SCHNELLE ALGORITHMEN FÜR RESSOURCENBESCHRÄNKTE KÜRZESTE WEGE IN VERKEHRSNETZEN

Diplomarbeit
bei Prof. Dr. R. H. Möhring

vorgelegt von Fabian Zenzinger
am Fachbereich Mathematik der
Technischen Universität Berlin

Berlin, 13. Mai 2002

Die selbständige und eigenhändige Anfertigung dieser Arbeit versichere ich an Eides Statt.

Berlin, den 13. Mai 2002

Fabian Zenzinger

Vorwort

Die Anzahl registrierter Autos hat in vielen Industrienationen inzwischen einen kritischen Stand erreicht. Da sie allem Anschein nach weiter wachsen wird, die Infrastruktur jedoch nicht mehr beliebig ausbaubar ist, droht aufgrund der daraus folgenden Verkehrsdichte schon bald ein rapides Zunehmen an Staus.

Um dies zu vermeiden, versuchen einige Automobilhersteller seit einiger Zeit, sogenannte Navigationssysteme für ihre Wagen zu entwickeln, mit denen die Verkehrsteilnehmer möglichst schnell durch den Verkehr geleitet werden sollen. Der nächstliegende Ansatz bestand zuerst darin, für jeden Teilnehmer den schnellsten Weg von seinem Start- zu seinem Zielort zu berechnen. Dies lässt sich mathematisch durch das sogenannte *Kürzeste-Wege-Problem* mit nicht-negativen Kantengewichten modellieren, welches in polynomialer Zeit lösbar ist. Wie sich jedoch schon bald herausstellte, sind in der Praxis weitere Komponenten zu berücksichtigen, die die Berechnung eines für jeden Fahrer akzeptablen Weges erschweren. So wäre es zum Beispiel denkbar, bei der Berechnung des schnellsten Weges zu fordern, dass der Fahrer keinen allzu grossen Umweg zu nehmen hat. Ein solcher *ressourcenbeschränkter* kürzester Weg lässt sich leider nicht in polynomialer Zeit ermitteln, da es sich dabei um ein sogenanntes *schwach NP-vollständiges* Problem handelt.

Ziel der vorliegenden Arbeit ist es, Algorithmen zu entwickeln, die dieses Problem möglichst schnell lösen, und zu untersuchen, welche am besten für den Einsatz in einem solchen *Route-Guidance*-System geeignet sind. Zu diesem Zweck wird der für das klassische Kürzeste-Wege-Problem häufig benutzte *Dijkstra-Algorithmus* an die neue Problemstellung angepasst und in mehreren Varianten mit unterschiedlichen Beschleunigungsmethoden implementiert. Diese werden dann auf verschiedenen Beispielinstanzen sowohl untereinander als auch im Vergleich mit für andere Projekte verwendeten Lösungsverfahren getestet. Anhand der daraus folgenden Ergebnisse werden dann noch einmal zusammenfassend die Vorzüge und Nachteile der jeweiligen Ansätze diskutiert, bevor wir abschließend vorschlagen, welches Verfahren für den Gebrauch als Unterproblem in einem an der *TU Berlin* entwickeltes Navigationssystem vorzuziehen ist.

Um dem Leser die Möglichkeit zu geben, die durch die zusätzliche Ressourcenbeschränkung vorgenommenen Änderungen am Ausgangsalgorithmus nachzuvoll-

ziehen, werden in einer Einführung (Kapitel 2) noch einmal das Kürzeste-Wege-Problem sowie der Dijkstra-Algorithmus skizziert. Für ein besseres Verständnis setzt die Arbeit lediglich Grundkenntnisse in Graphentheorie und Optimierung voraus.

Ein besonderes Dankeschön geht an Dr. Ekkehard Köhler für die Betreuung während der Erstellung dieser Arbeit, an meine Korrekturleser für das Aufspüren von Fehlern und an die anderen Diplomanden der Arbeitsgruppe für die hilfreichen Ratschläge während der Programmierung der Algorithmen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Kürzeste-Wege-Probleme entstehen immer dann, wenn Güter oder Daten in Netzwerken so schnell oder so günstig wie nur möglich von einem Punkt zu einem anderen gelangen sollen. Meistens reicht es aus, die Wege nur bezüglich einer Komponente zu betrachten, sei diese nun Zeitaufwand, Weglänge oder auch entstandene Kosten.

In der Praxis gibt es jedoch auch häufig komplexere Fragenstellungen; hin und wieder werden Wege gesucht, die hinsichtlich mehr als nur einer Komponente effizient sein sollen. Dabei passiert es häufig, dass die daraus resultierenden Ziele im Widerspruch zueinander stehen und deswegen nicht gleichzeitig optimiert werden können. Dies ist insbesondere bei betriebswirtschaftlichen Erwägungen der Fall. Ein Lastwagenfahrer hat zum Beispiel seine Waren so schnell wie möglich zum Zielort zu befördern, muss bei seiner Routenwahl aber auch darauf achten, dass er nicht zuviel Maut und Benzin zu zahlen hat. In solchen Fällen muss entschieden werden, welcher Parameter am wichtigsten ist. Es wird dann *eine* Komponente optimiert, während alle anderen eine gewisse Schranke nicht überschreiten dürfen.

In dieser Arbeit wird der Fall betrachtet, in dem es zusätzlich zu den zu minimierenden Kosten genau *eine* andere Ressource gibt, die unter einem vorgegebenen Wert bleiben muss. Dieses Kapitel zeigt einige Verwendungsmöglichkeiten aus der Praxis für dieses Problem auf, wobei wir genaueres Augenmerk auf die Projektumgebungen legen, in denen die hier erstellten Algorithmen getestet werden. Außerdem wird darauf eingegangen, welche besonderen Eigenschaften die Lösungen des Problems haben müssen.

1.1 Anwendungsgebiete

Die Suche nach einem günstigen Weg bezüglich zweier Komponenten tritt in einer Großzahl von Praxisbeispielen auf. So ergibt sich in Kommunikationsnetzwerken

der Fall, dass man sowohl günstig als auch relativ sicher Daten von einem Punkt zu einem anderen senden will. Dieses sogenannte *Quality of Service*-Problem wird unter anderen von Orda [15] betrachtet. Gerade durch die in den letzten Jahren vorangetriebene Entwicklung des Internets hat diese Fragestellung an Wichtigkeit gewonnen. Insbesondere in *IP*-basierten Netzwerken gibt es eine Vielzahl an Parametern, die als Nebenbedingung in Frage kommen, so dass *IT*-Firmen daran interessiert sind, eine Software zu entwickeln, welche für jede Systemumgebung die relevanten Parameter ermittelt und diese dann als Kostenarten für ein Kürzeste-Wege-Problem mit Nebenbedingung übernimmt.

Doch auch in traditionellen Industriebranchen gibt es interessante Einsatzmöglichkeiten. Elimam und Kohler [6] untersuchen zunächst die optimale Abfolge von Abwasserflüssen einer gleichzeitig von einer industriellen und einer sanitären Organisation benutzten Einrichtung in Kuwait und gehen danach auf das Problem ein, Gebäude möglichst günstig zu bauen unter der Voraussetzung, dass sie den staatlichen Sicherheitsvorschriften der USA entsprechen. Sie stellen dabei fest, dass sich beide Fragestellungen als ressourcenbeschränkte Kürzeste-Wege-Probleme modellieren lassen.

Auch Spaltengenerierungsmethoden in der linearen Optimierung benutzen häufig das Kürzeste-Wege-Problem mit Nebenbedingung als Unterproblem. Dieser Fall tritt zum Beispiel im *Scheduling*-Bereich oder auch bei komplexen Flußproblemen auf. Sowohl Lübbecke und Zimmermann [13] in ihrer Betrachtung der Umlaufplanung im Güterverkehr bei Werks- und Industriebahnen als auch Borndörfer und Löbel [4] bei ihrer Arbeit über Terminplanung im Personenverkehr greifen darauf zurück.

Wie man sieht, gibt es eine große Anzahl von Situationen, in denen eine Nebenbedingung zusätzlich zu der zu minimierenden Komponente auftritt.

1.2 Das CMCF Route-Guidance-Projekt an der TU Berlin

Navigationssysteme, die einfach für jeden Wagen den kürzesten Weg von Start- zu Zielort berechnen, haben den fundamentalen Nachteil, dass sie nicht die Auswirkungen ihrer eigenen Routenwahl auf das Verkehrsnetz betrachten. Aus diesem Grund funktioniert ein solches System nur, solange es von einer kleinen Zahl von Fahrern benutzt wird; denn sollte eine zu große Anzahl von Autos denselben schnellen Weg benutzen, so würde auf dieser Straße schon bald ein Stau entstehen, der diesen Vorteil zunichte machen würde. Wie Beccaria und Bolelli [3] vorschlagen, ist es also ratsam, ein System zu entwickeln, welches die Verkehrsnetzbelastung minimiert, aber gleichzeitig die individuellen Routen für jeden Fahrer erträglich hält. Diese Nebenbedingung ist wichtig, da es sich bei von Navigationssystemen errechneten Wegen nur um Vorschläge handelt und sich kaum ein

Fahrer freiwillig auf eine für ihn inakzeptable Route einlassen würde, nur weil es der Gesamtheit des Verkehrsflusses nützt. In diesem Projekt wird dies durch eine Längenbeschränkung für jeden berechneten Weg umgesetzt, d.h. jede Route darf nur um eine vorgebene Prozentzahl größer als der bezüglich der Länge optimale Weg sein.

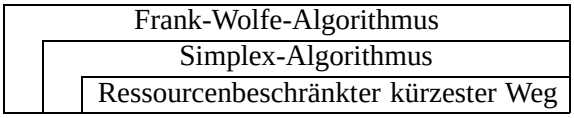

Tabelle 1.1: Einbettung des ressourcenbeschränkten Kürzeste-Wege-Problems in das Hauptproblem beim CMCF-Projekt

Das ressourcenbeschränkte Kürzeste-Wege-Problem wird somit zu einem Teilproblem des sogenannten *Constrained Multi-Commodity Flow* Problems (CMCF), welches während des Programmablaufs wiederholt aufgerufen wird. Genau genommen wird dabei eine Adaptation des sogenannten *Frank-Wolfe*-Algorithmus angewendet. Das Ausgangsproblem wird linearisiert und daraufhin mit dem *Simplex*-Algorithmus für lineare Optimierungsprobleme bearbeitet. Da dabei exponentiell viele Variablen entstehen, werden diese nicht alle gespeichert, sondern bei Bedarf mit der Spaltengenerierungsmethode erzeugt. Wie bereits erwähnt, wird dabei das Kürzeste-Wege-Problem mit Nebenbedingung als Unterproblem aufgerufen.

Als Lösungsansatz wird hier bisher eine mit einigen Beschleunigungsmethoden versehene Erweiterung des *Dijkstra*-Algorithmus verwendet. Jahn, Möhring und Schulz [10] stellen fest, dass bei der derzeitigen Implementation rund 95% des Gesamtaufwands mit der Ermittlung dieser Wege verbracht werden; ein besserer Algorithmus würde also zu großen Zeitersparnissen führen.

1.3 Das CNOP-Paket

Das *Constrained Network Optimization Package* wurde an der Universität des Saarlandes in Saarbrücken entwickelt. Es behandelt Optimierungsprobleme in Netzwerken, welche durch die Einführung einer Ressourcenbeschränkung nicht mehr in polynomialer Zeit lösbar sind. Diese werden in einem ersten Schritt durch obere und untere Schranken approximiert, bevor aus dem übriggebliebenen Bereich die Lösung ermittelt wird. Es werden hierbei mehrere Problemstellungen betrachtet; neben dem *Table Layout Problem*, Kurvenapproximationen und der Suche nach einem kostenminimalen aufspannenden Baum mit Nebenbedingung wird auch hier auf ein Routenplanungsmodell eingegangen, welches das ressourcenbeschränkte Kürzeste-Wege-Problem beinhaltet. Das verwendete Modell ist relativ simpel, so dass sich das für diese Arbeit relevante Teilproblem auch autonom ausführen lässt, was den Vergleich mit anderen Implementationen erleichtert.

Ziegelmann [21] hat sowohl für die Ermittlung der Schranken als auch für das darauffolgende Suchen der Optimallösung mehrere Algorithmen miteinander verglichen und hat schließlich mit seinen besten Ansätzen bei selbst durchgeführten Tests sehr gut abgeschnitten. Es bleibt jedoch die Frage, ob dieses Verfahren generell zu sehr schnellen Resultaten führt oder ob es für gewisse Situationen wie zum Beispiel dem Einsatz als Unterprogramm im *CMCF*-Projekt ungeeignet ist.

1.4 Anforderungen an die Problemlösungen

Hauptziel der vorliegenden Arbeit ist es letztlich zu untersuchen, ob ein schnellerer Lösungsansatz für das ressourcenbeschränkte Kürzeste-Wege-Problem zu einer spürbaren Reduzierung der Gesamtlaufzeit beim *CMCF*-Projekt führen kann. Um dies zu ermitteln, vergleichen wir unsere neu erstellten Algorithmen mit den in den letzten beiden Abschnitten vorgestellten Ansätzen. Voraussetzung dafür ist jedoch, dass alle Verfahren genau das gleiche Problem lösen. Deswegen sollten unsere Methoden den Anforderungen der vorgestellten Routenplanungssysteme genügen.

Da die Applikation des CNOP-Pakets eher einfach aufgebaut ist, sind für diesen Fall keine zusätzlichen Bedingungen zu beachten. Gesucht wird ein bezüglich einer Kostenkomponente minimaler Weg zwischen zwei Punkten, der hinsichtlich einer anderen Kostenart eine vorgegebene Schranke nicht überschreitet.

Komplexer sieht es beim CMCF-Problem aus. Hier wird nicht nur der kostenminimale ressourcenbeschränkte Weg gesucht, sondern es reicht teilweise auch, wenn er für beide Kostenarten unter vorgegebenen Schranken liegt. Ein hinsichtlich einer Komponente optimierter Weg würde zwar diese Bedingung erfüllen, jedoch kann man davon ausgehen, dass dessen Ermittlung mit einem zeitlichen Mehraufwand verbunden sein dürfte.

