in der Menge $E \setminus \bigcup_{b\in B} E(b)$ enthalten.

- $m \in \mathbb{N}$ ist die **maximale Anzahl an Bauwerken, die in einem Jahr gewartet werden dürfen**. Hier spielen Ressourcen, wie die pro Jahr verfügbaren Arbeitskräfte und Baumaschinen eine Rolle.

- $t_b^\times \in T$ ist der **letzte zulässige Wartungszeitpunkt** für ein Bauwerk b. Diese Deadline sorgt, dafür, dass sich der Zustand keines Bauwerks so sehr verschlechtert, dass es für den Verkehr gesperrt werden muss. Jedes Bauwerk muss im Jahr seiner Deadline oder davor gewartet werden.

- $\alpha_{b,t} \in \{0,1\}$ gibt an, ob ein Bauwerk b im Jahr t noch vor seiner Deadline steht.
  Es gilt: $\alpha_{b,t} = \begin{cases} 1, \text{ falls } t \leq t_b^\times \\ 0, \text{ falls } t > t_b^\times \end{cases}$

- $k_b \in \mathbb{Q}^+$ beschreibt die **Kosten**, die bei der Wartung des Bauwerks b anfallen. Die Kosten hängen dabei nur vom Bauwerk und nicht vom Zeitpunkt der Wartungsarbeiten ab.

- $\bar{k} \in \mathbb{Q}^+$ ist die **jährliche Kostenvorgabe**. Die Kosten für Wartungsarbeiten sollen jedes Jahr in etwa gleich hoch sein.

- $\varepsilon^+,\ \varepsilon^- \in \mathbb{Q} \cap [0,1]$ sind die nach oben, bzw. unten **zulässigen relativen Abweichungen von der Kostenvorgabe** $\bar{k}$.

- $\beta_t \in \{0,1\}$ sagt aus, **ob die untere Kostenvorgabe im Jahr t zu erfüllen ist**. In späteren Jahren kann es vorkommen, dass die Zahl der nicht gewarteten Bauwerke so gering ist, dass es nicht mehr möglich ist, die untere Kostenvorgabe einzuhalten. Außerdem wird über einen sehr langen Zeitraum geplant, sodass zu Beginn nicht sicher ist, wie viel Budget gegen Ende zur Verfügung steht. Es ist deswegen sinnvoll β_t für den Anfang des Planungszeitraums auf eins zu setzen und dann, ab einem bestimmten Jahr, bis zu Ende des Zeitraums auf null.
  Es gilt: $\beta_t = \begin{cases} 1, \text{ falls im Jahr } t \text{ die Bauarbeiten mindestens } \varepsilon^- \cdot \bar{k} \text{ kosten müssen.} \\ 0, \text{ falls die Wartungsarbeiten im Jahr } t \text{ auch günstiger sein dürfen.} \end{cases}$

- S ist die **Menge aller möglichen Sperrkombinationen**. Eine Sperrkombination $\sigma^{(s)} \in \{0,1\}^{|B|}$ gibt an, welche Bauwerke in dieser Kombination s gewartet werden sollen. Damit sind die Sperrungen über Bauwerke definiert. Anhand der gesperrten Bauwerke ist ersichtlich, auf welchen Kanten sich die Kapazität verringert. Es gibt $2^{|B|}$ mögliche, aber nicht zwangsweise zulässige Sperrkombinationen. Es gilt: $\sigma^{(s)} = \left(\sigma_1^{(s)}, \sigma_2^{(s)}, \ldots, \sigma_{|B|}^{(s)}\right)$.
  Außerdem gilt: $\sigma_b^{(s)} = \begin{cases} 1, \text{ falls Bauwerk } b \text{ in Sperrkombination } s \text{ gewartet wird.} \\ 0, \text{ falls } b \text{ in } s \text{ nicht gewartet wird.} \end{cases}$
  Im Jahr $t \in T$ wird genau eine Sperrkombination aus S gewartet. Jedes Bauwerk, das gewartet wird hat darin einen Eintrag gleich eins. Für jede Sperrkombination gibt es $|T|$ Variablen, die eins oder null sein können, je nach dem, ob die zugehörige Sperrkombination in einem Jahr t benutzt wird oder nicht. In Abschnitt 3.2 wird

erklärt, warum gerade dieser Ansatz gewählt wurde, obwohl er für ein Problem mit sehr vielen Variablen sorgt. Da die Anzahl der möglichen Sperrkombinationen exponentiell ist, wird ein Lösungsansatz mit Column Generation gewählt. Dabei beginnt man mit einer Teilmenge $S' \subseteq S$, mit der eine zulässige Lösung möglich ist. Im Folgenden werden weitere Sperrkombinationen zu S' hinzugefügt, die für eine Verbesserung des Zielfunktionswerts sorgen. In Abschnitt 3.3 wird erläutert, wie ein solches S', falls vorhanden, gefunden werden kann.

3.2 Berechnung der Fahrzeit für eine Sperrkombination

Um zu entscheiden, wie sich eine Sperrkombination σ auf den Verkehr auswirkt, ist es wichtig, den Fluss des Nutzeroptimums im Netzwerk zu kennen. Dieser Fluss f^* ist eine Lösung des Verkehrsflussproblems aus Abschnitt 2.2.6, wobei die Kapazitätsnebenbedingungen an die Sperrkombinationen und die daraus resultierenden Verringerungen von Kapazitäten angepasst werden müssen. Falls ein Bauwerk b gesperrt ist, so reduziert sich die Kapazität der betroffenen Kanten auf $\lfloor \kappa \cdot p_e \rfloor$. Aus diesem Grund muss die Zielfunktion des Verkehrsflussproblems angepasst werden, da die Kantenkapazitäten dort eine Rolle spielen. In Abhängigkeit des Flusses f und der jeweiligen Sperrkombination σ lässt sich diese Funktion wie folgt ausdrücken:

$$\tilde{H}_\kappa(f, \sigma) := \tag{3.1}$$

$$:= \sum_{b \in B} \sum_{e \in E(b)} \left(\tilde{h}_{t_{0,e}, p_e}(f_e) \cdot (1 - \sigma_b) + \tilde{h}_{t_{0,e}, \lfloor \kappa p_e \rfloor}(f_e) \cdot \sigma_b \right) + \sum_{e \in E \setminus \bigcup_{b \in B} E(b)} \tilde{h}_{t_{0,e}, p_e}(f_e)$$

Die Funktion $\tilde{h}_{t_{0,e}, p_e}(f_e)$ wurde in (2.13) und $\tilde{c}_{t_{0,e}, p_e}(f_e)$ in (2.14) wie folgt definiert:

$$\tilde{h}_{t_{0,e}, p_e}(f_e) := \tilde{c}_{t_{0,e}, p_e}(f_e) \cdot f_e$$
$$\tilde{c}_{t_{0,e}, p_e}(f_e) := t_{0,e} \cdot \left(1 + \frac{1}{3} \left(\frac{f_e}{p_e} \right)^2 \right)$$

Der Fluss f^* sei die Minimallösung des Verkehrsflussproblems für eine gegebene Sperrkombination σ. Dann gilt: f^* ist der Verkehrsfluss im Nutzeroptimum, wenn die Sperrkombination σ im Verkehrsnetz aktiv ist. Die Gesamtfahrzeit dieses Nutzeroptimums berechnet sich dabei wie folgt:

$$H_\kappa(f, \sigma) := \tag{3.2}$$

$$:= \sum_{b \in B} \sum_{e \in E(b)} \left(h_{t_{0,e}, p_e}(f_e) \cdot (1 - \sigma_b) + h_{t_{0,e}, \lfloor \kappa p_e \rfloor}(f_e) \cdot \sigma_b \right) + \sum_{e \in E \setminus \bigcup_{b \in B} E(b)} h_{t_{0,e}, p_e}(f_e)$$

Für $h_{t_{0,e},p_e}(f_e)$ gilt:

$$h_{t_{0,e},p_e}(f_e) := c_{t_{0,e},p_e}(f_e) \cdot f_e$$

$c_{t_{0,e},p_e}(f_e)$ wurde dabei in (2.9) wie folgt definiert:

$$c_{t_{0,e},p_e}(f_e) = t_{0,e} \cdot \left(1 + \left(\frac{f_e}{p_e}\right)^2\right)$$

Die Funktion für die Gesamtfahrzeit $H_\kappa(f, \sigma)$ ist ein wichtiger Bestandteil der Zielfunktion des Wartungszeitproblems. In diesem Kapitel wird allerdings deutlich, dass der Fluss f^* des Nutzeroptimums von σ ermittelt werden muss, um die Gesamtfahrzeit im Nutzeroptimum berechnen zu können. Dazu ist es allerdings notwendig, das Verkehrsflussproblem zu lösen. Da dem Verfasser der vorliegenden Arbeit keine Möglichkeit bekannt ist, wie man f^* über eine Abwandlung der Zielfunktion oder der Nebenbedingungen erhalten kann, wird das Wartungszeitproblem als sehr großes lineares Minimierungsproblem modelliert. Dabei wird, wie bereits in Abschnitt 3.1 beschrieben, über die Menge der Sperrkombinationen optimiert. Für diese Sperrkombinationen muss die Fahrzeit bereits im Vorhinein berechnet werden. Es ist also ein aufwendiges Preprocessing notwendig, bei dem das nichtlineare Verkehrsflussproblem viele Male gelöst werden muss.