Zusätzlich zu dieser Anforderung besitzen einige der neu erstellten Algorithmen die Eigenschaft, eine gewisse Anzahl an effektiven Wegen zu ermitteln, wobei die Effektivität darin besteht, dass ein Weg, der hinsichtlich der ersten Komponente etwas schwächer ist, dafür einen günstigeren Wert bezüglich der zweiten Komponente vorzuweisen hat. Dies ist zum Beispiel für Fragestellungen von Interesse, in denen mehrere alternative Wegvorschläge benötigt werden.

Somit ergeben sich drei verschiedene Problemstellungen für diese Arbeit:

- Berechnung eines bezüglich einer Kostenkomponente minimalen Weges bei Einhaltung einer oberen Schranke für eine zweite Komponente
- Berechnung eines Weges, der bezüglich beider Komponenten unterhalb der vorgegebenen Schranken liegt
- Berechnung mehrerer gleichermaßen effektiver Wege, die allesamt unterhalb einer vorgegebenen Schranke für eine Kostenart liegen

Da versucht werden soll, den für das klassische Kürzeste-Wege-Problem mit nichtnegativen Kantengewichten normalerweise benutzten Dijkstra-Algorithmus so zu erweitern, dass diese drei Probleme effizient gelöst werden, gehen wir im folgenden Kapitel zunächst auf grundlegende Eigenschaften des Algorithmus von Dijkstra ein und beschreiben für ihn einige Beschleunigungsmöglichkeiten, bevor wir die zur Erweiterung benötigten Schritte erläutern.

Kapitel 2

Beschleunigungsmethoden für das Kürzeste-Wege-Problem mit nicht-negativen Kantengewichten

Im folgenden Kapitel werden einige für den weiteren Verlauf der Arbeit relevante Eigenschaften des Kürzeste-Wege-Problems zusammengetragen. Zusätzlich zur allgemeinen Problemstellung und zum Dijkstra-Algorithmus wird auch auf mögliche Beschleunigungsmethoden zur Ermittlung der Lösung eingegangen.

2.1 Problemstellung

Ein Verkehrsnetz lässt sich als ein gerichteter Graph modellieren, indem man jede Straße als Kante und jede Kreuzung als Knoten darstellt. Weist man nun jeder Kante einen Kostenwert zu, so entspricht die Suche nach dem günstigsten Weg zwischen zwei Orten der Ermittlung des kürzesten Weges zwischen zwei Knoten auf dem erstellten Graphen.

Sei also ein gerichteter Graph $G = (V, E)$ gegeben, und für jede Kante $e \in E$ existiere ein nicht-negatives Kantengewicht c_e. Dann lässt sich das Kürzeste-Wege-Problem von einem gegebenen Knoten $s \in V$ zu einem gegebenen Knoten $t \in V$ schreiben als

$$\min \sum_{e \in p} c_e,$$

so dass p ein Weg von s nach t ist.

Da wir uns auf den Fall mit nicht-negativen Kantengewichten beschränken, stellen wir nun den Dijkstra-Algorithmus vor, der die Standardmethode zur Lösung eines solchen Problems ist.

2.2 Der Dijkstra-Algorithmus

2.2.1 Formulierung

Der Dijkstra-Algorithmus ist relativ simpel aufgebaut, löst das Kürzeste-Wege-Problem jedoch in polynomialer Zeit. In einer Initialisierungsphase wird zunächst allen Knoten ein Distanzwert zugewiesen, der dem bisher gefundenen kürzesten Weg vom Startknoten zum betrachteten Knoten entspricht. Somit bekommt der Startknoten s die Distanz 0 und alle anderen Knoten bekommen die Distanz $+\infty$ zugewiesen. Zusätzlich benötigen wir ein Vorgängerarray, welches zunächst für jeden Knoten mit $\emptyset$ initialisiert wird.

In der Hauptschleife wird nun der unmarkierte Knoten v mit der kleinsten endlichen Distanz gesucht und markiert; im ersten Schritt ist dies automatisch der Startknoten. Anschließend werden alle Knoten w betrachtet, für die eine Kante $e = (v,w)$ existiert. Sollte die Distanz von w größer als die Distanz von v plus dem Kantengewicht c_e sein, so wird sie mit dem kleineren Wert aktualisiert, da wir damit einen günstigeren Weg zu diesem Knoten gefunden haben. Im Vorgängerarray wird in diesem Fall dem Knoten w der Knoten v zugewiesen, da dieser auf dem derzeit kürzesten s-w-Weg der direkte Vorgänger von w ist.

Input : Gerichteter Graph $G = (V,E)$, nicht-negative c_e für alle $e \in E$.
Start- und Zielknoten $s,t \in V$.
Output : Ein kürzester Weg von s nach t in G, falls dieser existiert.

foreach $v \in V$ **do** $Vorgänger(v) \leftarrow \emptyset$;
foreach $v \in V \setminus \{s\}$ **do** $dist(v) \leftarrow +\infty$;
$dist(s) \leftarrow 0$;
while *Es existiert unmarkiertes v mit* $dist(v) < +\infty$. **do**
  Markiere unmarkiertes v mit minimaler Distanz ;
  if $v = t$ **then return** $dist(t)$ *und den zugehörigen s-t-Weg*;
  foreach *Kante* $e = (v,w) \in E$ **do**
    if $dist(v) + c_e < dist(w)$ **then**
      $dist(w) \leftarrow dist(v) + c_e$;
      $Vorgänger(w) \leftarrow v$;
return *„Es gibt keinen s-t-Weg.“*;

Algorithmus 1: Der Dijkstra-Algorithmus

Der Algorithmus terminiert, sobald entweder der Zielknoten t markiert, also der kürzeste Weg gefunden ist, oder aber kein unmarkierter Knoten mit endlicher Distanz mehr existiert. In diesem Fall gibt es keinen Weg von s nach t. Falls eine Lösung gefunden wird, so lässt sich anhand des Vorgängerarrays der kürzeste s-t-Weg ermitteln.

2.2.2 Korrektheit und Laufzeit

Die Korrektheit des Algorithmus basiert auf folgender Festellung:

Satz 2.1. *Ist ein Knoten v markiert, so entspricht die ihm zugewiesene Distanz dem kürzesten Weg vom Startknoten nach v.*

Beweis. Wir führen einen Beweis durch Widerspruch aus. Sei also $v \in V \setminus \{s\}$ der erste Knoten, auf den diese Eigenschaft nicht zutrifft. Dann gibt es einen Weg $p = (s, \ldots, u, \ldots, v)$, dessen Länge kleiner als $dist(v)$ ist. Wären alle Knoten auf diesem Weg bereits markiert, so hätte v diese Länge als Distanzwert bekommen; es muss also mindestens ein unmarkierter Knoten vorhanden sein. Sei u der erste unmarkierte Punkt auf der Wegstrecke.

Der Distanzwert von u kann nur noch von einem über einen bisher unmarkierten Knoten führenden Weg verringert werden. Da jedoch gerade v markiert worden ist, würde kein solcher Weg eine kleinere Länge als der bisherige Distanzwert von u erreichen. Somit muss der Distanzwert von u schon der Länge $\delta(s,u)$ des kürzesten Weges von s nach u entsprechen. Es gilt also:

$$dist(u) = \delta(s,u) \leq \delta(s,v) < dist(v)$$

Wir haben jedoch gerade v als kleinsten unmarkierten Distanzwert ermittelt, woraus ein Widerspruch folgt. □

Da der Algorithmus abbricht, wenn der Zielknoten markiert ist, folgt aus dem Satz unmittelbar:

Korollar 2.2. *Der Dijkstra-Algorithmus arbeitet korrekt und terminiert.*

Es ergibt sich hieraus auch eine andere für den weiteren Verlauf der Arbeit wichtige Folgerung:

Korollar 2.3. *Lässt man den Dijkstra-Algorithmus solange weiterlaufen, bis kein unmarkierter Knoten mit endlicher Distanz mehr existiert, so entsprechen die ermittelten Distanzen für jeden markierten Knoten $v \in V$ dem kürzesten Weg vom Startknoten zu ihm. Zu allen bei Programmende unmarkierten Knoten gibt es keinen Weg vom Startknoten aus.*

Dies bedeutet, dass die Berechnung aller kürzesten Wege von einem Startknoten aus eine Laufzeit gleicher Größenordnung wie die Berechnung eines kürzesten Weges von einem Start- zu einem Zielknoten hat.

Generell hängt die Laufzeit von der bei der Implementierung verwendeten Datenstruktur zur Verwaltung der unmarkierten Knoten und ihrer Distanzen ab. Wie in Tabelle 2.1 zu sehen ist, ist die Laufzeit aber polynomial.

Operation	Datenstruktur		
	ARRAY	HEAP	FIBONACCI-HEAP
Minimum entfernen	$O(n)$	$O(\log n)$	$O(\log n)$
Einfügen	$O(1)$	$O(\log n)$	$O(1)$
Geänderten Wert neu einfügen	$O(1)$	$O(\log n)$	$O(1)$
Initialisierung	$O(n)$	$O(1)$	$O(1)$
Laufzeit	$O(n^2)$	$O(m \cdot \log n)$	$O(m + n \cdot \log n)$

Tabelle 2.1: Aufwand des Dijkstra-Algorithmus bei Verwendung verschiedener Datenstrukturen

2.3 Beschleunigungsmethoden für den Dijkstra-Ansatz

Wie im vergangenen Abschnitt beschrieben wurde, ist der Dijkstra-Algorithmus im Grunde genommen darauf spezialisiert, kürzeste Wege von einem Start- zu mehreren Zielknoten zu berechnen. Möchte man nur den Weg zu einem bestimmten Knoten berechnen, so ändert sich der eigentliche Verlauf des Verfahrens nicht. Weiterhin breitet sich der Dijkstra-Algorithmus mehr oder weniger kreisförmig vom Startknoten aus, um in alle Richtungen kürzeste Wege zu erstellen, unabhängig davon, wo sich der Zielknoten befindet.

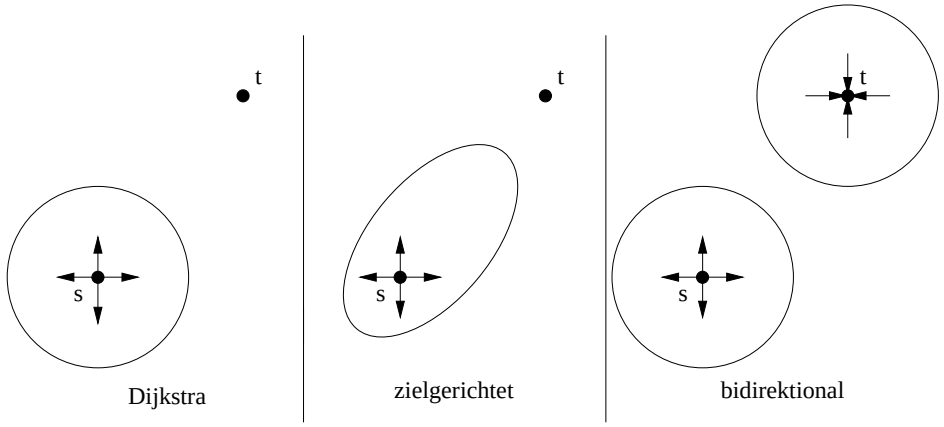

Abbildung 2.1: Der Suchverlauf der verschiedenen Ansätze im Vergleich

Um die Suche effizienter zu gestalten, scheint es also naheliegend, sich Methoden zu überlegen, welche entweder zusätzlich vom Zielknoten aus operieren oder sich etwas zielgerichteter in dessen Richtung verbreiten. Möglicherweise lässt sich der sich auf diese Weise ergebende Beschleunigungseffekt noch weiter verstärken, wenn beide Ideen miteinander kombiniert werden.

2.3.1 Der zielgerichtete Ansatz

Der zielgerichtete Ansatz versucht, den kürzesten Weg schneller zu finden, indem er Teilwege, die offensichtlich zu einem langem s-t-Weg führen, nicht weiter betrachtet. Dies wird erreicht, indem die Kantengewichte transformiert werden, so dass genau die Kanten, welche in die Richtung des Zielknotens führen, vom Algorithmus bevorzugt werden, während solche, die in die entgegengesetzte Richtung laufen, mit einem ungünstigeren Kostenwert bestraft werden.

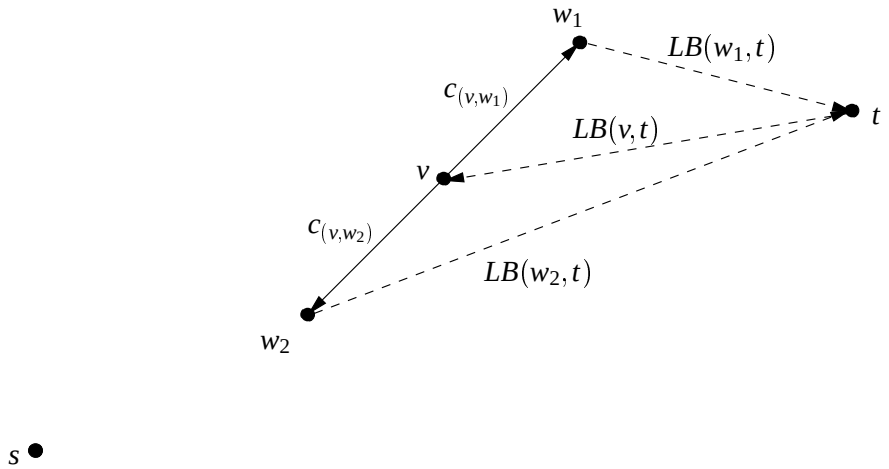

Abbildung 2.2: Transformation der Kantengewichte.

Zur korrekten Ausführung werden untere Schranken $LB_{(v,t)}$ für die kürzesten Wege von allen Knoten $v \in V$ zum Zielknoten t benötigt. Die folgenden modifizierten Kantenbewertungen werden dann benutzt:

$$c'_{(v,w)} := c_{(v,w)} - LB_{(v,t)} + LB_{(w,t)} \tag{2.1}$$

Wie wir in Abbildung (2.2) gut erkennen können, wird somit der gewünschte Effekt erreicht. Die Kante (v,w_1) wird gegenüber (v,w_2) bevorzugt behandelt, da die untere Schranke für den restlichen w_1-t-Weg kleiner als $LB(w_2,t)$ ist.

Der Dijkstra-Algorithmus funktioniert wie bereits erwähnt nur für nicht-negative Kantengewichte. Möchte man deswegen das Verfahren mit den transformierten Kantenkosten benutzen, so muss für die unteren Schranken folgende *Konsistenzbedingung* gelten:

$$c_{(v,w)} + LB_{(w,t)} \geq LB_{(v,t)} \tag{2.2}$$

Diese Eigenschaft wird von vielen Bewertungsfunktionen erfüllt. Bei der Suche nach einem Weg mit der kürzesten Länge wäre zum Beispiel der geographische Abstand zwischen den jeweiligen Knoten als untere Schranke benutzbar.

Wenden wir nun den Dijkstra-Algorithmus auf die neuen Kantengewichte an, so liefert er einen kürzesten *s*-*t*-Weg der Länge $dist'(t)$. Es lässt sich zeigen, dass dies auch der kürzeste Weg bezüglich der ursprünglichen Kosten $dist(t)$ ist.

Satz 2.4. *Seien auf einem Graphen $G = (V,E)$ für Wege von jedem Knoten v zum Zielknoten t untere Schranken $LB(v,t)$ gegeben, die die* Konsistenzbedingung (2.2) *erfüllen. Wird der Dijkstra-Algorithmus mit anhand der* Transformation (2.1) *modifizierten Kantengewichten gestartet, so ist die ermittelte Lösung ein kürzester s-t-Weg bezüglich der ursprünglichen Kantengewichte, und für die Kosten des Weges gilt:*

$$dist(t) = dist'(t) + LB_{(s,t)} - LB_{(t,t)}$$

Beweis. Sei $p = (s, v_1, v_2, \ldots, v_n, t)$ ein *s*-*t*-Weg. Für seine Länge gilt:

$$\begin{aligned} dist'(t) &= c'_{(s,v_1)} + c'_{(v_1,v_2)} + \ldots + c'_{(v_n,t)} \\ &= c_{(s,v_1)} - LB_{(s,t)} + LB_{(v_1,t)} \\ &+ c_{(v_1,v_2)} - LB_{(v_1,t)} + LB_{(v_2,t)} \\ &+ \ldots \\ &+ c_{(v_n,t)} - LB_{(v_n,t)} + LB_{(t,t)} \\ &= dist(t) - LB_{(s,t)} + LB_{(t,t)} \end{aligned}$$

Die Länge eines *s*-*t*-Weges unterscheidet sich also bei Benutzung der modifizierten Kantengewichte nur um einen kostanten Wert von der ursprünglichen Weglänge. Daraus folgt, dass der ermittelte kürzeste *s*-*t*-Weg auch ein kürzester Weg bezüglich des Ausgangsproblems sein muss. □

Input : Gerichteter Graph $G = (V,E)$, nicht-negative c_e für alle $e \in E$. Start- und Zielknoten $s,t \in V$. Untere Schranken $LB_{(v,t)}$ für jeden *v*-*t*-Weg.

Output : Ein kürzester Weg von *s* nach *t* in *G*, falls einer existiert.

Setze $c'_{(v,w)} \leftarrow c_{(v,w)} - LB_{(v,t)} + LB_{(w,t)}$ für alle $e = (v,w) \in E$;
Starte Dijkstra-Algorithmus mit Kantengewichten c'_e;
Transformiere für Lösung *p* die Länge zurück: $c_p \leftarrow c'_p + LB_{(s,t)} - LB_{(t,t)}$;

Algorithmus 2: Der zielgerichtete Dijkstra-Ansatz

Es ist zwar offensichtlich, dass der Beschleunigungseffekt dieses Ansatzes im Wesentlichen von der Qualität der benutzten unteren Schranken abhängt, dennoch benötigt der zielgerichtete Ansatz nie länger als der klassische Dijkstra-Algorithmus. Für viele Verteilungen der Kantenlängen werden sogar im Mittel lineare Laufzeiten erreicht [16].

2.3.2 Der bidirektionale Ansatz

Beim bidirektionalen Ansatz wird nicht versucht, vom Startknoten aus einen Weg zum Zielknoten zu erstellen, vielmehr werden gleichzeitig kürzeste Wege vom Start- und Zielknoten aus berechnet, die, sobald sie sich an einem Knoten treffen, zu einem kürzesten *s*-*t*-Weg zusammengefügt werden können.

Der Dijkstra-Algorithmus wird hierbei gleichzeitig vom Startknoten s und vom Zielknoten t aus gestartet, bei letzterem allerdings mit umgedrehten Kantenorientierungen. Jeder Knoten v hat somit zwei Distanzwerte $dist_s(v)$ und $dist_t(v)$, und es werden diesmal zwei Vorgängerarrays benötigt. Laut Ahuja, Magnanti und Orlin [1] kann die Suche beendet werden, sobald ein Knoten von beiden Richtungen aus markiert worden ist. Der kürzeste Weg besitzt dann eine Länge der Form $dist_s(u) + c_{(u,v)} + dist_t(v)$, wobei u ein von s aus markierter Knoten ist und v ein von t aus markierter Knoten.

Die Ermittlung des kürzesten Weges wird dann über eine Minimumssuche ausgeführt. Gesucht wird der Knoten, der im Schritt vor der erstmaligen beidseitigen Markierung eines Knotens den kleinsten Summenwert der Distanzen hatte, also $\min_u\{dist_s(u) + dist_t(u)\}$ für alle Knoten u mit $dist_s(u) + dist_t(u) < \infty$. Da sich diese Knoten zum Zeitpunkt des Abbruchs in den Datenstrukturen zur Ermittlung des unmarkierten Knotens mit der kleinsten Distanz befinden, ist die Verwaltung eines solchen Minimums nicht allzu aufwendig.

Es lässt sich jedoch noch eine effektivere Abbruchbedingung formulieren.

Satz 2.5. *Sei* $z := \min\{dist_s(u) + c_{(u,v)} + dist_t(v), u$ *von s aus markiert, v von t aus markiert*$\}$ *und* $z_0 := \min\{d_r(v), r \in \{s,t\}, v$ *nicht von r aus markiert)*$\}$.
Gilt $2 \cdot z_0 > z$*, so ist z die kürzeste Weglänge des Problems und der dazugehörige Weg der kürzeste s-t-Weg.*

Beweis. Würde man den Algorithmus laufen lassen, bis keine unmarkierten Knoten mit endlichen Distanzwerten mehr existieren, so wäre z offensichtlich die kürzeste Weglänge für unser Problem. Da z im Laufe des Algorithmus fällt und z_0 steigt, reicht es zu zeigen, dass kein *s*-*t*-Weg mit kleineren Kosten als z existiert, sobald $2 \cdot z_0 > z$ gilt.

Es gelte also unsere Voraussetzung. Wir berücksichtigen alle Kanten $e \in E$ als mögliche Verbindungen von zwei Halbwegen. Sind beide adjazenten Knoten der

betrachteten Kante bereits markiert, so wurde der daraus resultierende s-t-Weg bei der Minimumsbildung von z bereits berücksichtigt. Sind wiederum beide unmarkiert, so kann aufgrund unserer Voraussetzung dieser Weg unsere bisher minimale Weglänge nicht unterbieten, da jeder neu markierte Knoten einen Distanzwert erhalten wird, der mindestens so groß wie z_0 ist.

Für den Fall, dass nur ein adjazenter Knoten markiert ist, nehmen wir o.B.d.A. an, dass es sich dabei für die Kante $e = (u,v)$ um u handelt. Dann ist v zwar noch nicht vom Startknoten aus markiert, es gibt jedoch keinen s-v-Weg, der über e führt und einen besseren Kostenwert hat. Somit entspricht die Länge dieses Weges mindestens den Distanzwerten des von beiden Seiten aus unmarkierten Knotens v und ist aufgrund der Voraussetzung ebenfalls größer als z. □

```
...
while Es existiert unmarkierter Knoten mit minimaler Distanz do
    Markiere unmarkierten Knoten v mit minimaler Distanz dist_min;
    if 2 · dist_min > best_val then return gefundenen Weg;
    if Knoten von s aus markiert then
        foreach Kante e = (v,w) ∈ E do
            if w von t markiert und best_val > dist_s(v) + c_e + dist_t(w) then
                Füge Teilwege zu s-t-Weg über e zusammen;
                best_val ← dist_s(v) + c_e + dist_t(w);
            else if dist_s(v) + c_e < dist_s(w) then
                dist_s(w) ← dist_s(v) + c_e ;
                Vorgänger_s(w) ← v;
    if Knoten von t aus markiert then
        foreach Kante e = (u,v) ∈ E do
            if u von s markiert und best_val > dist_s(u) + c_e + dist_t(v) then
                Füge Teilwege zu s-t-Weg über e zusammen;
                best_val ← dist_s(u) + c_e + dist_t(v);
            else if dist_t(v) + c_e < dist_t(u) then
                dist_t(u) ← dist_t(v) + c_e ;
                Vorgänger_t(u) ← v;
```

Algorithmus 3: Der bidirektionale Dijkstra-Ansatz (Hauptschleife)

Um diese Bedingung nutzen zu können, müssen gewisse Anpassungen am Dijkstra-Algorithmus vorgenommen werden. Führt bei der Betrachtung der ausgehenden Kanten eines gerade erst markierten Knotens eine Kante zu einem Knoten, der bereits von der anderen Seite aus markiert worden ist, so merke man sich diesen Knoten und die bei der Zusammensetzung dieser beiden Pfade entstehende

Weglänge, falls bisher noch kein kürzerer Weg gefunden wurde. Sobald der Distanzwert des unmarkierten Knotens mit der kleinsten Distanz mehr als die Hälfte der bisher ermittelten kleinsten Weglänge beträgt, kann abgebrochen werden, weil diese dann nicht mehr unterboten werden kann. Dies passiert offensichtlich, bevor ein Knoten von beiden Seiten markiert worden ist, womit diese Bedingung zu bevorzugen ist. In der Praxis scheinen sich jedoch die Laufzeiten kaum zu verringern, da die Anzahl der betrachteten Knoten nur unwesentlich sinkt.

Unabhängig davon, welche Abbruchbedingung man letztendlich bevorzugt, entfällt im Gegensatz zum unidirektionalen Dijkstra-Algorithmus die Überprüfung, ob der betrachtete Teilweg gerade den Zielknoten markiert hat, wenn er von s gestartet ist, bzw. ob er den Startknoten markiert hat, wenn er von t kommt. Start- und Zielknoten werden bereits bei der Initialisierung einseitig markiert, so dass wir hierbei nichts anderes als eine Überprüfung durchführen würden, ob ein Knoten von beiden Seiten aus markiert ist. Wie wir gesehen haben, erreicht unser Abbruchkriterium nie diesen Zustand, während das von Ahuja, Magnanti und Orlin vorgestellte Kriterium bei Erreichen dieses Zustandes noch im selben Schleifendurchlauf abbrechen würde.

In den meisten Fällen stellt das bidirektionale Verfahren eine Beschleunigung zum allgemeinen Dijkstra-Algorithmus dar. Geht man von einer kreisförmigen Verbreitung dieses Algorithmus aus, so entstehen beim bidirektionalen Ansatz zwei Kreise mit halb so großem Radius wie der beim unidirektionalen Ansatz benötigte Kreis. Da das Verhältnis der Fläche der kleineren Kreise zum größeren Kreis durch

$$\frac{2 \cdot \pi(\frac{r}{2})^2}{\pi r^2} = \frac{1}{2}$$

beschrieben wird, halbiert sich also bei gewünschtem Verlauf die Anzahl der betrachteten Knoten. Allerdings ist dieses Verfahren nicht zu empfehlen, wenn kein s-t-Weg existiert, denn dann wird ein doppelt so hoher Aufwand wie beim klassischen Ansatz betrieben.

2.3.3 Verbindung des zielgerichteten und des bidirektionalen Ansatzes

Durch eine gleichzeitige Verwendung des zielgerichteten und des bidirektionalen Ansatzes erhofft man sich, die Vorteile beider Methoden miteinander zu vereinen und so eine weitere Beschleunigung zu erzielen. Leider liefert der auf die zielgerichteten Kantengewichte angewendete bidirektionale Ansatz jedoch nicht den kostenminimalen Weg bezüglich des Ausgangsproblems.

Betrachten wir einen Distanzwert, der einen vom Startknoten aus gestarteten s-u-Weg beschreibt, so lässt sich in gleicher Art und Weise wie beim Beweis von Satz

(2.4) das Verhältnis zwischen transformierten Distanzwerten und Distanzwerten hinsichtlich der Originalkosten durch

$$dist'_s(u) = dist_s(u) + LB(u,t) - LB(s,t) \tag{2.3}$$

darstellen. Für den Fall, dass wir vom Zielknoten aus gestartet sind, ergibt sich eine analoge Formel. Ist $e = (u,v)$ die Verbindungskante der in Frage kommenden Halbwege, so lassen sich also die Kosten des sich ergebenden Weges p wie folgt ausdrücken:

$$\begin{aligned} c'_p &= dist'_s(u) + c'_e + dist'_t(v) \\ &= dist_s(u) + c_e + dist_t(v) + LB(s,v) + LB(v,t) - 2 \cdot LB(s,t) \\ &= c_p + LB(s,v) + LB(v,t) - 2 \cdot LB(s,t) \end{aligned}$$

Offensichtlich verfälschen die unteren Schranken zu den Knoten der Verbindungskante das Ergebnis. Wir benötigen deswegen ein Abbruchkriterium, welches sich auf die modifizierten Distanzwerte $dist'$ bezieht, aber bezüglich der Original-Weglänge c_p Optimalität garantiert.

```
...
while Es existiert unmarkierter Knoten mit endlicher Distanz do
    Markiere unmarkierten Knoten v mit minimaler Distanz dist'_min;
    if dist'_min > best_val − LB(s,t) then return gefundenen Weg;
    if Knoten von s aus markiert then
        foreach Kante e = (v,w) ∈ E do
            if w von t markiert und best_val > dist_s(v) + c_e + dist_t(w) then
                Füge Teilwege zu s-t-Weg über e zusammen;
                best_val ← dist_s(v) + c_e + dist_t(w);
            else if dist'_s(v) + c'_e < dist'_s(w) then
                dist'_s(w) ← dist'_s(v) + c'_e ;
                Vorgänger'_s(w) ← v;
    if Knoten von t aus markiert then
        ...
```

Algorithmus 4: Der bidirektional zielgerichtete Dijkstra-Ansatz (Ausschnitt)

Satz 2.6. *Sei best_val die beim bidirektional zielgerichteten Ansatz bisher kürzeste gefundene s-t-Weglänge bezüglich der Originalkosten. Gilt für den kleinsten unmarkierten Distanzwert dist'_min die Ungleichung*

$$dist'\_min \geq best\_val - LB(s,t)$$

so lässt sich kein kostengünstigerer s-t-Weg mehr finden.