3.3 Ermittlung einer Anfangssperrkombination

Für die Menge der Anfangssperrkombinationen S' werden einfache zulässige Sperrkombinationen benötigt, auf welche die Column Generation angewendet werden kann. Zu diesem Zweck werden zunächst $|T|$ Kombinationen $\sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(|T|)}$ angelegt. Dabei sind in $\sigma^{(t)}$ genau die Einträge gleich eins, deren Bauwerke im Jahr t ihre Deadline haben. Auf diese Wiese wird die Deadline immer eingehalten und jedes Bauwerk wird genau einmal gewartet. Als Nächstes wird überprüft, ob diese Sperrkombinationen die maximale jährliche Wartungszahl an Bauwerken nicht übersteigen und das Kostenziel einhalten. Ist dies der Fall, so sind die Anfangssperrkombinationen gefunden und in der Startlösung wird jede Sperrkombination $\sigma^{(t)}$ im Jahr t angewendet.
Wird in einer Sperrkombination die maximale jährliche Wartungszahl an Bauwerken um genau $\tilde{m}$ überschritten, so müssen $\tilde{m}$ Bauwerke in Kombinationen der vorangehenden Jahre vorgezogen werden. Dieses Vorziehen erfolgt stufenweise, d. h. man versucht zuerst möglichst viele Bauwerke in die Kombination des Vorjahres zu integrieren. Dann wird versucht, die übrigen Bauwerke, die dort wegen der Beschränkung keinen Platz haben, im Vorvorjahr unterzubringen usw. Die Bauwerke können wegen der Deadline nur vorgezogen und nicht nach hinten verschoben werden. Es ist darauf zu achten, dass beim Vorverlegen der Bauwerke die untere Kostenschranke eingehalten wird.
Falls für eine Sperrkombination die obere Kostenschranke nicht erfüllt ist, müssen wiederum Wartungsmaßnahmen in Kombinationen vorhergehender Jahre vorverlegt werden.

Ist die untere Kostenbeschränkung verletzt, so müssen Bauwerke späterer Jahre in die aktuelle Sperrkombination vorverlegt werden. In späteren Jahren besteht die Gefahr, dass es nicht genügend Bauwerke gibt, die vorgezogen werden können. Aus diesem Grund ist es sinnvoll, die untere Kostenschranke für diese letzten Jahre zu ignorieren. Wenn Bauwerke wegen der Kostenvorgabe vorgezogen werden, erfolgt dies ebenfalls, wie oben beschrieben, stufenweise. Es ist dabei stets darauf zu achten, dass alle anderen Bedingungen eingehalten werden.
Falls es nicht möglich ist, die Sperrkombinationen so zu verändern, dass Kostenziel und jährliche Wartungsbeschränkung eingehalten werden, so gibt es keine zulässige Sperrkombinationsmenge. Damit ist wäre das Problem unzulässig.
Ist eine gültige Menge S' an Ausgangssperrkombinationen gefunden, so wird für jede Sperrkombination $\sigma^{(s)} \in S'$ der Fluss $f^{*(s)}$ des jeweiligen Nutzeroptimums ermittelt. Anschließend werden die Gesamtfahrzeiten aller Sperrkombinationen aus S' berechnet und aufaddiert. Damit erhält man den zu S' gehörigen Zielfunktionswert $\sum_{\sigma^{(s)} \in S'} H_\kappa \left(f^{*(s)}, \sigma^{(s)} \right)$.

3.4 Wartungszeitproblem

Das Wartungszeitproblem wird in diesem Abschnitt als lineares binäres Problem aufgestellt. Da Column Generation auf dieses Problem angewendet wird, werden auch das entsprechende Restricted Masterproblem und das Pricing-Problem ermittelt.

3.4.1 Formulierung als Optimierungsproblem

Für dieses Problem werden die Entscheidungsvariablen $x_t^{(s)}$ eingeführt. Diese geben an, ob ein Bauwerk s im Jahr t gewartet werden soll.
Es gilt: $x_t^{(s)} = \begin{cases} 1, \text{ falls } s \text{ im Jahr } t \text{ gewartet wird.} \\ 0 \text{ sonst} \end{cases}$
Es wird dabei über die Menge der Sperrkombinationen S optimiert.

Zielfunktion

In dieser Optimierungsaufgabe geht es darum, die Gesamtfahrzeit aller Verkehrsteilnehmer im Nutzeroptimum über alle Jahre des betrachteten Zeitraums T zu minimieren. Wie bereits in Abschnitt 3.2 erwähnt, werden die Fahrzeiten $H_\kappa(f^{*(s)}, \sigma^{(s)})$ der Sperrkombinationen vorher berechnet. Die Zielfunktion ist wie folgt definiert:

$$\min \left(\sum_{t \in T} \sum_{s \in S} H_\kappa \left(f^{*(s)}, \sigma^{(s)} \right) \cdot x_t^{(s)} \right) \tag{3.3}$$

Nebenbedingung: Jährlich nur eine Sperrkombination planbar

Mit dieser Nebenbedingung wird sichergestellt, dass für jedes Jahr genau eine Sperrkombination ausgewählt wird.

$$\sum_{s \in S} x_t^{(s)} = 1 \qquad \forall t \in T \tag{3.4}$$

Nebenbedingung: Wartung vor der Deadline

Diese Nebenbedingung sorgt dafür, dass jedes Bauwerk bis zu seiner Deadline genau einmal gewartet werden muss. Der Parameter $\alpha_{b,t}$ gibt an, ob das Bauwerk b im Jahr t noch vor seiner Deadline steht.

$$\sum_{t \in T} \sum_{s \in S} \left(\alpha_{b,t} \cdot \sigma_b^{(s)} \cdot x_t^{(s)} \right) = 1 \qquad \forall b \in B \tag{3.5}$$

Nebenbedingung: keine Wartung nach der Deadline

Aus dem Paradoxon von Braess aus Abschnitt 2.2.4 folgt, dass sich die Gesamtfahrzeit in einem Netzwerk verbessern kann, wenn eine Kante entfernt wird. Auf das Wartungszeitproblem übertragen bedeutet dies, dass es zu einer Verbesserung der Fahrzeit kommen kann, wenn ein Bauwerk gewartet wird. Damit beim Lösen des Problems aus diesem Grund kein Bauwerk mehrmals gewartet wird, sorgt diese Nebenbedingung dafür, dass keine Gebäude nach seiner Deadline gewartet wird.

$$\sum_{t \in T} \sum_{s \in S} \left((1 - \alpha_{b,t}) \cdot \sigma_b^{(s)} \cdot x_t^{(s)} \right) = 1 \qquad \forall b \in B \tag{3.6}$$

Nebenbedingung: Beschränkte Wartungszahl pro Jahr

Diese Nebenbedingung ist dafür zuständig, dass jedes Jahr nicht mehr Bauwerke gewartet werden als die maximale jährliche Wartungsanzahl erlaubt. Auf diese Weise werden in keinem Jahr mehr als m Bauwerke gewartet.

$$\sum_{s \in S} \sum_{b \in B} \left(\sigma_b^{(s)} \cdot x_t^{(s)} \right) \leq m \qquad \forall t \in T \tag{3.7}$$

Nebenbedingung: Beschränkte Kosten pro Jahr

Die jährlichen Baukosten sollen über die Jahre hinweg in etwa gleich bleiben. Da verschiedene Baumaßnahmen, trotz gleichen Personalaufwands unterschiedliche Kosten verursachen können, ist diese Nebenbedingung, zusätzlich zur beschränkten Wartungszahl pro Jahr notwendig.
In den letzten Jahren des Zeitraums T kann die untere Kostenschranke ignoriert werden.

Der Parameter β_t sagt aus, ob im Jahr t die untere Kostenschranke beachtet werden muss.

$$(1-\varepsilon^-)\cdot\bar{k}\beta_t \leq \sum_{s\in S}\sum_{b\in B}\left(\sigma_b^{(s)}\cdot x_t^{(s)}\cdot k_b\right) \leq \left(1+\varepsilon^+\right)\cdot\bar{k} \qquad \forall t\in T \tag{3.8}$$

Nebenbedingung: $x_t^{(s)}$ ist binär

Da keine Bruchteile von Sperrkombinationen gewartet werden können, müssen diese ganz oder gar nicht ausgewählt werden. Deswegen sind nur die Werte eins und null für $x_t^{(s)}$ zulässig.

$$x_t^{(s)} \in \{0,1\} \qquad \forall s\in S,\ t\in T \tag{3.9}$$

3.4.2 Restrictet Masterproblem

Im Restrictet Masterproblem (RMP) wird nicht über alle möglichen Sperrkombinationen in S optimiert sondern nur über $S'\subset S$. In S' sind die Anfangssperrkombinationen und die, durch das Pricing-Problem erzeugten Sperrkombinationen enthalten. Von den Anfangssperrkombinationen ist bereits bekannt, dass sie die obere Kostenschranke und die jährliche maximale Wartungszahl einhalten. Es macht Sinn, diese Bedingungen auch an die Sperrkombinationen, die im Pricing-Problem generiert werden, zu stellen. Auf diese Weise kann davon ausgegangen werden, dass alle Sperrkombinationen aus S' die Bedingungen (3.7) und die obere Schranke in (3.8) erfüllen. Alle anderen Nebenbedingungen werden im RMP berücksichtigt. Das RMP des Wartungszeitproblems sieht also wie folgt aus:

Restrictet Masterproblem des Wartungszeitproblems

$$\min \left(\sum_{t \in T} \sum_{s \in S'} H_\kappa \left(f^{*(s)}, \sigma^{(s)} \right) \cdot x_t^{(s)} \right)$$

3.4 :	$\sum_{s \in S} x_t^{(s)}$	$=$	1	$\forall t \in T$
3.5 :	$\sum_{t \in T} \sum_{s \in S} \left(\alpha_{b,t} \cdot \sigma_b^{(s)} \cdot x_t^{(s)} \right)$	$=$	1	$\forall b \in B$
3.6 :	$\sum_{t \in T} \sum_{s \in S} \left((1 - \alpha_{b,t}) \cdot \sigma_b^{(s)} \cdot x_t^{(s)} \right)$	$=$	1	$\forall b \in B$
3.8' :	$\sum_{s \in S} \sum_{b \in B} \left(\sigma_b^{(s)} \cdot x_t^{(s)} \cdot k_b \right)$	$\geq$	$(1 - \varepsilon^-) \cdot \bar{k} \cdot \beta_t$	$\forall t \in T$
3.9 :	$x_t^{(s)}$	$\in$	$\{0, 1\}$	$\forall s \in S',\ t \in T$

3.4.3 Pricing-Problem

Sinn und Zweck des Pricing-Problems ist es, neue Sperrkombinationen zu finden, mit denen eine Verbesserung der Lösung des RMP erreicht werden kann. Die Zielfunktion des Pricing-Problems sind die reduzierten Kosten des Masterproblems. In dieser Funktion werden die, zum RMP dualen Variablen z verwendet. Der erste Index einer dualen Variable gibt an, zu welcher Nebenbedingungsklasse des RMP diese gehört. Es wird das duale Problem zur relaxierten Version des RMP betrachtet.
Die Nebenbedingung (3.9) des RMP, die ein binäres $x_t^{(s)}$ verlangt, wird zur neuen Nebenbedingung $x_t^{(s)} \in [0, 1]$. Da die Nebenbedingung (3.4) bereits fordert, dass $x_t^{(s)}$ kleiner oder gleich eins ist, genügt es, die relaxierte Version von (3.9) auf die Bedingung $x_t^{(s)} \geq 0$ zu reduzieren.
Das duale Problem zum relaxierten RMP hat somit folgende Gestalt:

Duales Problem zum relaxierten RMP des Wartungszeitproblems

$$\max \left(\sum_{t \in T} z_{(3.4),t} + \sum_{b \in B} z_{(3.5),b} - \sum_{t \in T} (1 - \varepsilon^{-}) \cdot \bar{k} \cdot \beta_t \cdot z_{(3.8'),t} \right)$$

$$z_{(3.4),t} + \sum_{b \in B} \alpha_{b,t} \sigma_b^{(s)} z_{(3.5),b} +$$

$$+ \sum_{b \in B} (1 - \alpha_{b,t}) \sigma_b^{(s)} z_{(3.6),b} - \sum_{b \in B} \sigma_b^{(s)} k_b \; z_{(3.8'),t} \quad \geq \quad H_\kappa \left(f^{*(s)}, \sigma^{(s)} \right) \qquad \forall t \in T, \; s \in S'$$

$$z_{(3.4),t} \quad \in \quad \mathbb{R} \qquad \forall t \in T$$

$$z_{(3.5),b} \quad \in \quad \mathbb{R} \qquad \forall b \in B$$

$$z_{(3.6),b} \quad \in \quad \mathbb{R} \qquad \forall b \in B$$

$$z_{(3.8'),t} \quad \geq \quad 0 \qquad \forall t \in T$$

Die reduzierten Kosten einer Sperrkombination s, im Jahr t werden auf folgende Weise berechnet:

$$\bar{c}_t^{(s)} = \left(H_\kappa \left(f^*, \sigma \right) - z_{(3.4),t} - \sum_{b \in B} \alpha_{b,t} \sigma_b z_{(3.5),b} - \sum_{b \in B} (1 - \alpha_{b,t}) \sigma_b z_{(3.6),b} + \sum_{b \in B} \sigma_b k_b \; z_{(3.8'),t} \right)$$

Beim Lösen des Optimierungsproblems, wird für jedes Jahr ein eigenes Pricing-Problem gelöst. Die dabei gefundenen Sperrkombinationen werden dann zur Menge S' hinzugefügt. Da die Anzahl der betrachteten Jahre in der Regel überschaubar ist, bietet sich dieses Verfahren an.
Da im Pricing-Problem nach Sperrkombinationen gesucht wird, sind die σ_b, dort Variablen. Nebenbedingungen sind die jährliche Wartungsbeschränkung (3.7), sowie die Kostenbeschränkungen (3.8). Da ein Bauwerk entweder ganz oder gar nicht gewartet wird, muss σ_b darüber hinaus noch binär sein. Mit diesen Bedingungen ergibt sich zunächst folgendes Pricing-Problem für das Jahr t:

Pricingproblem ohne Flussbedingungen

$$\min \left(H_\kappa\left(f^*, \sigma\right) - z_{(3.4),t} - \sum_{b \in B} \alpha_{b,t} \sigma_b z_{(3.5),b} - \sum_{b \in B} (1 - \alpha_{b,t}) \sigma_b z_{(3.6),b} + \sum_{b \in B} \sigma_b k_b \; z_{(3.8'),t} \right)$$

$$\begin{array}{lccll}
3.7: & \sum\limits_{b \in B} \sigma_b & \leq & m & \\
3.8': & \sum\limits_{b \in B} \sigma_b k_b & \geq & (1 - \varepsilon^-) \cdot \bar{k} \cdot \beta_t & \\
3.8'': & \sum\limits_{b \in B} \sigma_b k_b & \leq & (1 + \varepsilon^+) \cdot \bar{k} & \\
& \sigma_b & \in & \{0, 1\} & \forall b \in B
\end{array}$$

Mit der Funktion $H_\kappa\left(f^*, \sigma\right)$ aus Abschnitt 3.2, wird die Summe der Fahrzeiten aller Verkehrsteilnehmer im Nutzeroptimum berechnet. Der optimale Fluss f^* für eine Sperrkombination $s \notin S'$ ist allerdings noch nicht bekannt und muss erst, wie in Abschnitt 2.2.6 beschrieben ermittelt werden. Aus diesem Grund müssen die Nebenbedingungen des Verkehrsflussproblems zum Pricing-Problem hinzugefügt werden.
Die Zielfunktion des Verkehrsflussproblems $\tilde{H}$ unterscheidet sich aber von H. Um für die praktische Anwendung eine einigermaßen einfach zu handhabende Zielfunktion zu haben, wird H einfach durch $\tilde{H}$ ersetzt. Dabei ändern sich natürlich auch die Zielfunktionswerte, sodass für jede Lösung dieses veränderten Pricing-Problems im Nachhinein überprüft werden muss, ob die ursprünglichen reduzierten Kosten tatsächlich negativ sind. Nur wenn dies der Fall ist, wird die gefundene Sperrkombination zu S' hinzugefügt.
H als Zielfunktion, oder eine Mischung aus H und $\tilde{H}$ würde in den meisten Fällen zu Flüssen führen, die nicht dem Nutzeroptimum entsprechen würden. In diesem Fall müsste, für eine gefundene Sperrkombination zuerst das Nutzeroptimum berechnet werden, bevor die eigentlichen Fahrtkosten ermittelt werden können. Ein solcher Ansatz wurde in der vorliegenden Arbeit nicht verfolgt.

Das vollständige Pricing-Problem enthält nun auch noch die Flussbedingungen:

Pricingproblem des Wartungszeitproblems

$$\min \left(H_\kappa\left(f^*, \sigma\right) - z_{(3.4),t} - \sum_{b\in B} \alpha_{b,t}\sigma_b z_{(3.5),b} - \sum_{b\in B} (1-\alpha_{b,t})\sigma_b z_{(3.6),b} + \sum_{b\in B} \sigma_b k_b \; z_{(3.8'),t} \right)$$

$$\begin{aligned}
\sum_{b\in B} \sigma_b &\leq m && \\
\sum_{b\in B} \sigma_b k_b &\geq (1-\varepsilon^-)\cdot\bar{k}\cdot\beta_t && \\
\sum_{b\in B} \sigma_b k_b &\leq (1+\varepsilon^+)\cdot\bar{k} && \\
\sigma_b &\in \{0,1\} && \forall b \in B \\
f_e &= \sum_{r\in R}\sum_{\substack{r'\in R\\ r'\neq r}} f_e^{(r,r')} && \forall e \in E \\
f_e^{(r,r')} &\geq 0 && \forall e \in E \\
& && \forall r,\ r' \in R, \text{ mit } r \neq r' \\
\sum_{u:(u,v)\in E} f_{(u,v)}^{(r,r')} &= \sum_{u:(v,u)\in E} f_{(v,u)}^{(r,r')} && \forall r,\ r' \in R, \text{ mit } r \neq r' \\
& && v \in V \setminus (L(r) \cup L(r')) \\
\sum_{v\in L(r)} \left(\sum_{u:(v,u)\in E} f_{(v,u)}^{(r,r')} - \sum_{u:(u,v)\in E} f_{(v,u)}^{(r,r')} \right) &= d_{r,r'} && \forall r,\ r' \in R, \text{ mit } r \neq r'
\end{aligned}$$

4 Implementierung und Tests

In diesem Kapitel wird die Arbeitsweise des Optimierungsprogramm, welches in der Programmiersprache C++ implementiert wurde, beschrieben. Das Restricted Masterproblem und das Pricing Problem werden mit Hilfe von GNU Octave gelöst, wobei die dafür nötigen Quelldateien vom Optimierungsprogramm immer wieder neu generiert werden. Das Verkehrsflussproblem und das Pricing Problem werden mit den nichtlinearen Solvern IPOPT, bzw. `BONMIN` gelöst. Dabei wird das C++ Interface dieser beiden Solver benutzt. Die dafür nötigen C++ Quelldateien werden bereits vor dem Durchlauf des Optimierungsprogramms, mit einem Hilfsprogramm erzeugt.
Außerdem wird im diesem Kapitel erklärt, wie man die Quelldateien des implementierten Optimierungsprogramms kompiliert. Dann wird das Programm noch an zwei Beispielen getestet.

4.1 Generierung der nichtlinearen Unterprobleme

Dieser Abschnitt beschreibt, wie das oben erwähnte Hilfsprogramm das Pricing-, bzw. das Verkehrsflussproblem erzeugt. Folgendes Beispiel dient dabei zur Veranschaulichung:

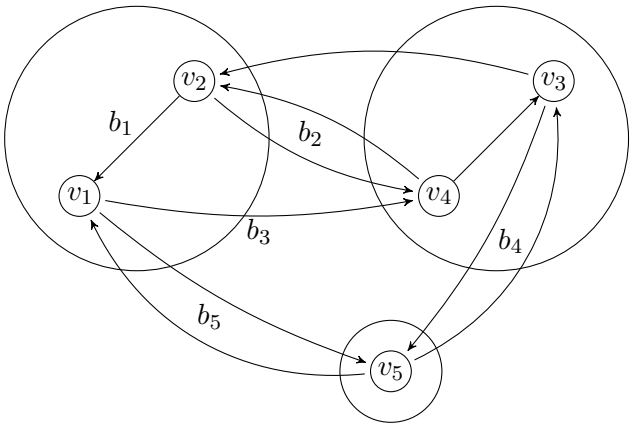

Abbildung 4.1: Die Knoten v_1 und v_2 gehören zum ersten Bezirk, die Knoten v_3 und v_4 zum zweiten Bezirk und Knoten v_5 ist der einzige Knoten des dritten Bezirks. Die Bauwerke b_1 und b_3 gehören jeweils nur zu einer Einbahnstraße, während b_2, b_4 und b_5 jeweils zu einer Straße mit zwei Fahrtrichtungen gehören. Wartungsmaßnahmen an den Bauwerken b_2, b_4 oder b_5 schränken somit die Kapazität auf zwei Kanten gleichzeitig ein.