Beweis. Sei o.B.d.A. der kleinste Distanzwert aus einem vom Startknoten aus gestarteten Weg zum Knoten u entstanden. Wie wir in (2.3) sehen, entspricht eine solche modifizierte Distanz genau der Länge des s-u-Teilweges plus dem Wert der unteren Schranke für einen u-t-Weg minus der unteren Schranke für einen s-t-Weg, welche für alle transformierten Distanzwerte gleich ist. Gilt unsere Ungleichung, dann kann aus dem betrachteten s-u-Weg kein kostengünstigster s-t-Weg mehr werden. Da wir immer den kleinsten Distanzwert markieren, kann in diesem Fall also *best_val* nicht mehr unterboten werden. □

Dieses Kriterium hat den Nachteil, dass es die Tatsache, dass wir einen bidirektionalen Ansatz verfolgen, überhaupt nicht ausnutzt. Es ist also zu erwarten, dass der Algorithmus, nachdem er den besten Weg gefunden hat, noch eine Weile weiter läuft.

Insgesamt ergibt sich nunmehr folgender Algorithmus: Wir starten den bidirektionalen Dijkstra-Algorithmus auf einem Graphen mit transformierten Kantengewichten. Finden wir einen Weg, so transformieren wir seine Länge zurück, um seine Originalkosten zu ermitteln, und speichern ihn, wenn er kostenminimal bezüglich des Ausgangsproblems ist. Der Algorithmus bricht ab, sobald die in (2.6) formulierte Abbruchbedingung gilt.

Kapitel 3

Ressourcenbeschränkte Wege

Ausgehend von den in Kapitel 2 erworbenen Kenntnissen bei der Lösung des Kürzeste-Wege-Problems zwischen zwei Knoten erweitern wir nun unsere Fragestellung. Besitzt jede Kante noch ein zweites Kantengewicht, so führen wir für dieses Gewicht eine obere Schranke ein; kein zulässiger Weg darf einen höheren Verbrauch dieser Kostenart vorweisen. Wir untersuchen drei Probleme, die zusätzlich diese Nebenbedingung erfüllen müssen: die Suche nach einem kürzesten Weg zwischen zwei Knoten, die Ermittlung von k sogenannten *Pareto-optimalen* Wegen zwischen zwei Punkten, sowie die Suche nach einem Weg, dessen durch das erste Kantengewicht bestimmte Kosten ebenfalls unter einer vorgegebenen oberen Schranke bleiben müssen. Wir erweitern den Dijkstra-Algorithmus derart, dass er den neuen Problemstellungen gerecht wird und lösen somit unsere neuen Fragestellungen. Die in allen drei Fällen auftretende Einführung der oberen Ressourcenschranke erhöht den Aufwand erheblich; so ist das in diesem Kapitel eingeführte ressourcenbeschränkte Kürzeste-Wege-Problem nicht in polynomialer Zeit lösbar, sondern *schwach NP-vollständig*. Aus diesem Grund machen wir von diversen Beschleunigungsmethoden Gebrauch. Schließlich werden die Methoden vorgestellt, die zur Lösung des ressourcenbeschränkten Kürzeste-Wege-Problems im *CMCF*-Projekt und dem *CNOP*-Paket verwendet werden.

3.1 Problemstellung

Wie im vorangegangenen Kapitel betrachten wir ein Verkehrsnetzmodell, in dem die Straßen in einem Graphen als Kanten und die Kreuzungen als Knoten dargestellt werden. Jede dieser Kanten hat nun zwei verschiedene Kostenwerte, zum Beispiel Zeitverbrauch und Weglänge.

Sei also $G = (V,E)$ ein gerichteter Graph und für jede Kante $e \in E$ gebe es ein nicht-negatives Kostenpaar (c_e, r_e). Wir bezeichnen im weiteren Verlauf der Arbeit

die c_e als Kosten und r_e als Ressource. Seien nun ein Startknoten s, ein Zielknoten t sowie eine obere Ressourcenschranke UBR (*Upper_Bound_Resource*) gegeben.

Wir werden im Folgenden unsere drei Problemstellungen anhand dieser Ausgangsdaten definieren.

3.1.1 Das ressourcenbeschränkte Kürzeste-Wege-Problem

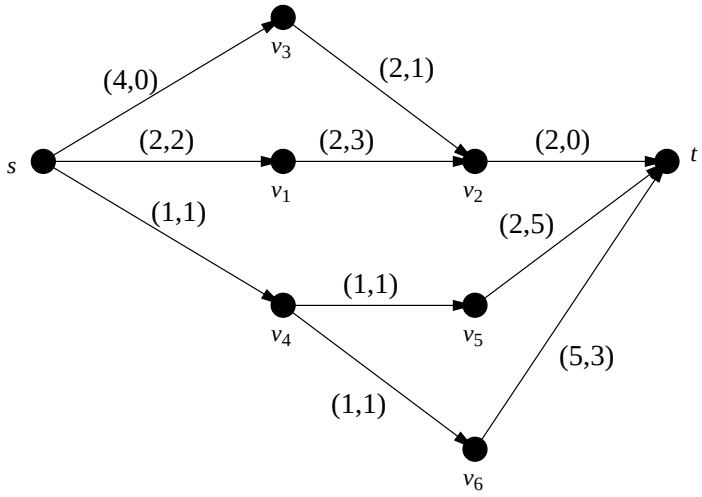

Abbildung 3.1: Beispielgraph

Schauen wir uns zunächst den Beispielgraphen aus Abbildung (3.5) an. Die Wege $p_1 = (s, v_4, v_5, t)$, $p_2 = (s, v_1, v_2, t)$, $p_3 = (s, v_3, v_2, t)$ und $p_4 = (s, v_4, v_6, t)$ führen allesamt von s nach t. Das klassische Kürzeste-Wege-Problem würde p_1 als Optimallösung ermitteln, da dieser Weg Kosten von 4 aufweist. Führen wir nun als obere Schranke für das zweite Kantengewicht den Wert 6 ein, so ist p_1 nicht mehr zulässig, da sein Ressourcenverbrauch zu hoch ist. Der kostenminimale zulässige Weg ist in diesem Fall p_2. Er ist somit der ressourcenbeschränkte kürzeste Weg auf unserem Graphen.

Formal lässt sich dieses Problem wie folgt ausdrücken:

$$min \sum_{e \in p} c_e,$$

$$\text{so dass} \quad \sum_{e \in p} r_e \leq UBR \text{ und}$$
$$p \text{ ein Weg von } s \text{ nach } t.$$

Wir werden im weiteren Verlauf diese Fragestellung mit dem Kürzel *CSP* (*Constrained Shortest Path*-Problem) bezeichnen.

3.1.2 Pareto-optimale ressourcenbeschränkte Wege

Existieren für einen Graphen zwei unterschiedliche Kantengewichte, so ist vor der Lösung des oben genannten Problems festzusetzen, welches Gewicht als Kosten fungiert, also minimiert wird, und welches als Ressource nur unter einer oberen Schranke bleiben muss. In vielen Fällen dürfte jedoch ein geringfügig teuerer Weg mit viel niedrigerem Ressourcenverbrauch bei der Anwendung auf reale Fragestellungen erfolgsversprechender sein. Wir haben bereits in Kapitel 1 das Beispiel eines Lastwagenfahrers erwähnt, der so schnell wie möglich zu seinem Lieferort fahren soll, dabei aber nur einen gewissen Geldbetrag für Benzin und Maut ausgeben darf. Definieren wir die verbrachte Zeit auf einer Straße als Kosten der zugehörigen Kante und den für die Nutzung dieser Straße zu entrichtenden Geldbetrag als Ressource, so können wir durch Lösung von *CSP* auf dem so erstellten Graphen eine optimale Fahrtroute ermitteln. In der Praxis wären die meisten Arbeitgeber jedoch wahrscheinlich zufriedener, wenn ihr Fahrer wenige Minuten länger benötigte, aber dafür wesentlich weniger Geld ausgäbe.

Ein Algorithmus, der mehrere alternative Routenvorschläge zurückliefert, kann also für bestimmte Problemstellungen aussagekräftigere Resultate liefern als einer, der nur den kostenminimalen Weg ermittelt. Zu diesem Zweck führen wir den Begriff der *Pareto-Optimalität* ein.

Definition 3.1. *In einem gerichteten Graphen mit Kostenpaar* (c_e, r_e) *für jede Kante* $e \in E$ *gilt ein v-w-Weg p als von einem v-w-Weg* p' dominiert, *wenn die zugehörigen Kostenpaare* (c_p, r_p) *und* $(c_{p'}, r_{p'})$ *ungleich sind und folgende Eigenschaft gilt:*

$$c_p \geq c_{p'} \text{ und } r_p \geq r_{p'}$$

Ein v-w-Weg p heißt Pareto-optimal, *falls es keinen v-w-Weg* p' *gibt, der ihn dominiert.*

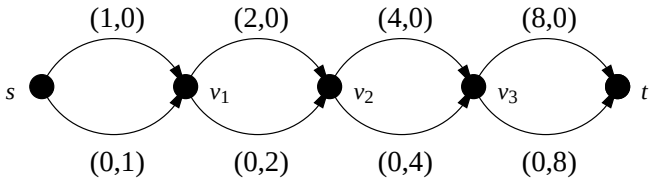

Abbildung 3.2: Beispielgraph für Pareto-Optimalität

Es stellt sich die Frage, wie viele Pareto-optimale Wege für einen Graphen existieren können. Wie wir am Beispielgraphen (3.2) sehen, kann ihre Anzahl exponentiell zur Anzahl der Knoten werden, denn dort sind alle 2^4 *s-t*-Wege Pareto-optimal. Da wir einerseits die Laufzeiten unserer Algorithmen niedrig halten möchten und andererseits weiterhin nach Wegen suchen, welche nicht zu viele Ressourcen verbrauchen, schränken wir die Anzahl der zu findenden Wege ein. Aus der Menge

der zulässigen Wege suchen wir die k kostenminimalen heraus, die von keinem anderen zulässigen Weg dominiert werden.

Betrachten wir noch einmal den Beispielgraphen aus dem letzten Abschnitt. Da p_1 unzulässig ist, bleiben drei für uns interessante Wege zur Verfügung. Der Weg p_4 wird jedoch von p_2 dominiert, da er bei gleichem Ressourcenverbrauch höhere Kosten verursacht. Somit bleiben uns zwei Pareto-optimale Wege, p_2 und p_3.

Formulieren lässt sich die soeben erläuterte Fragestellung wie folgt:

Finde k Pareto-optimale s-t-Wege p^j mit Gewichten (c_{p^j}, r_{p^j}) für $j = 1, \ldots, k$,

$$\begin{aligned} \text{so dass} \qquad & r_{p^j} \leq UBR \text{ für } j = 1, \ldots, k \text{ und} \\ & \text{es existiert kein Weg } p' \text{ mit} \\ & c_{p'} < \max\{c_{p^j}, j \in \{1, \ldots, k\}\}, r_{p'} \leq UBR. \end{aligned}$$

Dieses Problem wird in späteren Abschnitten als *POCP*(*Pareto-Optimal Constrained Paths*-Problem) auftreten.

3.1.3 Das doppelt beschränkte Problem

Wie wir in Kapitel 1 gesehen haben, tritt das ressourcenbeschränkte Kürzeste-Wege-Problem auch häufig als Unterproblem in größeren Projekten auf. Es wird dabei wiederholt aufgerufen und ist nicht selten für den Großteil der verbrauchten Gesamtlaufzeit verantwortlich. Da es weniger aufwändig ist, einen Weg zu finden, der auch bezüglich der Kosten nur unter einer vorgegebenen Schranke liegt, ist es denkbar, dass gewisse Programme darauf verzichten, den kostenminimalen Weg zu finden. Sie begnügen sich mit einem Weg, der hinsichtlich beider Gewichte zulässig ist, und senken dadurch den Gesamtaufwand.

Kommen wir noch einmal auf unseren Beispielgraphen zurück. Wir nennen unsere obere Kostenschranke UBC ($Upper\_Bound\_Cost$) und weisen ihr den Wert 7 zu. Somit sind p_1 und p_3 unzulässig, während p_2 und p_4 mögliche Lösungen des Problems sind. Welcher Weg schließlich ermittelt wird, hängt von der Vorgehensweise des benutzten Algorithmus ab.

Letztlich entspricht unser Problem folgender Fragestellung:

Finde einen Weg von s nach t,

$$\begin{aligned} \text{so dass} \quad & \textstyle\sum_{e \in p} c_e \leq UBC \text{ und} \\ & \textstyle\sum_{e \in p} r_e \leq UBR. \end{aligned}$$

Wir weisen dieser Problemstellung das Kürzel *DB* (*Double Bounded*-Problem) zu.