Das Netzwerk ist ein gerichteter Graph, bestehend aus fünf Knoten und zehn Kanten. Zur Darstellung des Graphen und der Nummerierung der Kanten dient die Inzidenzmatrix:

$$\begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 & e_{10} & \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -1 & v_1 \\ 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & v_2 \\ 0 & 1 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & v_3 \\ 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & v_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & v_5 \end{pmatrix}$$

Die Kanten haben darüber hinaus noch weiter Eigenschaften, wie Kapazität, Fahrzeit im unbelasteten Zustand und die Zugehörigkeit zu Bauwerken:

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}
p_e	60	40	60	60	60	60	40	40	50	40
$t_{0,e}$	5	5	8	5	4	5	2	5	3	5
Bauwerke	b_1	.	b_2	b_2	b_3	.	b_4	b_4	b_5	b_5

Das Netzwerk ist außerdem noch in drei Bezirke unterteilt, wobei die Knoten v_1 und v_2 zum ersten Bezirk, die Knoten v_3 und v_4 zum zweiten Bezirk gehören und v_5 zum dritten Bezirk gehört. Die Verkehrsnachfrage wird mit der Nachfragematrix wie folgt dargestellt:

$$\begin{pmatrix} 0 & 100 & 120 \\ 100 & 0 & 100 \\ 100 & 120 & 0 \end{pmatrix}$$

Die Kapazitätsminderung ist $\kappa = 0,4$, das Kostenziel ist $\bar{k} = 150$, die zulässige obere Abweichung von der Kostenschranke ist $\varepsilon^+ = 0,2$ und die zulässige untere Abweichung von der Kostenschranke ist $\varepsilon^- = 0,4$. Es wird außerdem ein Zeitraum von drei Jahren betrachtet, in dem pro Jahr nicht mehr als drei Bauwerke gewartet werden dürfen. Im dritten Jahr muss die untere Kostenbeschränkung nicht mehr beachtet werden. Für die Deadlines und Reparaturkosten der Bauwerke gilt:

Bauwerk	b_1	b_2	b_3	b_4	b_5
Deadline im Jahr	3	1	2	2	3
Reperaturkosten	80	82,5	95,4	90	82

4.1.1 Vorbereitung der Generierung

Das Hilfsprogramm liest zunächst eine Textdatei ein, in welcher alle notwendigen Daten des Modells abgespeichert sind. Alle Zahlen in dieser Datei sind durch Leerzeichen oder Zeilenumbrüche getrennt. Das Hilfsprogramm prüft nicht, ob die Eingabedatei fehlerhaft ist oder sein könnte. Alle Zahlen müssen in der richtigen Reihenfolge angegeben sein. Wie diese Reihenfolge genau aussieht, wird in diesem Abschnitt erklärt. Eine solche Datei sollte mit Sorgfalt erstellt werden.
Zunächst stehen in der Eingabedatei folgende Zahlen: Die Anzahl der Kanten, die Anzahl der Bauwerke, die Anzahl der Bezirke und die Anzahl der Knoten. Diese Zahlen werden

unter anderem benötigt, damit das Programm beim Einlesen weiterer Daten genügend Speicherplatz reserviert. Die nächsten Zahlen in der Datei sind die Kapazitätsminderung κ, das Kostenziel $\bar{k}$, die obere und untere Kostenbeschränkung, ε^+ und ε^-, die Anzahl der Jahre $|T|$, sowie die Wartungsbeschränkung m. Für das obige Beispiel hat die erste Zeile der Eingabedatei folgende Gestalt:

```
10  5  3  5  0.4  150  0.2  0.4  3  3
```

Danach werden zuerst die Kapazitäten p_e und dann die Fahrzeiten $t_{0,e}$ eingelesen. Die nächsten beiden Zeilen der Datei sehen also folgendermaßen aus:

```
60  40  60  60  60  60  40  40  50  40
 5   5   8   5   4   5   2   5   3   5
```

Als nächstes wird der Graph, in Gestalt seiner Inzidenzmatrix eingelesen. Dabei geht das Programm alle Knoten und für jeden Knoten alle Kanten durch. Eine `-1` bedeutet, dass eine Kante im aktuell betrachteten Knoten endet, eine `1` bedeutet, das sie dort beginnt und eine `0` sagt aus, dass der aktuelle Knoten gar nicht an die Kante angeschlossen ist. Werden Zahlen angegeben, die weder `-1`, noch `1`, noch `0` sind, besteht die Gefahr, dass das Hilfsprogramm einen fehlerhaften Output produziert. Der nächste Abschnitt der Eingabedatei des Beispiels sieht also so aus:

```
-1   0   0   0   1   0   0   0   1  -1
 1  -1   1  -1   0   0   0   0   0   0
 0   1   0   0   0  -1  -1   1   0   0
 0   0  -1   1  -1   1   0   0   0   0
 0   0   0   0   0   0   1  -1  -1   1
```

Danach ist die Nachfragematrix an der Reihe:

```
  0  100  120
100    0  100
100  120    0
```

Dann wird angegeben, wie die Knoten den Bezirken zugeordnet sind. Das Programm liest für jeden Bezirk so viele Zahlen ein, wie es Knoten gibt. Ist die $i-$te Zahl eine `1`, so gehört der $i-$te Knoten zum aktuellen Bezirk. Ist die $i-$te Zahl eine `0`, so ist der $i-$te Knoten auch nicht dem entsprechenden Bezirk zugeordnet. Hier sind die nächsten Zeilen der Beispieldatei:

```
1  1  0  0  0
0  0  1  1  0
0  0  0  0  1
```

Als nächstes wird die Zuordnung zwischen Bauwerken und Kanten eingelesen. Diese Angabe erfolgt analog zur Zuordnung zwischen Bezirken und Knoten: Für jedes Bauwerk werden so viele Zahlen eingelesen, wie es Kanten gibt. Eine `1`, bedeutet, dass die Kante das Bauwerk benutzt, bei einer `0` ist dies nicht der Fall. In der Datei des Beispiels geht es also wie folgt weiter:

```
1  0  0  0  0  0  0  0  0  0
0  0  1  1  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0
0  0  0  0  0  0  1  1  0  0
0  0  0  0  0  0  0  0  1  1
```

Der nächste Teil der Eingabe betrifft die Deadlines. Dabei werden die Variablen $\alpha_{b,t}$, die in Abschnitt 3.1 erklärt wurden eingelesen. Die Anzahl der Variablen, die für jedes Bauwerk eingelesen werden entspricht der Anzahl der Jahre. Eine `1` bedeutet, dass das Bauwerk noch vor seiner Deadline steht. Andernfalls wird eine `0` eingelesen. Die Beispieldatei setzt sich wie folgt fort:

```
1  1  1
1  0  0
1  1  0
1  1  0
1  1  1
```

Danach werden die Baukosten der Bauwerke angegeben:

```
80  82.5  95.4  90  82
```

Schließlich wird noch festgestellt, ab welchem Jahr die untere Kostenschranke nicht mehr eingehalten werden muss. Dafür werden die β_t aus Abschnitt 3.1 eingelesen:

```
1  1  0
```

Um eine Column Generation durchzuführen ist eine Startsperrkombinationsmenge notwendig. Auch diese wird in die Eingabedatei hineingeschrieben. Dabei wird für jedes Jahr genau eine Sperrkombination eingelesen. Der letzte Teil der Datei sieht folgendermaßen aus:

```
0  1  0  1  0
0  0  1  0  0
1  0  0  0  1
```

Hier ist noch einmal der gesamte Inhalt der Eingabedatei des obigen Beispiels:

```
10  5  3  5  0.4  150  0.2  0.4  3  3
60  40  60  60  60  60  40  40  50  40
 5   5   8   5   4   5   2   5   3   5
-1   0   0   0   1   0   0   0   1  -1
 1  -1   1  -1   0   0   0   0   0   0
 0   1   0   0   0  -1  -1   1   0   0
 0   0  -1   1  -1   1   0   0   0   0
 0   0   0   0   0   0   1  -1  -1   1
  0  100  120
100    0  100
100  120    0
1  1  0  0  0
0  0  1  1  0
0  0  0  0  1
1  0  0  0  0  0  0  0  0  0
0  0  1  1  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0
0  0  0  0  0  0  1  1  0  0
0  0  0  0  0  0  0  0  1  1
1  1  1
1  0  0
1  1  0
1  1  0
1  1  1
80  82.5  95.4  90  82
1  1  0
0  1  0  1  0
0  0  1  0  0
1  0  0  0  1
```