3.1.4 Beziehung zwischen den drei betrachteten Problemen

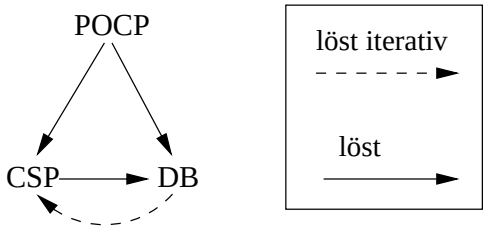

Abbildung 3.3: Überblick über die Beziehungen zwischen unseren drei Problemen

Unsere soeben vorgestellten Probleme haben einige Gemeinsamkeiten. Es wird jeweils ein *s*-*t*-Weg gesucht, der einen vorgegebenen Ressourcenverbrauch nicht überschreiten darf. Bei näherer Betrachtung fällt auf, dass aus der Lösung *eines* Problems auch die Lösung mindestens einer anderen Problemstellung abgeleitet werden kann. So liefert ein Algorithmus, der die k kostenminimalen Pareto-optimalen ressourcenbeschränkten Wege findet, automatisch den ressourcenbeschränkten kürzesten Weg zurück. Ein solcher Weg löst wiederum das doppelt beschränkte Problem; da er ebenfalls die Ressourcenschranke unterbietet, sind die Kosten des ermittelten Weges mit der oberen Kostenschranke des verwandten Problems zu vergleichen. Wird die Schranke unterboten, so ist der Weg zulässig und eine Lösung gefunden; ansonsten gibt es keinen zulässigen Weg.

Letztendlich lässt sich auch durch wiederholte Anwendung eines Algorithmus für das doppelt beschränkte Problem ein ressourcenbeschränkter Weg finden. Nach jedem Durchlauf wird einfach der Kostenwert des gefundenen Weges zur neuen oberen Kostenschranke. Sobald keine Lösung mehr gefunden wird, ist der zuletzt gefundene Weg der ressourcenbeschränkte kürzeste *s*-*t*-Weg.

Allerdings ist eine solche iterative Herangehensweise sehr zeitaufwändig. Je mehr sich die obere Kostenschranke dem Optimalwert nähert, desto länger dauert ein einzelner Durchlauf. Die letzten Iterationen dürften fast genauso aufwändig wie die direkte Ermittlung des Minimums sein, da das Problem immer schwerer lösbar wird. Aus diesem Grund dürfte die Laufzeit dieses Verfahrens den Aufwand zur direkten Lösung des ressourcenbeschränkten Kürzeste-Wege-Problems bei weitem überschreiten, weswegen wir sie in dieser Arbeit nicht implementiert haben.

3.2 Grundlegende Eigenschaften des ressourcenbeschränkten Kürzeste-Wege-Problems

3.2.1 Komplexität

Für das ressourcenbeschränkte Kürzeste-Wege-Problem wurde bisher kein polynomialer Algorithmus gefunden. In der Tat handelt es sich hierbei um ein *schwach NP-vollständiges* Problem. Garey und Johnson [7] führen es zum Beispiel als Problem *ND30* auf und beweisen seine Komplexität mit einer Transformation des Partition-Problems. Handler und Zang [8] beweisen diese Eigenschaft auch für den Fall mit nicht-negativen Kantengewichten. Sie tun dies, indem sie das *Knapsack*-Problem auf das Kürzeste-Wege-Problem mit nicht-negativen Kantengewichten reduzieren.

Satz 3.2. *Das ressourcenbeschränkte Kürzeste-Wege-Problem mit nicht-negativen Kantengewichten ist* schwach NP-vollständig.

Beweis. Wir beschreiben zunächst das Knapsack-Problem. Seien n verschiedene Elemente gegeben, die jeweils einen Wert v_i und ein Gewicht w_i haben. Ziel des Knapsack-Problems ist es, eine Anzahl an Elementen auszuwählen, die zusammen eine vorgegebene Gewichtsgrenze UBR nicht übersteigen, aber einen höchstmöglichen Gesamtwert erzielen. Dies lässt sich wie folgt formulieren:

$$\max \sum_{i=1}^{n} v_i k_i,$$

$$\text{so dass } \sum_{i=1}^{n} w_i k_i \leq UBR,$$

$$k_i \in \{0,1\},\ i = 1,\ldots,n,$$

$$v_i \geq 0,\ w_i \geq 0,\ i = 1,\ldots,n,$$

$$UBR \geq 0.$$

Wir erstellen nun einen Graphen G mit $n+1$ Knoten $v_1,\ldots,v_{n+1}$ und fügen für jedes Knotenpaar (v_i, v_{i+1}) zwei Kanten $e^1_{(i,i+1)}$ und $e^2_{(i,i+1)}$ hinzu.

Sei $M := \max\{v_i : i = 1,\ldots,n+1\}$. Setzen wir $c_{e^1_{(i,i+1)}} = M - v_i$, $r_{e^1_{(i,i+1)}} = w_i$ für die erste und $c_{e^2_{(i,i+1)}} = M$, $r_{e^2_{(i,i+1)}} = 0$ für die zweite Kante, so kann man auf G ein ressourcenbeschränktes Kürzeste-Wege-Problem lösen.

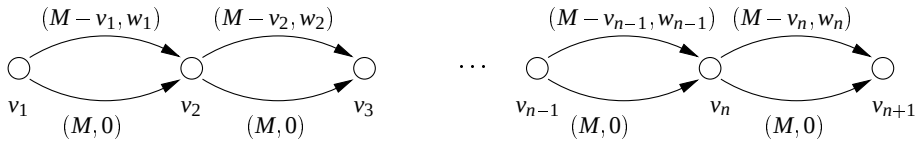

Abbildung 3.4: Der aus dem Knapsack-Problem entstandene Graph

Der ermittelte Weg wird offensichtlich so viele $e^1_{(i,i+1)}$ Kanten wie möglich enthalten. Müssen zum Zwecke der Zulässigkeit $e^2_{(i,i+1)}$ Kanten verwendet werden, so wird man dies bei den Knotenpaaren (v_i, v_{i+1}) tun, für die der Wert $M - v_i$ am größten ist.

Betrachten wir nun die Optimallösung des Kürzeste-Wege-Problems mit Nebenbedingung. Die Kanten $e^1_{(i,i+1)}$ werden bei den Knotenpaaren eingesetzt, für die v_i die größten Werte erreichen. Setzen wir für solche Knotenpaare (v_i, v_{i+1}) im Knapsack-Problem $x_i = 1$ und für die restlichen Paare $x_i = 0$, so haben wir auch dort eine optimale Lösung gefunden. Da das Knapsack-Problem *NP-vollständig* ist, ist auch das ressourcenbeschränkte Kürzeste-Wege-Problem *schwach NP-vollständig*. □

Es sei an dieser Stelle erwähnt, dass diverse Approximationsschemata zur Berechnung Pareto-optimaler Wege existieren, die teilweise sogar polynomial in der Größe des Graphen und dem Inversen $\frac{1}{\varepsilon}$ der Genauigkeit sind. Ihre praktische Relevanz ist bisher jedoch sehr gering. Einen Überblick über die verschiedenen Approximationsansätze liefert zum Beispiel Möhring [14].

3.2.2 Bisherige Ansätze

Die ersten nennenswerten Arbeiten zum ressourcenbeschränkten Kürzeste-Wege-Problem entstanden Mitte der 60er Jahre. Unabhängig voneinander entwickelten sowohl Witzgall und Goldmann [19] als auch Joksch [12] eine Lösungsidee, die auf dynamischer Programmierung aufbaut.

Bezeichnen wir einen Weg, dessen Ressourcenverbrauch α nicht übersteigt, als α-Pfad, so wird beim *CSP*-Problem nach dem kostenminimalen *UBR*-Pfad vom Startknoten 1 zum Zielknoten n gesucht. Seien nun $c_v(\alpha)$ die Kosten des kürzesten α-Weges von 1 nach v. Dann gilt folgende Rekursion:

$$c_v(\alpha) = \min\{c_v(\alpha-1), \min_{(u,v)\in E, r_{(u,v)}\leq UBR}\{c_u(UBR - r_{(u,v)}) + c_{(u,v)}\}\}$$

Nachdem wir $c_1(\alpha) = 0$ für alle $0 \leq \alpha \leq UBR$ und $c_v(0) = \infty$ für $v = 2, \cdots, n$ gesetzt haben, können wir unsere Optimallösung $c_n(UBR)$ rekursiv mit einem Aufwand von $O(m \cdot UBR)$ bestimmen. Die Laufzeit ist also pseudopolynomial.

Aufbauend auf diese Idee entstanden in der Folgezeit eine Vielzahl an sogenannten Labeling-Ansätzen, zu denen auch unsere für diese Arbeit erstellten Algorithmen zu zählen sind. Da wir im nächsten Abschnitt unsere Ansätze genau erklären, gehen wir an dieser Stelle nicht auf den typischen Verlauf dieser Verfahren ein. Die wichtigste Verbesserung gegenüber dem erstgenannten Ansatz besteht allerdings darin, dass dominierte Teilwege nicht weiter betrachtet werden. Somit ist im Normalfall die Lösung viel schneller gefunden. Obwohl etliche Varianten existieren, die Beschleunigungseffekte für die meisten Problemstellungen erzielen, z.B. von Desrochers und Soumis [5] oder auch Stroetmann [18], entspricht die Laufzeit im ungünstigsten Fall dennoch der für die dynamische Programmierungs-Rekursion gefundenen Schranke, nämlich genau dann, wenn keine dominierten Wege existieren.

Einen völlig anderen Lösungsweg verfolgt die 2-*Phasen-Methode*. Wie wir in Abschnitt 3.4.2 bei der Erläuterung des im *CNOP*-Paket verwendeten Algorithmus detailliert darstellen werden, besteht der erste Schritt darin, anhand einer Relaxation des Problems möglichst gute untere und obere Schranken für die Kosten der Optimallösung zu finden. Sollten die Schranken voneinander abweichen, so ist in einem zweiten Schritt aus den bezüglich der ermittelten Kostenschranken zulässigen Wegen die Lösung des Problems herauszusuchen. Es gibt mehrere Möglichkeiten, dies zu bewerkstelligen. Die bekanntesten Arbeiten zu diesem Thema sind die Lösungsvorschläge von Handler und Zang [8], welche den zweiten Schritt mit einem *k Kürzeste-Wege*-Algorithmus ausführen, sowie die von Beasley und Christofides [2], die hierfür eine *Branch-and-Bound*-Methode einsetzen.

Ein sehr simpler Ansatz besteht darin, zunächst die Ressourcenschranke zu ignorieren und mittels des *k Kürzeste-Wege*-Problems die kostenminimalen Wege in nicht-absteigender Reihenfolge hinsichtlich der Kosten zu generieren. Jeder neu ermittelte Weg wird daraufhin bezüglich seines Ressourcenverbrauchs überprüft. Der erste Weg, der die obere Ressourcenschranke nicht überschreitet, ist optimal. Diese Idee wurde zuerst von Yen [20] dargestellt und in den folgenden Jahren weiter verbessert, insbesondere von Jimenez und Marzal [11], deren Algorithmus eine Laufzeit von $O(m + k \cdot n \cdot \log(m/n))$ benötigt, wobei n die Anzahl der Knoten und m die Anzahl der Kanten ist. Wie wir bereits bei der Einführung der Pareto-Optimalität gesehen haben, kann die Anzahl der zu betrachtenden s-t-Wege exponentiell zur Anzahl der Knoten sein. Aus diesem Grunde ist es nicht verwunderlich, dass dieses Verfahren bei experimentellen Vergleichen (z.B. den von Skicism und Golden [17] durchgeführten Tests) den anderen Ansätzen unterlegen ist.

3.3 Der erweiterte Dijkstra-Algorithmus

Es werden nun mehrere auf dem Dijkstra-Algorithmus basierende Lösungsmethoden für die bereits erwähnten Problemstellungen vorgestellt.

3.3.1 Die Grundversion

Trotz der zusätzlichen Nebenbedingung ist die grundsätzliche Idee des Dijkstra-Algorithmus auch auf das *CSP*-Problem anwendbar, denn schließlich wird weiterhin nach einem kürzesten *s*-*t*-Weg bezüglich einer Kostenart gefragt. Offensichtlich ist ein mit dem klassischen Dijkstra-Ansatz ermittelter Weg jedoch nicht notwendigerweise zulässig, da er unter Umständen die Nebenbedingung verletzt. Somit reicht es nicht mehr aus, sich für jeden Knoten v den bisher errechneten kürzesten *s*-*v*-Weg zu merken. Auch Wege mit schlechteren Kosten, aber günstigerem Ressourcenverbrauch sind nun von Interesse, denn sie haben eine größere Chance, die Nebenbedingung zu erfüllen.

Es stellt sich nun die Frage, welche Wege nicht weiter berücksichtigt werden müssen. Es liegt auf der Hand, dass dominierte Teilwege nicht zu einem optimalen Gesamtweg führen können, da der sie dominierende Weg zu einer Verbesserung des Optimalwerts führen würde. Somit bleiben alle Pareto-optimalen Wege übrig, welche solange von Interesse sind, bis sie die obere Ressourcenschranke überschreiten.

Input : Gerichteter Graph $G = (V,E)$, nicht-negative (c_e, r_e) für alle $e \in E$. Start- und Zielknoten $s,t \in V$, obere Ressourcenschranke UBR. Eine gewünschte Anzahl k an Pareto-optimalen Wegen.
Output : Bis zu k ressourcenbeschränkte kürzeste Wege von s nach t.

foreach $v \in V \backslash \{s\}$ **do** Füge $((+\infty,+\infty),\emptyset)$ in $list(v)$ ein;
Füge $((0,0),\emptyset)$ in $list(s)$ ein;
while *Es existiert ein unmarkierter Listeneintrag mit endlicher Distanz.* **do**
  Markiere unmarkierten Listeneintrag mit minimaler Distanz $dist_{min}$;
  Sei v der der Liste zugehörige Knoten;
  if $v = t$ **then**
    Speichere $dist_{min}$ und den zugehörigen *s*-*t*-Weg;
    if *k Wege gefunden* **then return** *alle gespeicherten Wege*;
    continue;
  foreach *Kante* $e = (v,w) \in E$ **do**
    $new\_value := (c_{new\_value}, r_{new\_value}) \leftarrow dist_{min} + (c_e, r_e)$
    if $r_{new\_value} \leq UBR$ **then**
      Füge new_value zu $list(w)$ hinzu;
      Streiche nicht Pareto-optimale Wege aus $list(w)$;
return *alle bisher gespeicherten s-t-Wege.*;

Algorithmus 5: Der erweiterte Dijkstra-Algorithmus

Aus dieser Feststellung folgt die für den Algorithmus weitestreichende Änderung: Reichte es früher aus, sich für jeden Knoten den bisher berechneten kürzesten

Weg mitsamt seiner Länge zu merken, so sind nun für jeden Knoten alle Pareto-optimalen Wege zu betrachten. Dies führt zu einer erheblichen Steigerung des Aufwands, denn die Anzahl der Pareto-optimalen Wege kann exponentiell zur Anzahl der Knoten werden, wie wir bereits in Abschnitt 3.1.2 erwähnt haben.

Somit benötigt der Algorithmus nun für jeden Knoten v eine Liste $list(dist(v), Vorgänger(v))$ zur Abspeicherung der Kostenpaare $dist(v)$ sowie des jeweils zuletzt durchlaufenen Knotens $Vorgänger(v)$ der in Frage kommenden Pareto-optimalen Wege. Diese Listen sollten lexikographisch geordnet sein, damit wir schnell Zugriff auf den vielversprechendsten Weg haben. Es werden nun auch nicht mehr die Knoten selbst, sondern die Listeneinträge markiert.

Die Initialisierungsphase verläuft analog zum einfachen Dijkstra-Algorithmus: Der Liste des Startknotens wird das $(0,0)$-Paar zugefügt und alle anderen Listen bekommen den Eintrag (∞, ∞).

Es folgt nun die Hauptschleife. Wir suchen jedes Mal den lexikographisch kleinsten unmarkierten Wert $dist_{min}$ aller Listeneinträge und markieren ihn. Für den Knoten v, in dessen Liste der Eintrag gefunden wurde, werden alle ausgehenden Kanten $e = (v,w)$ betrachtet. Das Kostenpaar $dist_{min} + (c_e, r_e)$ wird mit den Listeneinträgen des Knotens w verglichen und dort eingefügt, falls es von keinem der vorhandenen Werte dominiert wird. Dominiert es wiederum selber Listeneinträge, so werden diese Einträge gestrichen.

Dies wird solange durchgeführt, bis ein Listeneintrag des Zielknotens markiert ist. Dieser liefert somit den kostenminimalen zulässigen Weg. Lässt man den Algorithmus weiterlaufen, so wird jedes Mal, wenn ein Element der Liste des Zielknotens markiert wird, der nächstgünstigste zulässige Weg gefunden. Dies bedeutet, dass man mit diesem Algorithmus auch eine vorgegebene Anzahl an Pareto-optimalen Wegen ermitteln kann. Insgesamt löst dieser Algorithmus also alle drei vorgegebenen Problemstellungen.

3.3.2 Die zielgerichtete Version

Genau wie beim normalen Dijkstra-Algorithmus ist auch im erweiterten Fall eine Transformation der Kantengewichte zwecks zielgerichteter Suche problemlos möglich, wie wir in Satz (3.3) sehen werden. Wir ermitteln zunächst untere Schranken $LBC_{(v,t)}$ und $LBR_{(v,t)}$ für Kürzeste-Kosten-Wege und Günstigste-Ressourcen-Wege vom Zielknoten t zu jedem Knoten v. Da unser Problem nun *NP-vollständig* ist, müssen wir nicht zwingend auf bereits vorgegebene Schranken zurückgreifen. Eine Methode, welche genauere Schranken in polynomialer Laufzeit berechnet, sorgt für schneller ermittelte Lösungen, da der damit verbundene zusätzliche Aufwand durch den erreichten Nutzen bei weitem aufgewogen wird.

Wie wir in Korollar (2.3) gesehen haben, berechnet der klassische Dijkstra-Algorithmus kürzeste Wege von *einem* Start- zu mehreren Zielknoten in poly-

nomialer Zeit. Da die ohne Nebenbedingung ermittelten kürzesten Wege nicht größer als die ressourcenbeschränkten Wege sind und die Konsistenzbedingung (2.2) erfüllen, sind sie als untere Schranken für unsere Zwecke einsetzbar. Wir berechnen zunächst alle kürzesten Wege von t aus bezüglich der Ressourcen r_e. Hierbei kann abgebrochen werden, sobald ein ermittelter kürzester Weg größer als die vorgegebene obere Ressourcenschranke ist. Wir ermitteln für den kürzesten s-t-Weg p_r zusätzlich den Kostenwert $c_{LBR_{(s,t)}}$ und starten daraufhin noch einmal den Dijkstra-Algorithmus, aber diesmal bezüglich der Kosten c_e und mit $c_{LBR_{(s,t)}}$ als oberer Schranke, da ein Weg mit höheren Kosten von p_r dominiert werden würde. Auch hier brechen wir ab, sobald die obere Schranke erreicht wird. Alle noch nicht berechneten Schranken werden daraufhin mit $+\infty$ initialisiert, denn sie können nicht zu einem zulässigen Weg für unser Problem führen.

Die Transformation der Ressourcenwerte ist sicherlich nicht zwingend, da ja nicht bezüglich dieser Kostenart minimiert wird. Von Interesse ist sie nur dann, wenn zwei zu vergleichende Wege die gleichen Kosten haben. Da wir immer das Kostenpaar (c_e, l_e) betrachten und lexikographisch sortieren, entscheidet in dem Fall der Ressourcenverbrauch darüber, welcher Weg zu bevorzugen ist.

Analog zum zielgerichteten Ansatz für den klassischen Dijkstra-Algorithmus definieren wir nun die transformierten Kantengewichte wie folgt:

$$(c'_{(v,w)}, r'_{(v,w)}) := (c_{(v,w)} - LBC_{(v,t)} + LBC_{(w,t)}, r_{(v,w)} - LBR_{(v,t)} + LBR_{(w,t)}) \quad (3.1)$$

Für die obere Schranke müssen ebenfalls Anpassungen vorgenommen werden, damit sie auf die neuen Gewichte anwendbar ist:

$$UBR' := UBR - LBR_{(s,t)} \quad (3.2)$$

Starten wir nun den erweiterten Dijkstra-Algorithmus auf unserem Problem mit transformierten Kantengewichten und neuer oberen Schranke, so sind die ermittelten Lösungen genau die kürzesten Wege bezüglich der Originalgewichte und die oberen Schranken werden unterboten.

Satz 3.3. *Der erweiterte Dijkstra-Algorithmus liefert sowohl auf dem Ausgangsproblem als auch den gemäß* (3.1) *und* (3.2) *transformierten Werten die gleichen Wege für unsere betrachteten Probleme zurück, wenn dabei kürzeste Wege als untere Schranken verwendet werden.*

Beweis. Da sowohl unsere unteren Kosten- als auch unsere unteren Ressourcenschranken wie bereits erwähnt die Kosistenzbedingung (2.2) erfüllen, lässt sich analog zum Beweis von Satz (2.4) zeigen, dass die auf den transformierten Werten ermittelten s-t-Wege in beiden Komponenten nur um einen konstanten Term von ihren jeweiligen Kosten und Ressourcen bezüglich der Ausgangsdaten abweichen.

Es bleibt zu beweisen, dass die Transformation der oberen Ressourcenschranke zu einem korrekten Ergebnis führt. Da durch die von uns benutzten kürzesten Wege

die untere Schranke für einen Weg vom Ziel- zum Zielknoten immer 0 ist, entspricht unsere transformierte Schranke der Originalschranke plus genau dem Term, der auch einem *s*-*t*-Weg durch die Transformation aufaddiert wird. Somit ist ein bezüglich des Originalproblems unzulässiger Weg auch auf den transformierten Werten unzulässig. Da außerdem alle Ressourcen der Kanten nicht-negativ sind, sind sowohl ein zulässiger *s*-*t*-Weg als auch alle seine Teilwege hinsichtlich der transformierten Werte ebenfalls zulässig, woraus die Korrektheit des Algorithmus folgt. □

Da der zielgerichtete Ansatz das gleiche Verhalten wie der erweiterte Dijkstra-Algorithmus ohne Beschleunigung aufweist, löst er also ebenso unsere drei Ausgangsprobleme.

Input : Gerichteter Graph $G = (V,E)$, nicht-negative (c_e, r_e) für alle $e \in E$. Start- und Zielknoten $s,t \in V$, obere Ressourcenschranke UBR. Eine gewünschte Anzahl k an Pareto-optimalen Wegen.
Output : Bis zu k ressourcenbeschränkte kürzeste Wege von s nach t.

Ermittle untere Schranken $LBR_{(v,t)}$ und $LBC_{(v,t)}$;
Setze $(c'_{(v,w)}, r'_{(v,w)}) \leftarrow (c_{(v,w)} - LBC_{(v,t)} + LBC_{(w,t)}, r_{(v,w)} - LBR_{(v,t)} + LBR_{(w,t)})$ für alle Kanten $e = (v,w)$ von G;
Setze $UBR' \leftarrow UBR - LBR_{(s,t)}$;
Starte erweiterten Dijkstra-Algorithmus mit (c'_e, r'_e) und UBR';
Transformiere für alle ermittelten Wege p die Kantengewichte zurück: $(c_p, r_p) \leftarrow (c'_p + LBC_{(s,t)}, r'_p + LBR_{(s,t)})$;

Algorithmus 6: Der zielgerichtete erweiterte Dijkstra-Algorithmus

Genau genommen müssen wir bei dieser Variante gar nicht warten, bis der Zielknoten erreicht wird, um den jeweils nächsten kostenminimalen ressourcenbeschränkten Weg zu finden. Wir könnten nämlich die Tatsache ausnutzen, dass die unmarkierten Listeneinträge hinsichtlich ihrer transformierten Kostenwerte lexikographisch geordnet sind. Diese entsprechen wie gesagt einem Pareto-optimalen Weg zu dem betrachteten Knoten plus für beide Komponenten einer unteren Schranke für den Restweg zum Zielknoten abzüglich der unteren Schranke für einen *s*-*t*-Weg, wobei letztere für jeden umarkierten Listeneintrag gleich ist. Da unsere unteren Schranken (nicht notwendigerweise bezüglich beider Kantengewichte zulässige) Teilwege beschreiben, bedeutet dies, dass der jeweils lexikograpisch kleinste unmarkierte Listeneintrag zu einem *s*-*t*-Weg komplettiert werden könnte, den bezüglich der Kosten keiner der aus den restlichen Einträgen hervorgehenden *s*-*t*-Wege unterbieten kann. Somit haben wir unseren ressourcenbeschränkten kürzesten Weg gefunden, soweit wir nachweisen können, dass eine derartige Erweiterung eines lexikographisch kleinsten Listeneintrags zu einem zulässigen *s*-*t*-Weg führt.

Bei der Ermittlung der unteren Kostenschranken ist es ohne weiteren Aufwand möglich, sowohl den Ressourcenverbrauch $r_{LBC_{(v,t)}}$ als auch die jeweiligen Vorgängerknoten eines kostenminimalen v-t-Weges zu speichern. In der Hauptschleife fügen wir daraufhin eine zusätzliche Abfrage bei der Betrachtung des lexikograpisch kleinsten unmarkierten Listeneintrags am Knoten v ein. Wir wollen wissen, ob die Komplettierung durch den kostenminimalen v-t-Weg den Ressourcenverbrauch über die obere Schranke ansteigen lässt. Da der transformierte Ressourcenwert des Listeneintrags sich jedoch auf den ressourcengünstigsten v-t-Weg bezieht, müssen wir ihn so modifizieren, dass er den Verbrauch des kostenminimalen Weges angibt. Der Weg ist deswegen genau dann zulässig, wenn folgende Eigenschaft gilt:

$$r'_{dist\_min'} - LBR(v,t) + r_{LBC_{(v,t)}} \leq UBR'$$

Ist dies der Fall, so ist der zugehörige s-t-Weg kostenminimal.

Dieses Abbruchkriterium findet den Weg normalerweise, bevor der Zielknoten erreicht wird, allerdings war bei durchgeführten Tests der erreichte Nutzen bei Weitem geringer als erhofft und eher marginal. Außerdem besitzt dieses Verfahren den Nachteil, dass es zwar den kostenminimalen, aber nicht notwendigerweise einen Pareto-optimalen Weg liefert, denn es ist durchaus möglich, dass ein weiterer Listeneintrag zu einem Weg mit gleichen Kosten, aber weniger Ressourcenverbrauch führt. Aus diesem Grund haben wir uns entschlossen, dieses Kriterium in unserer Variante nicht zu berücksichtigen und stattdessen erst dann abzubrechen, wenn ein Eintrag des Zielknotens markiert wird.

3.3.3 Die bidirektionale Version

Der bidirektionale Ansatz lässt sich ebenfalls auf die erweiterte Fragestellung anwenden. Es ist allerdings zu beachten, dass von nun an nicht mehr die Knoten, sondern die Listeneinträge markiert werden. Dies führt zu gewissen Veränderungen bei der Zusammenfügung der Teilwege und dem Abbruchkriterium.

Wir beschreiben o.B.d.A. den Fall, dass wir gerade einen Listeneintrag vom Startknoten aus markiert haben. Betrachten wir nun eine ausgehende Kante $e = (v,w)$, so suchen wir nach einem vom Zielknoten aus markierten und bezüglich des Kostenpaares lexikograpisch kleinsten Listeneintrag in w. Existiert dieser, so beschreibt er den kostengünstigsten zulässigen w-t-Weg. Damit haben wir kostenminimale s-v- und w-t-Wege; ist der zusammengesetzte Weg zulässig, so ist er der beste Weg, der über e führt. Ansonsten suchen wir in der aufsteigend lexikographisch geordneten Liste von w weiter. Falls wir keinen zulässigen s-t-Weg ermitteln können, läuft der Algorithmus weiter und startet den nächsten Hauptschleifendurchlauf.