4.1.2 Generierung des Verkehrsflussproblems

Nachdem die Informationen eingelesen wurden, erstellt das Hilfsprogramm eine C++ Quelldatei für das Verkehrsflussproblem der Anfangssperrkombinationen. Dabei handelt es sich im wesentlichen um eine, von `TNLP` abgeleitete Klasse. Bei `TNLP` handelt es sich um eine virtuelle Klasse, mit Hilfe derer im C++ Interface von IPOPT nichtlineare Optimierungsprobleme repräsentiert werden.
Die Implementierung des Verkehrsflussproblems orientiert sich dabei am Standardbeispiel für das C++ Interface von IPOPT. In dieser Klasse sind zehn Funktionen definiert. In der ersten Funktion, `get_nlp_info` werden einige grundlegende Informationen zum Optimierungsproblem angegeben. Dabei handelt es sich um die Anzahl der Variablen, die Anzahl

der Nebenbedingungen und die Zahl der Nichtnullstellen in der Jacobimatrix der Nebenbedingungen. Außerdem könnte noch die Anzahl der Nichtnullstellen in der Hessematrix der Lagrangefunktion angegeben werden. Dies ist in der vorliegenden Arbeit allerdings nicht notwendig, da IPOPT angewiesen wird, die Hessematrix der Lagrangefunktion zu approximieren. Die nächste Funktion heißt `get_bounds_info`. In dieser Funktion werden obere und untere Schranken für Variablen und Nebenbedingungen angegeben. In der Funktion `get_starting_point` werden Startwerte für die Variablen angegeben. Mit `eval_f` wird der Zielfunktionswert berechnet und die Funktion `eval_grad_f` berechnet deren Gradient. In `eval_g` werden die, zu den Nebenbedingungen gehörenden Funktionen ausgewertet. Die Schranken der Nebenbedingungen wurden bereits in `get_bounds_info` angegeben. Die Jacobimatrix der Nebenbedingungen wird durch `eval_jac_g` repräsentiert. Eine weitere Funktion ist `eval_h` mit derer die Hessematrix der Lagrangefuntion ermittelt werden könnte. Da aber wie bereits erwähnt, die Hessematrix approximiert wird, gibt das Programm eine Fehlermeldung aus, sobald diese Funktion aufgerufen wird. Zum Schluss kommt noch `finalize_solution`. Dabei handelt es sich um die Funktion, die nach erfolgreicher Optimierung die Lösung weiterverarbeitet.
Die Anzahl der Variablen wird als erstes berechnet. Dabei gibt es für jede Kante e eine Variable, die den Gesamtfluss f_e darauf beschreibt und weitere Variablen $f_e^{(r,r')}$, welche die Flüsse zwischen den einzelnen Bezirken auf dieser Kante beschreiben. Dabei ist zu beachten, dass der Fluss von Bezirk r nach r' eine Variable und der Fluss von r' nach r eine weitere Variable erhält. Der Fluss innerhalb eines Bezirks wird natürlich nicht berücksichtigt. Die Gesamtzahl der Variablen beträgt somit $|E| + (|R| \cdot (|R| - 1)) \cdot |E|$. Im obigen Beispiel sind das $10 + (3 \cdot (3 - 1)) \cdot 10 = 70$ Variablen.
Die Zielfunktion setzt sich aus $|E|$ Summanden zusammen. Das Programm erstellt jeden einzelnen dieser Summanden, wobei eventuell die Zugehörigkeit der Kanten zu Bauwerken berücksichtigt werden muss. Im Beispiel sind das zehn Summanden, von denen acht durch Wartungsmaßnahmen beeinflusst werden können. Analog werden auch die ersten $|E|$ Einträge des Gradienten ermittelt. Alle weiteren Einträge sind null, da die Variablen $f_e^{(r,r')}$ nicht explizit in der Zielfunktion vorkommen.
Neben der Zielfunktion müssen natürlich auch die Nebenbedingungen generiert werden. Die Nebenbedingungsklasse $f_e^{(r,r')} \geq 0$ muss dabei nicht in der Funktion `eval_g` oder `eval_jac_g` berücksichtigt werden, da bereits in `get_bounds_info` null als untere Schranke für alle $f_e^{(r,r')}$ angegeben wird. Alle weiteren Nebenbedingungen sind lineare, bzw. affine Funktionen, die von mehreren Variablen abhängen können.

Nebenbedingungsklasse: die richtige Zusammensetzung des Gesamtflusses

Der Gesamtfluss auf einer Kante setzt sich aus den Flüssen zwischen den einzelnen Bezirken zusammen:

$$f_e \; - \sum_{r \in R} \sum_{\substack{r' \in R \\ r' \neq r}} f_e^{(r,r')} \; = \; 0 \qquad \forall e \in E$$

Hier wird für jede Kante eine Nebenbedingung erzeugt, wobei null sowohl die obere als auch die untere Schranke ist. Von f_e werden dabei die $|R| \cdot (|R| - 1)$ Teilflüsse subtrahiert. Auf diese Weise werden $|E|$ Nebenbedingungen und $|E| \cdot (|R| \cdot (|R| - 1) + 1)$ Nichtnulleinträge in der Jacobimatrix erzeugt. Im aktuellen Beispiel sind das also zehn Nebenbedingungen und 70 Nichtnulleinträge in der Jacobimatrix.

Nebenbedingungsklasse: Flusserhaltung außerhalb von Start- und Zielbezirk

Für jeden Knoten v gilt: Alle Flüsse zwischen Bezirken, denen v nicht zugeordnet ist müssen in der selben Größe aus v hinaus fließen, wie sie hinein geflossen sind:

$$\sum_{u:(u,v) \in E} f_{(u,v)}^{(r,r')} \; - \sum_{u:(v,u) \in E} f_{(v,u)}^{(r,r')} \; = \; 0 \qquad \forall r,\ r' \in R, \text{ mit } r \neq r' \; v \in V \setminus \big(L(r) \cup L(r')\big)$$

Zur Erzeugung dieser Nebenbedingungsklasse geht das Programm dabei folgendermaßen vor: Für jeden Knoten v werden alle Paare von Bezirken gesucht, denen v nicht zugeordnet ist. Dann werden für jedes derartige Bezirkspaar (r, r') zwei Nebenbedingungen erzeugt, da der Fluss in beide Richtungen berücksichtigt werden muss. Falls eine Kante e aus v hinausführt, so wird die entsprechende Variable $f_e^{(r,r')}$ zur Nebenbedingungsfunktion addiert. Führt e in v hinein so wird $f_e^{(r,r')}$ subtrahiert. Auch hier sind obere und untere Schranke gleich null. Im obigen Beispiel ist jeder Knoten genau einem Bezirk zugeordnet. Es gibt also für jeden Konten nur ein Paar zweier Bezirke, denen er nicht zugeordnet ist. Auf diese Weise entstehen $5 \cdot 1 \cdot 2 = 10$ Nebenbedingungen. Da jeder Knoten an genau vier Kanten angeschlossen ist, entstehen pro Nebenbedingung vier Nichtnulleinträge. Insgesamt werden durch diese Nebenbedingungsklasse für das Beispiel 40 Nichtnulleinträge für die Jacobimatrix erzeugt.

Nebenbedingungsklasse: Erfüllung der Verkehrsnachfrage

Die in der Nachfragematrix definierte Verkehrsnachfrage muss erfüllt werden. Aus diesem Grund gilt für den Fluss von einem r zu einem anderen Bezirk r': Die Differenz des Flusses von r nach r' muss auf allen Knoten von r' zusammen gleich der Verkehrsnachfrage von r nach r' sein. Die Nebenbedingungsklasse lautet:

$$\sum_{v \in L(r)} \left(\sum_{u:(v,u) \in E} f_{(v,u)}^{(r,r')} \; - \sum_{u:(u,v) \in E} f_{(v,u)}^{(r,r')} \right) \; = \; d_{r,r'} \qquad \forall r,\ r' \in R, \text{ mit } r \neq r'$$

Für jeden Bezirk r' werden so viele Nebenbedingungen erzeugt, wie es weitere Bezirke gibt. Im obigen Beispiel sind das somit zwei Nebenbedingungen für jeden Bezirk und sechs Nebenbedingungen insgesamt. In einer Nebenbedingung werden dabei alle Kanten betrachtet, die aus den Bezirk hinaus oder hinein führen. Kanten, die innerhalb eines Bezirks verlaufen müssen nicht beachtet werden, da die Teilflüsse drauf im obigen Term wegfallen. Zu einer Nebenbedingungsfunktion, in welcher der Fluss von r nach r' betrachtet wird, werden alle Flüsse $f_e^{(r,r')}$, die in r' hineinführen addiert, während die Flüsse, die aus r' hinausführen subtrahiert werden. Die obere und untere Schranke einer solchen Nebenbedingung ist gleich dem entsprechenden Eintrag in der Nachfragematrix. Im Beispiel sind der erste und zweite Bezirk jeweils mit sechs Kanten nach außen verbunden und der dritte Bezirk mit vier. Somit ergeben sich $2 \cdot 6 + 2 \cdot 6 + 2 \cdot 4 = 32$ weitere Nichtnulleinträge in der Jacobimatrix.

Insgesamt hat das das Verkehrsflussproblem in diesem Beispiel 70 Variablen, 26 Nebenbedingungen und 142 Nichtnulleinträge in der Jacobimatrix. Nach erfolgreicher Berechnung des Verkehrsflusses im Nutzeroptimum werden in der Funktion `finalize_solution` die richtigen Fahrtkosten berechnet.

4.1.3 Generierung des Pricing-Problems

Im Pricing-Problem treten nicht nur die Flüsse sondern auch die Sperrzustände der einzelnen Bauwerke als Variablen auf. Da diese Variablen binär sind, reicht es nicht mehr aus, nur IPOPT zu verwenden. In diesem Fall wird `BONMIN` verwendet, da dieser Solver auch diskrete Optimierung zulässt. Das Hilfsprogramm erstellt dabei für jedes Jahr ein Pricing-Problem in Form einer C++ Klasse vom Typ `TMINLP`. Mit der virtuellen `TMINLP`-Klasse werden nichtlineare, gemischt-ganzzahlige Probleme in `BONMIN` repräsentiert.
Auch hier orientiert sich die Implementierung am Standardbeispiel für `BONMIN`. Zusätzlich zu den Funktionen aus der `TNLP`-Klasse kommen hier noch drei weitere Funktionen vor: `get_variables_types`, mit der bestimmt wird, welchen Typ die Variable hat. Eine Variable kann dabei stetig, diskret oder binär sein. In diesem Fall werden die Flüsse alle stetig gewählt und die Sperrzustände binär. In `get_variables_linearity` wird festgelegt, ob eine Variable linear oder nichtlinear ist. Die Gesamtflüsse und die Sperrzustände sind nichtlinear, da sie in der nichtlinearen Zielfunktion verwendet werden. Die Einzelflüsse sind linear, da sie ausschließlich in den linearen Nebenbedingungen vorkommen. In der Funktion `get_constraints_linearity` wird dem Solver schließlich noch mitgeteilt, dass alle Nebenbedingungsfunktionen linear sind.
Bei den Pricing-Problemen ist zu beachten, dass auch die Sperrzustände in der Zielfunktion vorkommen. Aus diesem Grund sollte diese konvex sein. Zu diesem Zweck wurde folgende

Funktion aus [Wol13] für die Zielfunktion für die Pricing Probleme verwendet:

$$\hat{H}_\kappa(f,\sigma) = \sum_{b\in B}\sum_{e\in E(b)} t_{0,e}\cdot\left(1+\frac{f_e^2}{\left(p_e-\sigma_b\left(p_e-\lfloor\kappa\cdot p_e\rfloor\right)\right)^2}\right)+\sum_{e\in E\setminus\bigcup\limits_{b\in B}E(b)} h_{t_{0,e},p_e}(f_e)$$

Diese Funktion ist konvex und nimmt für $\sigma_b \in \{0,1\}$ die selben Werte an, wie die eigentliche Fahrzeitfunktion H. Indem man an den richtigen Stellen mit $\frac{1}{3}$ multipliziert wird auch diese Funktion an die Zielfunktion des Verkehrsflussproblems angepasst und kann dann in die Zielfunktion des Pricing-Problems eingebaut werden. Wie bereits beim Verkehrsflussproblem werden nun auch für die Zielfunktion jedes Pricing-Problems die Summanden jeder einzelnen Kantenfahrzeitfunktion erstellt. Falls eine Kante zu einem Bauwerk gehört, muss die entsprechende Variable für den Sperrzustand dort berücksichtigt werden. Zusätzlich werden noch die Summanden mit den dualen Variablen zu den Termen des Verkehrsflusses hinzu addiert. Für den Gradienten muss beachtet werden, dass dort jeder Sperrzustand eine Komponente hat. In einer solchen Komponente werden alle Kantenfunktionen, die vom zugehörigen Bauwerk betroffen sind, berücksichtigt, sowie die Terme, in denen die dualen Variablen vorkommen. Die Komponenten, die für die Flüsse stehen, werden analog zum Verkehrsflussproblem gebildet.
Bei den Nebenbedingungen ist zu beachten, dass es genau zwei Funktionen mehr gibt als beim Verkehrsflussproblem. Die erste Funktion betrifft die jährlich Wartungsbeschränkung. Hier werden einfach alle Sperrvariablen aufaddiert. Die jährlich Wartungsbeschränkung m dient dabei als obere Schranke dieser Nebenbedingungsfunktion. Die zweite Funktion behandelt die Kostenschranken. Dabei werden die, jeweils mit den zugehörigen Wartungskosten multiplizierten Sperrvariablen aufaddiert. Die obere Schranke dieser Funktion ist immer die obere Kostenbeschränkung, während die untere Schranke null oder die untere Kostenbeschränkung ist. Dies hängt davon ab, ob in dem Jahr, für das gepriced wird, die untere Kostenbeschränkung eingehalten werden muss oder nicht. Die Zahl der Nichtnulleinträge der Jacobimatrix ist damit um zweimal der Anzahl der Bauwerke höher als beim Verkehrsflussproblem.
Für das obige Beispiel bedeutet das, dass es in den Pricing-Problemen $70 + 5 = 75$ Variablen, $26 + 2 = 28$ Nebenbedingungen und $142 + 2 \cdot 5 = 152$ Nichtnulleinträge in der Jacobimatrix gibt.
In der Funktion `finalize_solution` wird dann überprüft, ob die, im aktuellen Pricing-Problem gefundene optimale Sperrkombination bereits in der Menge der generierten Sperrkombinationen S' vorhanden ist. Ist dies der Fall, so wird überprüft, ob die reduzierten Kosten negativ sind. Ist dies der Fall, so wird die gefundene Sperrkombination zu S' hinzugefügt.

4.2 Funktionsweise des Optimierungsprogramms

Nachdem die Unterprobleme erzeugt wurden, kann das eigentliche Optimierungsprogramm seine Arbeit aufnehmen. Dieses muss allerdings erst mit den erzeugten Quelltexten für Verkehrsfluss- und Pricing-Problem kompiliert werden. Dazu mehr in Abschnitt 4.3. Bevor die Optimierung startet, wird erneut die Eingabedatei aus Abschnitt 4.1.1 eingelesen. Dabei werden allerdings nicht alle Daten benötigt. Die Datei sollte aber trotzdem vollständig sein, da es sonst zu Fehlern kommen kann.
Zu Beginn berechnet das Programm die Fahrzeiten aller Anfangssperrkombinationen. Dabei wird die aktuell zu berechnende Sperrkombination als globale Variable definiert, auf die das Verkehrsflussproblem zugreift. Sind die Startwerte berechnet, so wird die Funktion `Tree_Manager` aufgerufen. Dort wird zunächst ein Branch-and-Bound-Verfahren vorbereitet. Konkret bedeutet das, dass ein Branch-and-Bound-Baum, mit genau einem Knoten erzeugt wird. Bei den Daten, die in einem Knoten enthalten sind handelt es sich um die Sperrkombinationsmenge des vorigen Knotens und die dazugehörigen Fahrzeiten, sowie um einen Vektor, der angibt, wie in vorhergehenden Knoten verzweigt wurde. Die Länge dieses Vektors ist die Anzahl der Sperrkombinationen des vorigen Knotens mal die Anzahl der Jahre. Hat eine Komponente den Wert -1, so wurde gar nicht danach verzweigt, bei den Werten eins oder null wurde entsprechend auch nach eins oder null verzweigt. Die Daten eines solchen Knotens werden in einer `struct`-Variable abgespeichert. Ein Branch-and-Bound-Baum ist in diesem Fall ein Vektor aus `struct`-Variablen. Zusätzlich spielen noch drei Variablen eine wichtige Rolle. Eine davon steht für den besten, bisher gefundenen Wert einer ganzzahligen Lösung. Im Verlauf des Branch-and-Bound-Verfahrens wird immer wieder überprüft, ob es sich lohnt zu verzweigen. Dazu müssen die gefundenen Lösungen stets besser sein als die bisher beste ganzzahlige Lösung. Am Anfang erhält diese Variable den Wert unendlich, bzw. einen sehr hohen Zahlenwert. Die beiden anderen Variablen geben an, welche Sperrkombinationsmenge und welcher Lösungsvektor zur besten ganzzahligen Lösung gehören. Beide sind am Anfang leere Vektoren.
Ist der erste Knoten erstellt, so wird der Baum, sowie die Nummer des aktuellen Knotens an die Funktion `Column_Generation` übergeben. Am Anfang ist dies die Nummer Null. In dieser Funktion wird anhand der übergebenen Daten eine Column Generation durchgeführt. Dort wird die Funktion `Master_und_Dual` aufgerufen, welche mit Hilfe von GNU Octave das Restricted Master Problem und das dazu duale Problem löst. Genau genommen erstellt diese Funktion eine Datei, welche beide Probleme, für GNU Octave verständlich formuliert und den Befehl gibt diese zu lösen. Anschließend werden die Ergebnisse in zwei separaten Dateien abgespeichert, die dann in der Funktion `Column_Generation` eingelesen werden. Nachdem nun die Lösung des RMP und die dualen Variablen ermittelt wurden, beginnt das Pricing. Dazu wird die Funktion `Pricing` aufgerufen. Diese Funktion sorgt dafür, dass nacheinander die Pricing-Probleme aller Jahre gelöst werden. Findet ein Pricing-Problem eine passende Sperrkombination, so wird

diese zur aktuellen Sperrkombinationsmenge sofort hinzugefügt. Nachdem das Pricing beendet ist, wird in `Column_Generation` überprüft, ob sich die Menge der betrachteten Sperrkombinationen vergrößert hat. Ist dies der Fall, so wird mit den neuen Sperrkombinationen das RMP und das duale Problem erneut gelöst, so lange bis keine weiteren Sperrkombinationen mehr hinzugefügt werden. Dann wird überprüft, ob die Lösung des RMP besser ist, als die beste, bisher gefundene ganzzahlige Lösung. Ist dies der Fall, so wird der Lösungsvektor an die Funktion `Branching` übergeben. Dort wird überprüft, ob die Lösung ganzzahlig ist. Ist dies der Fall, so liefert die Funktion den Wert -1 zurück. Andernfalls gibt der Rückgabewert an, nach welcher Variable verzweigt werden soll. Ist also die Lösung ganzzahlig und besser als die bisherige ganzzahlige Lösung, so ersetzt sie diese. Ist sie besser, aber nicht ganzzahlig, so wird verzweigt. Das bedeutet, dass dann an den Branch-and-Bound-Baum zwei weitere Knoten angehängt werden. Danach wird die Funktion `Column_Generation` beendet.
In der Funktion `Tree_Manager` wird dann `Column_Generation` für den nächsten Knoten aufgerufen. Dies wird solange wiederholt, bis es keine weiteren Knoten mehr gibt. Am Ende gibt das Programm die beste ganzzahlige Lösung, die gefunden wurde, aus.