Wird hierbei jedoch ein Weg gefunden, so brechen wir in einer ersten Variante dieses Ansatzes an dieser Stelle ab und liefern eine Lösung für das *DB*-Problem, welche im Allgemeinen nur wenig schlechter als der ressourcenbeschränkte kürzeste Weg, allerdings meistens nicht Pareto-optimal ist.

```
Markiere unmarkierten Listeneintrag mit minimaler Distanz dist_min;
if 2 · dist_min >_lex best_val then return gefundenen Weg;
Sei v der der Liste zugehörige Knoten;
if Eintrag von s aus markiert then
    foreach Kante e = (v,w) ∈ E do
        new_val := (c_new_val, r_new_val) ← dist_min + (c_e, r_e)
        foreach Eintrag in w von t markiert do
            Füge Teilwege zu s-t-Weg p über e zusammen;
            if best_val <_lex (c_p, r_p) then break;
            if r_p > UBR then continue;
            Lösche Eintrag aus w;
            best_val ← (c_p, r_p);
        if kein Eintrag in w von t markiert und r_new_val ≤ UBR then
            Füge new_val zu list(w) hinzu;
            Streiche nicht Pareto-optimale Wege aus list(w);
else
    foreach Kante e = (u,v) ∈ E do
        new_val := (c_new_val, r_new_val) ← dist_min + (c_e, r_e)
        foreach Eintrag in u von s markiert do
            Füge Teilwege zu s-t-Weg p über e zusammen;
            if best_val <_lex (c_p, r_p) then break;
            if r_p > UBR then continue;
            Lösche Eintrag aus u;
            best_val ← (c_p, r_p);
        if kein Eintrag in u von s markiert und r_new_val ≤ UBR then
            Füge new_val zu list(u) hinzu;
            Streiche nicht Pareto-optimale Wege aus list(u);
```

Algorithmus 7: Der bidirektionale erweiterte Dijkstra-Ansatz für das CSP-Problem (Hauptschleife)

Suchen wir hingegen den kostenminimalen Weg, so lässt sich das Abbruchkriterium aus Satz (2.5) relativ problemlos übernehmen, da wir weiterhin den kostenminimalen Weg suchen. Die einzige Änderung liegt darin, dass wir numehr zwei Kostenarten haben. Es wäre möglich, das Abbruchkriterium nur auf die Kosten anzuwenden, dann wäre der ermittelte Weg aber unter Umständen nicht Pareto-

optimal, da ein weiterer Weg mit gleichen Kosten einen niedrigeren Ressourcenverbrauch aufweisen könnnte. Deswegen führen wir die Vergleichsoperation im folgenden Satz nach der lexikographischen Ordnung durch.

Satz 3.4. *Sei* $z := \min_{lex}\{dist_s(u) + (c_{(u,v)}, r_{(u,v)}) + dist_t(v), u \text{ von } s \text{ aus markiert, } v \text{ von } t \text{ aus markiert}\}$ *und* $z_0 := \min_{lex}\{d_r(v), r \in \{s,t\}, v \text{ nicht von } r \text{ aus markiert}\}$. *Gilt* $2 \cdot z_0 >_{lex} z$, *so ist der zu z gehörende Weg der ressourcenbeschränkte kürzeste Weg für das vorliegende Problem.*

Beweis. Der Beweis verläuft analog zum Beweis von Satz (2.5). □

Das *POCP*-Problem ist mit diesem Ansatz nicht effizient zu lösen. Da die gefundenen Wege nicht in lexikograpischer Reihenfolge ermittelt werden und teilweise nicht einmal Pareto-optimal sind, müsste im schlechtesten Fall der Algorithmus solange ausgeführt werden, bis kein unmarkierter Listeneintrag mehr existiert, der die Ressourcenschranke nicht verletzt, bevor dann aus den gefundenen Wegen die k kostenminimalen Pareto-optimalen ermittelt werden. Da dies zu keiner Beschleunigung dem erweiterten Dijkstra-Algorithmus gegenüber führen würde, haben wir diesen Fall nicht implementiert und nur Varianten zur Lösung der *CSP*- und *DB*-Probleme erstellt.

3.3.4 Die bidirektional zielgerichtete Version

Da - wie in den vorangegangen Abschnitten gezeigt - der zielgerichtete und der bidirektionale Ansatz zur beschleunigten Lösung des Kürzeste-Wege-Problems mit Nebenbedingung benutzbar sind, ist auch der Ansatz, der beide Verfahren gleichzeitig anwendet, auf unsere neuen Probleme anwendbar.

Es treten hierbei die gleichen Schwierigkeiten auf wie beim Kürzeste-Wege-Problem ohne Nebenbedingung, denn sowohl unser zielgerichtetes als auch unser bidirektionales Verfahren haben im erweiterten Fall die gleiche Verlaufsstruktur wie bei der Ermittlung des kürzesten Weges ohne Nebenbedingung. Somit ergibt sich bei der Zusammensetzung eines Weges p aus Teilwegen $s \to u$ und $v \to t$ durch den bidirektional zielgerichteten Ansatz analog zu den Betrachtungen in Abschnitt 2.3.3 folgende Formel, wenn o.B.d.A. zuletzt ein Eintrag in u von s aus markiert wurde:

$$\begin{aligned}(c'_p, r'_p) &= dist'_s(u) + (c'_e, r'_e) + dist'_t(v) \\ &= (c_p, r_p) + (LBC(s,v), LBR(s,v)) + (LBC(v,t), LBR(v,t)) \\ &- 2 \cdot (LBC(s,t), LBR(s,t))\end{aligned}$$

Somit verfälschen auch hier die unteren Schranken der Wege von Start- und Zielknoten zum Verbindungsknoten v die Werte des transformierten Kostenpaars des

ermittelten Weges. Wir benötigen also erneut ein Abbruchkriterium, welches sich auf die vom bidirektional zielgerichteten Ansatz verwendeten Distanzwerte $dist'$ bezieht, aber die Ermittlung des hinsichtlich der Originalkantengewichte (c_p, r_p) lexikographisch kleinsten Weges p ermöglicht.

Zu diesem Zwecke passen wir unser Abbbruchkriterium aus Satz (2.6) an die erweiterte Fragestellung an.

Satz 3.5. *Sei best_val das Originalkostenpaar des hinsichtlich der Ausgangsgewichte lexikographisch kleinsten s-t-Weges p, der bisher vom bidirektional zielgerichteten Ansatz gefunden wurde. p ist optimal, falls für den kleinsten unmarkierten Listeneintrag dist'_min folgende Ungleichung gilt:*

$$dist'\_min \geq_{lex} best\_val - (LBC(s,t), LBR(s,t))$$

Beweis. Der Beweis verläuft analog zum Beweis von Satz (2.6). □

while *Es existiert unmarkierter Eintrag mit endlicher Distanz* **do**
- Markiere unmarkierten Eintrag mit minimaler Distanz $dist'_{min}$;
- **if** $dist'\_min \geq_{lex} best\_val - (LBC(s,t), LBR(s,t))$ **then** return gefundenen Weg;
- Sei v der der Liste zugehörige Knoten;
- **if** *Eintrag von s aus markiert* **then**
  - **foreach** *Kante* $e = (v,w) \in E$ **do**
    - $new\_val' := (c'_{new\_val'}, r'_{new\_val'}) \leftarrow dist'_{min} + (c'_e, r'_e)$
    - **foreach** *Eintrag in w von t markiert* **do**
      - Füge Teilwege zu s-t-Weg p über e zusammen;
      - **if** $best\_val <_{lex} (c_p, r_p)$ **then** break;
      - **if** $r'_p > UBR'$ **then** continue;
      - Lösche Eintrag aus w;
      - best_val $\leftarrow (c_p, r_p)$;
    - **if** *kein Eintrag in w von t markiert und* $r'_{new\_val'} \leq UBR'$ **then**
      - Füge new_val' zu $list(w)$ hinzu;
      - Streiche nicht Pareto-optimale Wege aus $list(w)$;
- **if** *Eintrag von t aus markiert* **then**
  - ...

Algorithmus 8: Der bidirektional zielgerichtete erweiterte Dijkstra-Ansatz für das CSP-Problem (Ausschnitt)

Auch in diesem Fall könnte man das Abbruchkriterium nur auf die Kosten des Weges anwenden, wenn der zu findende Weg nicht notwendigerweise Pareto-

optimal sein soll. Da der Aufwand sich dabei jedoch nur unwesentlich verringern würde, benutzen wir bei der Vergleichsoperation die Kostenpaare.

Genau wie beim rein bidirektionalen Ansatz haben wir keine Version zur Lösung des *POCP*-Problems entwickelt. Da auch hier nicht jeder gefundene Weg gleich Pareto-optimal ist, wäre zunächst eine im schlechtesten Fall exponentielle Anzahl an Wegen zu ermitteln, was zu einer inakzeptablen Laufzeit führen würde. Somit löst dieser Algorithmus das *CSP*- und das *DB*-Problem.

3.3.5 Versionen mit zusätzlichem Abbbruchkriterium

Da wir das *DB*-Problem mit erweiterten Dijkstra-Methoden lösen, werden im Verlauf des Algorithmus die Kosten so niedrig wie möglich gehalten, während für die Ressourcen keine Minimierung durchgeführt wird, sondern nur ein Test auf Zulässigkeit vorgesehen ist. Sollte die Kostenschranke UBC genauso weit vom Kostenwert der kostenminimalen Lösung abweichen wie die Ressourcenschranke UBR vom Ressourcenverbrauch des ressourcenschonendsten Weges, so werden unsere betrachteten Teilwege deswegen eher dazu tendieren, einen unzulässigen Ressourcenverbrauch vorzuweisen, als dass sie hinsichtlich der Kosten über der vorgegebenen Schranke liegen. Basierend auf dieser Feststellung erscheint es also erfolgversprechend, vom Dijkstra-Algorithmus ermittelte Teilwege durch einen möglichst ressourcengünstigen Restweg zu komplettieren, um somit schneller einen zulässigen *s*-*t*-Weg zu finden.

Unsere zielgerichteten Versionen berechnen kürzeste Wege von allen Knoten zum Zielknoten bei der Ermittlung der unteren Schranken. Speichert man bei der Ermittlung der ressourcengünstigsten Wege auch deren Kosten c_{LBR}, so steht uns für jeden Knoten v das Kostenpaar sowie der Wegverlauf des bezüglich der Ressourcen effizientesten Weges von v zum Zielknoten t zur Verfügung.

Betrachten wir zunächst den zielgerichteten erweiterten Dijkstra-Algorithmus. Wir erweitern die Hauptschleife und untersuchen für jeden Eintrag, den wir in eine Liste einfügen wollen, zunächst die Kosten, die entstehen würden, wenn der entsprechende Weg p mit dem ressourcengünstigsten Teilweg zum Zielknoten vervollständigt werden würde. Damit der Weg zulässig ist, muss folgende Ungleichung für die Kosten gelten:

$$c'_p - LBC(v,t) + c_{LBR_{(v,t)}} \leq UBC'$$

Für die Ressourcen muss keine neue Ungleichung erstellt werden; es genügt, wenn der transformierte Ressourcenwert unter der transformierten Ressourcenschranke bleibt. Liegen die untersuchten Werte unter den vorgegebenen Schranken, so ist der Weg zulässig und eine Lösung gefunden. Ansonsten wird der erweiterte Dijkstra-Algorithmus normal weitergeführt.

while *Es existiert ein unmarkierter Listeneintrag mit endlicher Distanz.* **do**
  Markiere unmarkierten Listeneintrag mit minimaler Distanz $dist'_{min}$;
  Sei v der der Liste zugehörige Knoten;
  foreach *Kante* $e = (v,w) \in E$ **do**
    $new\_value' := (c'_{new\_value'}, r'_{new\_value'}) \leftarrow dist'_{min} + (c'_e, r'_e)$
    if $r'_{new\_value'} \leq UBR'$ **then**
      if $c'_{new\_value'} - LBC(w,t) + c_{LBR_{(w,t)}} \leq UBC'$ **then**
        return vervollständigter Weg;
      Füge new_value' zu $list(w)$ hinzu;
      Streiche nicht Pareto-optimale Wege aus $list(w)$;

Algorithmus 9: Der zielgerichtete erweiterte Dijkstra-Algorithmus mit zusätzlichem Abbruchkriterium (Hauptschleife)

Beim bidirektional zielgerichteten Ansatz ist ein solches Abbruchkriterium natürlich ebenfalls einsetzbar, wobei hier selbstverständlich auch bei den ressourcengünstigsten Wegen vom Startknoten zu allen Knoten der zugehörige Kostenwert gespeichert werden muss.

Im Allgemeinen führt dieses Kriterium zu einer signifikanten Verringerung der Hauptschleifendurchläufe; allerdings ist ein Durchlauf aufgrund der zusätzlich zu überprüfenden Kriterien aufwändiger als bei den Versionen, die den kostenminimalen Weg suchen. Deswegen ergeben sich Nachteile, wenn die obere Kostenschranke nur sehr knapp über dem letztlich ermittelten minimal zulässigen Kostenwert liegt, da in diesem Falle die Anzahl der Schleifendurchläufe fast genauso hoch wie beim Ansatz ohne zusätzliches Abbruchkriterium sein dürfte. Generell ist zu erwarten, dass die ermittelten Lösungen bezüglich der Kosten bisweilen recht weit vom minimalen zulässigen Weg abweichen. Somit eignen sie sich für Problemstellungen, in denen eher auf Geschwindigkeit als auf die Güte der Lösung geachtet wird.

3.4 Weitere Lösungsmethoden

Bevor wir im nächsten Kapitel die Algorithmen bezüglich ihrer Geschwindigkeit testen, gehen wir noch einmal etwas genauer auf die verwendeten Lösungsmethoden im *CMCF*-Projekt und im *CNOP*-Paket ein.

3.4.1 Der Labeling-Ansatz im CMCF-Projekt

Der bei diesem Route-Guidance-Projekt verwendete Algorithmus ist im Wesentlichen eine Variante des erweiterten Dijkstra-Algorithmus, der sowohl den ressour-

cenbeschränkten kürzesten Weg suchen kann als auch einen Weg, der nur unter den vorgegebenen Schranken bleibt, und einige simple Beschleunigungsmethoden enthält. Hierbei werden gewisse Eigenschaften des Hauptprogramms benutzt, die nicht generell für alle ressourcenbeschränkte Kürzeste-Wege-Probleme genutzt werden können.

So wird bei der *CMCF*-Implementation unser betrachtetes Problem wiederholt als Unterroutine aufgerufen. Dabei werden immer die gleichen Ressourcenwerte für die Kanten als Ausgangsdaten übergeben, aber die Kosten werden vor jedem Aufruf neu ermittelt. Aus diesem Grund wird zu Beginn des Programms einmalig der kürzeste *s*-*t*-Weg *p* bezüglich der Ressourcen berechnet. Liegt er über der zu benutzenden Ressourcenschranke, so werden wir keine Lösung für unser Problem finden. Ansonsten wird vor jedem Aufruf der Kostenwert von *p* ermittelt. Die Kosten des Weges werden dann als obere Kostenschranke *cost_bound* für unsere Lösung benutzt, da jeder andere *s*-*t*-Weg mehr Ressourcen verbraucht und damit bei Verursachung höherer Kosten von unserem Ausgangsweg dominiert würde.