4.3 Handhabung der Software

Das Hilfsprogramm, welches die Quelltexte für die nichtlinearen Unterprobleme erstellt, wurde mit der C++ Standardbibliothek erstellt. Dem Compiler sollte allerdings mitgeteilt werden, dass der Standard C++11 verwendet wird. Für das Hilfsprogramm gibt es drei Quelltexte: `Problemgenerator.cpp`, `PricingErstellen.cpp` und `PricingErstellen.hpp`. Diese können mit folgendem Befehl kompiliert werden:

```
g++ -std=c++11 Problemgenerator.cpp PricingErstellen.cpp -o ProbGen
```

Damit das Programm auch ordentlich funktioniert, wird die, in Abschnitt 4.1.2 beschriebene Eingabedatei benötigt. Sie sollte den Namen `Info.txt` haben.
Um das Optimierungsprogramm benutzten zu können sollten `BONMIN`, Ipopt und GNU Octave installiert sein. Um das Programm kompilieren zu können, sollten fünf Quelltexte bereits vorhanden sein: `BuPrice.cpp`, `BuPrice.hpp`, `Main.cpp`, `Pricing.hpp` und `VProblem.hpp`. Darüber hinaus sollten noch die Quelltexte hinzugefügt werden, die mit dem Hilfsprogramm erzeugt werden. Das sind: `Pricing.cpp`, `VProblem.cpp`, `PriceJahr0.cpp`, `PriceJahr0.hpp`, `PriceJahr1.cpp`, `PriceJahr1.hpp`, usw. Darüber hinaus wird zur Kompilierung das Makefile benötigt. Dieses orientiert sich am Makefile des `BONMIN`-Standardbeispiels. Beim Makefile ist außerdem zu beachten, dass in der Zeile, an deren Anfang „`OBJS =` “ steht, die Pricing-Probleme aller Jahre berücksichtigt werden. Das kompilierte Programm sollte ebenfalls auf die Datei `Info.txt` zugreifen können.
Die Software wurde unter einem Linux-Betriebssystem getestet. Falls sie unter einem anderen Betriebssystem verwendet wird, ist darauf zu achten, dass in der Datei `BuPrice.cpp`

der Header `unistd.h` eingebunden wurde und damit zweimal der Befehl `usleep(26)` verwendet wurde. Außerdem könnte der Befehl `system("octave LP.m")`, der ebenfalls in `BuPrice.cpp` zu finden ist, unter einem anderen Betriebssystem nicht funktionieren.

4.4 Tests an Beispielen

In diesem Abschnitt wird das Beispiel aus Abschnitt 4.1 getestet und dann noch ein weiteres Beispiel. Bei beiden Beispiele ist das Optimierungsprogramm zu einem ordentlichem Ende gekommen. Es gab auch Fälle, in denen das Pricing-Problem nicht gelöst werden konnte, was dazu führte, dass das Optimierungsprogramm ohne Ergebnisse abgebrochen ist. Diese Fälle werden allerdings hier nicht erläutert.

4.4.1 Test am ersten Beispiel

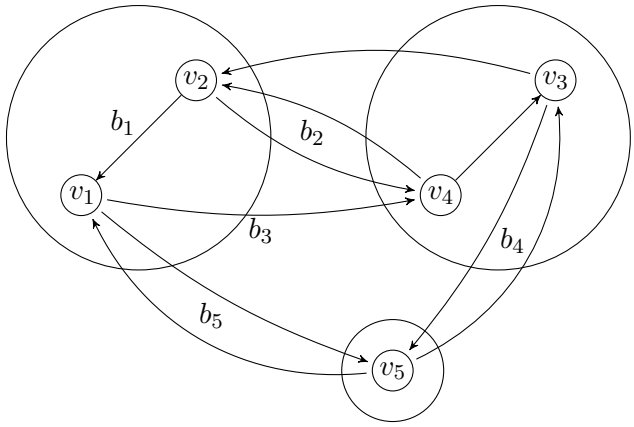

Abbildung 4.2: Alle Daten des Beispiels sind in Abschnitt 4.1.1 zu finden.

Beim ersten Testdurchlauf wurde mit folgender Anfangssperrkombination getestet: Im ersten Jahr sind b_2 und b_4 gesperrt, im zweiten Jahr b_3 und im dritten Jahr werden b_1 und b_5 gewartet. Die Fahrtkosten für die erste Sperrkombination betragen 23730, die Fahrtkosten für die zweite Sperrkombination sind 14731,8 und für die dritte Sperrkombination betragen die Kosten 22002,3. Im Durchlauf des Optimierungsprogramms wurde nur eine Sperrkombination hinzugefügt. In dieser werden b_1 und b_3 gewartet, wobei sich die Fahrtkosten auf 14865,1 belaufen. Letztlich wurde diese Kombination allerdings nicht zum Wartungsplan hinzugefügt.
Nach etwa 10,6 Sekunden war das Optimierungsprogramm fertig und präsentierte eine Lösung, in der genau die Anfangssperrkombinationen vorkommen. Die Gesamtfahrzeit beträgt 60464,1. Das Programm lieferte dabei folgenden Output:

```
*  *  *  *  *  *  *  *  *  *  *
 Kombination0  :   0 1 0 1 0
 Kombination1  :   0 0 1 0 0
 Kombination2  :   1 0 0 0 1
 Kombination3  :   1 0 1 0 0
Fahrzeit Kombination0 : 23730
Fahrzeit Kombination1 : 14731.8
Fahrzeit Kombination2 : 22002.3
Fahrzeit Kombination3 : 14865.1


LSG: 60464.1
Jahr0   1
Jahr0   0
Jahr0   0
Jahr0   0
Jahr1   0
Jahr1   1
Jahr1   0
Jahr1   0
Jahr2   0
Jahr2   0
Jahr2   1
Jahr2   0

Benötigte Zeit : 10.6214 Sekunden.
```

Die einzige, zusätzlich hinzugefügte Sperrkombination wurde im ersten Pricing-Durchlauf gefunden. Der zweite Pricing-Durchlauf konnte keine weitere geeignete Sperrkombination mehr finden. Daneben wurde beobachtet, dass die Lösung dieser ersten Column Generation bereits ein ganzzahliges Ergebnis lieferte. Auf diese Weise wurden insgesamt sechs Pricing-Probleme gelöst.
Bei einem zweiten Testdurchlauf wurde eine andere Startsperrkombination verwendet: Im ersten Jahr waren b_1 und b_2 gesperrt, im zweiten Jahr waren es b_3 und b_4 und im dritten Jahr wurde b_5 gewartet. Beim Programmdurchlauf wurden zwei weitere Kombinationen generiert, von denen ebenfalls keine für die finale Lösung verwendet wurde. Die Gesamtfahrzeit der finalen Lösung ist 59777,8 und damit besser als die Lösung des ersten Tests. Bereits dieses kleine Beispiel zeigt, dass das Verfahren nicht zwangsweise eine optimale Lösung findet.

4.4.2 Test am zweiten Beispiel

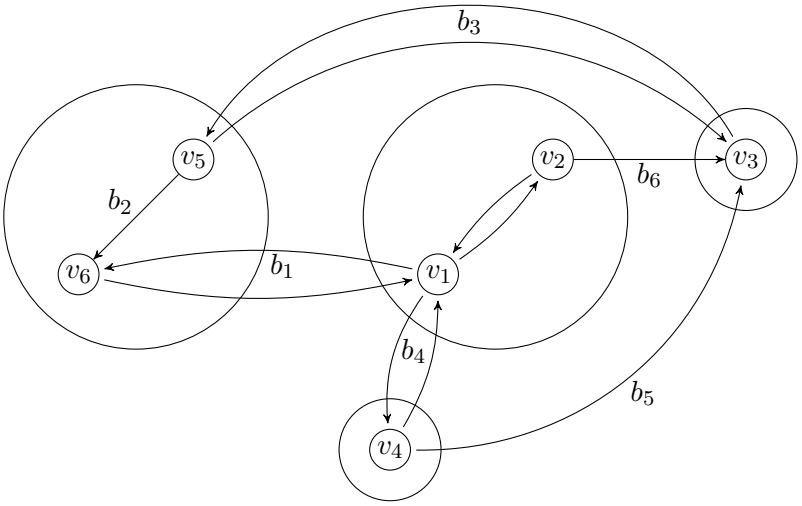

Abbildung 4.3: Dieses Beispiel hat sechs Knoten, elf Kanten, sechs Bauwerke und vier Bezirke. Die Knoten v_1 und v_2 sind dem ersten Bezirk zugeordnet, v_3 ist der einzige Knoten des zweiten Bezirks und v_4 gehört als einziger Knoten zum dritten Bezirk. Zum vierte Bezirk gehören v_5 und v_6.

Im zweiten Beispiel besteht das Netzwerk aus sechs Knoten und elf Kanten. Der Graph hat folgende Inzidenzmatrix:

$$\left(\begin{array}{cccccccccccc}
e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 & e_{10} & e_{11} & \\
-1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & v_1 \\
1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & v_2 \\
0 & 0 & 0 & 0 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & v_3 \\
0 & 0 & -1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & v_4 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & v_5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & v_5
\end{array}\right)$$

Zu den Kanten liegen folgende Informationen vor:

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}
p_e	20	20	20	30	20	20	10	30	20	40	40
$t_{0,e}$	5	5	8	5	4	5	2	5	3	5	5
Bauwerke	.	.	b_4	b_4	b_6	b_5	b_3	b_3	b_2	b_1	b_1

Die Knoten v_1 und v_2 sind dem ersten Bezirk zugeordnet, v_3 ist der einzige Knoten des zweiten Bezirks und v_4 gehört als einziger Knoten zum dritten Bezirk. Zum vierte Bezirk gehören v_5 und v_6. Für die Verkehrsnachfrage gilt:

$$\begin{pmatrix} 0 & 100 & 200 & 100 \\ 100 & 0 & 100 & 0 \\ 100 & 200 & 0 & 300 \\ 0 & 0 & 200 & 0 \end{pmatrix}$$

Der Wartungszeitraum beträgt vier Jahre. Pro Jahr dürfen nicht mehr als drei Bauwerke gewartet werden. Die Kapazitätsminderung ist $\kappa = 0,4$, das Kostenziel ist $\bar{k} = 150$, die zulässige obere Abweichung von der Kostenschranke ist $\varepsilon^+ = 0,3$ und die zulässig untere Abweichung von der Kostenschranke ist $\varepsilon^- = 0,5$. Die untere Kostenschranke muss nur in den ersten beiden Jahren beachtet werden. Für die sechs Bauwerke gilt:

Bauwerk	b_1	b_2	b_3	b_4	b_5	b_6
Deadline im Jahr	4	2	2	3	4	4
Reperaturkosten	80	82,5	95,4	90	82	85

Die Zahl der Variablen im Verkehrsflussproblem beträgt 143. Zusätzlich gibt es 59 Nebenbedingungen und 323 Nichtnulleinträge in der Jacobimatrix. Man kann also sagen, dass das Verkehrsflussproblem in diesem Beispiel mehr als doppelt so groß ist, wie im ersten Beispiel.
Für den ersten Testdurchlauf wurden im ersten Jahr b_2 und b_3, im zweiten Jahr b_4 und b_6, sowie im dritten Jahr b_1 und b_5 gewartet. Im vierten Jahr fanden keine Wartungsarbeiten statt. In diesem Beispiel benötigte das Programm etwa 15,7 Sekunden, wobei keine neuen Kombinationen hinzugefügt wurden. Die Fahrtkosten von 19558700 konnten also nicht verbessert werden. Folgender Output kam am Ende heraus:

```
*  *  *  *  *  *  *  *  *  *  *
 Kombination0  :   0 1 1 0 0 0
 Kombination1  :   0 0 0 1 0 1
 Kombination2  :   1 0 0 0 1 0
 Kombination3  :   0 0 0 0 0 0
Fahrzeit Kombination0 : 3.66263e+06
Fahrzeit Kombination1 : 8.23362e+06
Fahrzeit Kombination2 : 4.22556e+06
Fahrzeit Kombination3 : 3.43688e+06


LSG: 1.95587e+07
Jahr0   0
Jahr0   1
Jahr0   0
Jahr0   0
Jahr1   1
Jahr1   0
```

```
Jahr1   0
Jahr1   0
Jahr2   0
Jahr2   0
Jahr2   0
Jahr2   1
Jahr3   0
Jahr3   0
Jahr3   1
Jahr3   0

Benötigte Zeit : 15.7812 Sekunden.
```

In einigen Durchläufen der Column Generation wurde keine ganzzahlige Lösung gefunden, sodass das Branch-and-Bound-Verfahren insgesamt vier Verzweigungen erstellte. Das sind neun Knoten insgesamt, wobei in keinem einzigen Knoten zusätzliche Sperrkombinationen gefunden wurden. Auf diese Weise wurden während der gesamten Laufzeit 36 Pricing-Probleme gelöst. Bemerkenswert ist, dass diese große Zahl an Problemen in vergleichsweise kurzer Zeit gelöst wurden. Verglichen mit dem ersten Beispiel wurden diese Probleme deutlich schneller gelöst, obwohl sie um einiges größer sind.
Wie bereits im ersten Beispiel konnten mit einer anderen Anfangssperrkombinationsmenge bessere Fahrtkosten erreicht werden. Dabei wurde b_3 im ersten Jahr, b_2 und b_5 im zweiten Jahr, b_4 im dritten Jahr und die Bauwerke b_1 und b_6 im vierten Jahr gewartet. Im ersten Durchlauf der Column Generation wurde zwar keine Kombination hinzugefügt, dafür wurde aber eine ganzzahlige Lösung gefunden. Nach einer Berechnungsdauer von etwa 1,9 Sekunden wurden genau die Startkombinationen als Lösung ausgegeben. Die Fahrtkosten sind mit 19541600 etwas geringer als beim erstem Durchlauf.
Ein Grund warum die Pricing-Probleme so schnell gelöst wurden könnte sein, dass es für viele Verkehrsteilnehmer nur wenig sinnvolle Möglichkeiten gibt, ans Ziel zu gelangen. So gibt es vom zweiten Bezirk, also von v_3 aus, jeweils nur eine Route zu jedem anderen Bezirk, bei der keine Kante mehrmals benutzt wird.

5 Zusammenfassung und Fazit

Das Ziel der vorliegenden Arbeit war es, das Modell für Wartungsplanung in Verkehrsnetzen aus der Vorgängerarbeit [Ell12] in C++ zu implementieren. Dabei wurden die nichtlinearen Solver Ipopt und `BONMIN` verwendet, um wichtige Unterprobleme zu lösen. Im Gegensatz zur Vorgängerarbeit konnte das Optimierungsprogramm mit den Ganzzahligkeitsbedingungen des Pricing-Problems besser umgehen. Statt einer Rundungsheuristik wurden ein Branch-and-Bound-Verfahren für die Pricing-Probleme verwendet.
Das implementierte Verfahren konnte bei den Tests die Anfangssperrkombination nicht verbessern, obwohl eine Verbesserung möglich gewesen wäre. Auch in der Vorgängerarbeit trat das Problem auf, dass das Verfahren keine Verbesserungen gefunden hat. Ein weiteres Problem, das sowohl in der Vorgängerarbeit als auch in der vorliegenden Arbeit aufgetreten ist, sind Fehlschläge beim Lösen von Pricing-Problemen. Es gab Fälle, in denen das Optimierungsprogramm abbrechen musste, ohne eine Lösung zu liefern, da ein Pricing-Problem nicht gelöst werden konnte. Dieses Problem könnte behoben werden, indem man die Pricing-Probleme nicht ins Hauptprogramm einbaut, sondern jedes Einzelne davon in ein eigenes Programm auslagert, das vom Hauptprogramm aufgerufen wird. Liefert ein Pricing-Programm nach einer bestimmten Zeit keine Antwort, so sollte das Hauptprogramm den Befehl zum beenden geben und ohne Ergebnisse weitermachen. Dieser Ansatz wurde aus zeitlichen Gründen nicht mehr verfolgt.
Eine weitere Möglichkeit, wie man versuchen könnte das Optimierungsprogramm zu verbessern wäre eine Änderung der Zielfunktion der Pricing-Probleme. Diese könnte, wie in Abschnitt 3.4.3 vorgeschlagen eine Mischung aus der Fahrtkostenfunktion und der Zielfunktion des Verkehrsflussproblems enthalten. Für eine gefundene Sperrkombination muss erst das Verkehrsflussproblem gelöst werden, um dann die Fahrtkosten für die errechneten nutzeroptimalen Flüsse ermitteln zu können.
Eine ganz andere Möglichkeit das Problem zu lösen wäre, alle zulässigen Sperrkombinationen aufzustellen, und dann für jede Einzelne davon das Verkehrsflussproblem zu lösen. Durch ein solches Preprocessing erhält man ein großes lineares Optimierungsproblem, bei dem keine nichtlinearen, ganzzahligen Pricing-Probleme auftreten und das eine exakte Lösung liefert. Das Problem dabei ist, dass die Zahl der zulässigen Sperrkombinationen, und damit auch die Zahl der zu lösenden Verkehrsflussprobleme exponentiell mit der Zahl der Bauwerke ansteigt. Andererseits wurde beobachtet, dass ein Verkehrsflussproblem in der Regel um ein Vielfaches schneller gelöst wurde als ein Pricing-Problem. Außerdem stellt sich die Frage, ob es eine wirklich optimale Lösung nicht doch wert ist, eine lange Berechnungsdauer in Kauf zu nehmen.
Am Ende bleibt nur anzumerken, dass das Wartungszeitproblem nach wie vor eine intellektuelle Baustelle ist, die verhindert, dass Autofahrer weniger unter den Folgen suboptimaler Wartungsplanung zu leiden haben.

Abbildungsverzeichnis

2.1 Pseudocode für Columnn Generation 7
2.2 Beispiel für einen Branch-and-Bound-Baum 8
2.3 Skizze zum Beispiel von Pigou 12
2.4 Paradoxon von Braess - erste Skizze 13
2.5 Paradoxon von Braess - zweite Skizze 13

4.1 Erstes Beispiel, zur Erklärung 29
4.2 Erstes Beispiel, zum Test 40
4.3 Zweites Beispiel 42

Literaturverzeichnis

[AMO93] Ahuja, Ravindra K., Magnanti, Thomas L. und Orlin, James B.: *Network flows*. Upper Saddle River, NJ, Prentice Hall, 1993.

[AKM04] Achterberg, Tobias, Koch, Thorsten und Martin, Alexander: *Branching rules revisited*. Operations Research Letters 33 (2005) 42–54.

[Bon13] Bonami, Pierre und Lee, John: *BONMIN Users' Manual*, Mai 2013.

[Des05] Desaulniers, Guy: *Column generation*. New York, NY, Springer, 2005.

[DSD84] Desrosiers, J., F. Soumis und M. Desrocher: *Routing with time windows by column generation*. Networks, 14:545–565, 1984.

[Ell12] Ellssel, Susanne: *Optimale Terminplanung für Instandsetzungsarbeiten an der Verkehrsinfrastruktur*. Diplomarbeit, Technische Universität München, 2012.

[Ger07] Gerdts, Matthias: *Optimierung*. 2007.

[KLW11] Kawajir, Yoshiaki, Laird, Carl und Wächter, Andreas: *Introduction to Ipopt: A tutorial for downloading, installing, and using Ipopt.*, 2011.

[Pap98] Papadimitriou, Christos und Steiglitz, Kenneth: *Combinatorial optimization*, Dover Publ., 1998.

[PTV10] PTV AG: *Visum 11.5 - Grundlagen*, 2010.

[PTV11] PTV AG: *Visum 11.52 - Benutzerhandbuch*, Februar, 2011.

[Rou05] Roughgarden, Tim: *Selfish Routing and the Price of Anarchy*. MIT Press, 2005.

[Ulb11] Ulbrich, Michael und Ulbrich, Stefan: *Nichtlineare Optimierung*. Birkhäuser, 2011.

[Ulb12] Ulbrich, Michael: *Grundlagen der konvexen Optimierung*. Vorlesungsskript, Technische Universität München, Juli 2012.

[Wol13] Wolter, Jens: *Analyse eines Verkehrsmodells für die Planung von Brückenwartungsarbeiten*. Masterarbeit, Technische Universität München, 2013.

[Yin97] Ye, Yinyu: *Interior Point Algorithms: Theory and Analysis*. New York [u.a.], Wiley, 1997.