Suchen wir nur eine Lösung für das *DB*-Problem, so überprüfen wir, ob unser zuerst ermittelte Weg auch unter der Kostenschranke *UBC* bleibt. In diesem Fall können wir abbrechen und ihn als Lösung zurückgeben. Ansonsten starten wir die übliche Dijkstra-Prozedur durch Markierung unseres Startknotens.

In der Hauptschleife werden ähnlich wie bei unseren zielgerichteten Versionen kürzeste Kostenwege von allen Knoten zum Zielknoten benötigt. Allerdings werden keine transformierten Kantengewichte benutzt; hier liegt ihr Nutzen darin, voraussagen zu können, ob ein Teilweg zu einem für uns relevanten *s*-*t*-Weg führen kann. Haben wir gerade einen Knoten markiert, so komplettieren wir den zugehörenden Teilweg mit dem bezüglich der Kosten minimalen Restweg zu einem *s*-*t*-Weg. Ist dessen Ressourcenverbrauch zulässig und liegen seine Kosten unter der oberen Schranke *cost_bound*, so wird dieser Weg zur neuen oberen Schranke gemacht. Suchen wir nur einen Weg für das *DB*-Problem, so können wir abbrechen, sobald $cost\_bound \leq UBC$ gilt, ansonsten werden wie beim üblichen Dijkstra-Algorithmus die ausgehenden Kanten des zum gerade markierten Listeneintrag gehörigen Knotens betrachtet. Es findet also eine schrittweise Reduzierung unserer Kostenschranke statt. Dies wird solange fortgesetzt, bis der lexikograpisch kleinste Listeneintrag einen höheren Kostenwert als *cost_bound* hat. Der Weg, welcher die obere Kostenschranke erzeugte, ist daraufhin die kostenminimale Lösung, da er nicht mehr unterboten werden kann. Kommt man hingegen beim *DB*-Problem zu diesem Punkt, so bedeutet dies, dass kein *s*-*t*-Weg zulässig ist, da die bisherige obere Schranke bezüglich der Kosten noch nicht zulässig war.

Die in diesem Ansatz verwendeten Beschleunigungsmethoden sind auf unsere Algorithmen nicht anwendbar. Die schrittweise Senkung der oberen Kostenschranke durch Ermittlung von besseren Wegen benötigt wie gesehen kürzeste Wege von allen Knoten zu den Zielknoten, wären also nur für unsere zielgerichteten Ansätze von Belang. Diese benutzen jedoch immer transformierte Kostenpaare, weswe-

gen ihre Kostenwerte den bei Komplettierung des betrachteten Teilweges durch den kürzesten Weg zum Zielknoten entstandenen Resultaten entsprechen. Da wir immer den lexikographisch kleinsten Eintrag in der Hauptschleife benutzen, gibt es keinen zulässigen *s*-*t*-Weg, der einen kleineren Kostenwert hat als der durch den kürzesten Weg zum Zielknoten erweiterte momentan betrachtete Teilweg. Somit wird die Speicherung einer oberen Schranke hinfällig. Für die Versionen zur Lösung des *DB*-Problems gilt dies ebenfalls, da auch sie mit transformierten Kantengewichten arbeiten.

3.4.2 Die 2-Phasen-Methode im CNOP-Paket

Der für das *CNOP*-Paket entwickelte Algorithmus besteht aus zwei Hauptschritten, für die es mehrere Lösungsmöglichkeiten gibt.

Zunächst wird das ressourcenbeschränkte Problem als ganzzahliges lineares Problem formuliert, die Ressourcenbeschränkung wird relaxiert, und schließlich wird das sich ergebende lineare Programm gelöst, um für das Anfangsproblem Schranken zu ermitteln. Die zweite Stufe besteht darin, aus dem durch die Schranken vorgegebenen Bereich die Optimallösung zu finden.

Das *CNOP*-Paket bietet zum experimentellen Vergleich mehrere Algorithmen für beide Phasen an; wir beschränken uns in dieser Arbeit jedoch auf die schnellsten Methoden.

Der Hüllenansatz

Für den ersten Schritt wird der sogenannte *Hüllenansatz* verwendet, welcher im Wesentlichen der bereits erwähnten Methode von Handler und Zang ähnelt. Formulieren wir unser Problem zunächst als ganzzahliges lineares Optimierungsproblem. Für jeden *s*-*t*-Weg führen wir eine 0-1-Variable x_p ein. Dann löst

$$\min \sum_p c_p x_p,$$

$$\text{so dass } \sum_p x_p = 1,$$

$$\sum_p r_p \leq UBR,$$

$$0 \leq x_p \leq 1,\ x_p \text{ ganzzahlig}.$$

das ressourcenbeschränkte Kürzeste-Wege-Problem, denn nur ein x_p wird nicht den Wert 0 erhalten, womit auch nur ein Weg betrachtet wird. Wenn wir die Ganzzahligkeitsbedingung fallen lassen, so ergibt sich eine *LP-Relaxation*, deren Lösung *LBC* eine untere Schranke für unser Ausgangsproblem darstellt. Da es jedoch eine exponentiell große Anzahl an zulässigen *s*-*t*-Wegen und damit an Variablen gibt, ist dieses lineare Problem in der Form nicht effizient lösbar. Betrachten wir also das zugehörige duale Problem. Es lässt sich formulieren als:

$$\max u + UBR \cdot v,$$

$$\text{so dass } u + r_p v \leq c_p \ \forall p,$$

$$v \leq 0.$$

Der Hüllenansatz beruht nun auf folgender geometrischer Interpretation. Wir stellen uns einen zweidimensionalen Raum mit Koordinatensystem (r,c) vor. Jedes Paar (u,v) des dualen Problems wird in diesem Raum als die durch $c = v \cdot r + u$ gegebene Gerade interpretiert und jede Bedingung $u + r_p v \leq c_p$ als Punkt (r_p, c_p). Unser duales Problem entspricht dann der Suche nach einer Geraden mit nichtnegativem Anstieg (da $v \leq 0$), die für $r = UBR$ einen maximalen *c*-Wert hat und für die alle Nebenbedingungspunkte (r_p, c_p) entweder auf oder über ihr liegen. Wäre dies nicht der Fall, so würde unsere Gerade in r_p einen größeren *c*-Wert als c_p vorweisen und damit die Nebenbedingung des dualen Problems verletzen.

Zunächst wird der ressourcengünstigste Weg des Ausgangsproblems ermittelt, dessen Kosten eine obere Schranke unserer Lösung darstellen. Ist er zulässig, so berechnen wir den kostengünstigsten Weg. Dieser ist unsere Optimallösung, falls sein Ressourcenverbrauch die obere Schranke nicht überschreitet. Ansonsten wird sein Kostenwert zur unteren Schranke unsere Lösung.

Jeder *s*-*t*-Weg entspricht einer Nebenbedingung im dualen Problem, ist also in unserem Graphen als Punkt darstellbar. Wir erstellen eine Gerade durch die Punkte der soeben ermittelten Wege und untersuchen, ob sie eine Nebenbedingung verletzt. Liegen einige Nebenbedingungspunkte noch unter ihr, so bewegen wir unsere Gerade in normaler Richtung nach unten, bis wir auf den letzten Punkt stoßen oder unsere Gerade die durch $r = UBR$ beschriebene Gerade im Punkt $(UBR, 0)$ schneidet. Sind alle Nebenbedingungen erfüllt, so ist die Optimallösung der dualen Relaxation durch den *c*-Wert des Schnittpunkts unserer Geraden mit der durch $UBR = r$ definierten Gerade gegeben. Dies ist also die untere Schranke für die Kosten unseres Ausgangsproblems. Da der zuletzt betrachtete Nebenbedingungspunkt einem Weg in unserem eigentlichen Problem entspricht, ist sein Kostenwert als obere Schranke für unsere zu ermittelnde Lösung verwendbar.

Ist die Lösung des dualen Problems noch nicht gefunden, so erstellen wir eine neue Gerade, die den zuletzt erreichten Nebenbedingungspunkt mit einem unserer

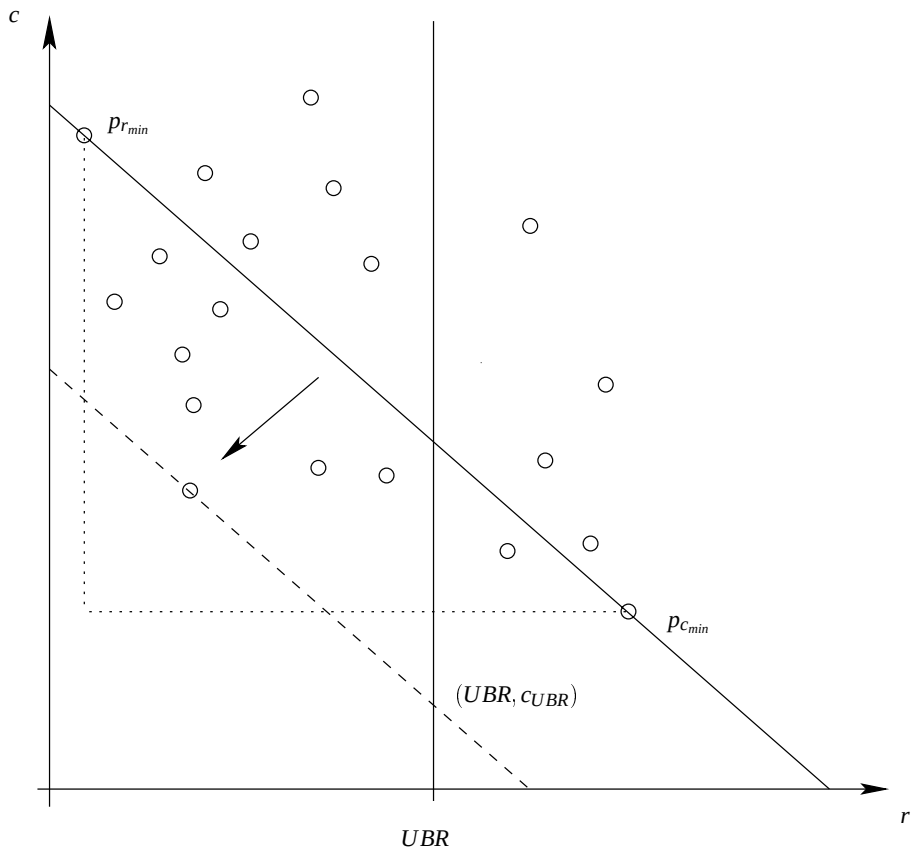

Abbildung 3.5: Geometrische Interpretation des dualen Problems

beiden Ausgangspunkte verbindet. Wir wählen hierbei immer den Punkt, der zu einer Gerade mit größerem c-Wert im Punkt (UBR, c) führt. Die Prozedur wird daraufhin solange wiederholt, bis keine Nebenbedingung mehr verletzt ist.

Das soeben erläuterte Verfahren ist sehr schnell. Sind alle Kosten und Ressourcen ganzzahlig und durch die Werte C bzw. R beschränkt, so beweist Ziegelmann [21] gar eine polynomiale Laufzeit von $O(\log(n \cdot R \cdot C)(n \log n + m))$. Er zeigt auch, dass die Schranken für bestimmte Problemstellungen sehr weit von der Optimallösung abweichen können, behauptet aber, dass sie im Normalfall sehr nahe am letztlich zu ermittelnden Wert liegen, was wir bei unseren Tests überprüfen werden.

Problemreduktion

Aufgrund der im letzten Abschnitt dargestellten und für gewisse Eingabegrößen sogar polynomialen Laufzeit für die Ausführung des Hüllenansatzes ist davon auszugehen, dass die im zweiten Schritt folgende Ermittlung der Optimallösung der aufwändigere Teil dieses Algorithmus ist. Aus diesem Grund kann es von Vorteil

sein, zunächst Knoten und Kanten, über die der Lösungsweg nicht führen kann, aus dem Graphen zu löschen, um das Restproblem zu verkleinern. Dies kann zum Teil schon vor der ersten Phase geschehen. Ist $R_{(u,v)}$ der minimale Ressourcenverbrauch für einen Weg von u nach v, so wird ein Knoten v, für den

$$R_{(s,v)} + R_{(v,t)} > UBR$$

gilt, nicht auf dem Lösungsweg liegen; er kann also aus dem Graphen gelöscht werden. Eine analoge Formel existiert für Kanten.

Nachdem die erste Phase abgeschlossen ist, können wir unser Problem auch bezüglich der Kosten reduzieren. Sei ein zulässiger Weg p mit Gewichten (c_p, r_p) gegeben und ein beliebiges $v \leq 0$. Haben wir eine obere Kostenschranke UBC ($Upper\_Bound\_Cost$), so gilt folgende Ungleichung:

$$c_p - v \cdot r_p + v \cdot UBR \leq UBC$$

Sei nun $\bar{c}_{(u,v)}$ der kürzeste u-v-Weg bezüglich der Kosten $\bar{c}_e = c_e - v \cdot r_e$. Gilt für einen Knoten u nun die Ungleichung

$$\bar{c}_{(s,u)} + \bar{c}_{(v,t)} + v \cdot UBR > UBC, \tag{3.3}$$

so existiert kein zulässiger s-t-Weg durch u, dessen Kosten unter UBC liegen. Somit kann dieser Knoten aus dem Graphen gelöscht werden. Auch hier lässt sich eine analoge Formel für Kanten ableiten.

Es stellt sich nun die Frage, für welche Wahl des Wertes v das Problem am besten reduziert wird. Sei $(\bar{u}, \bar{v})$ die Lösung der im Hüllenansatz auftretenden dualen Relaxation. Wie wir bei der geometrischen Interpretation der Fragestellung gesehen haben, werden alle Nebenbedingungen des dualen Problems erfüllt; es gilt also $\bar{u} \leq c_p - \bar{v} \cdot r_p$ für alle Wege p. Da wir während des Verlaufs des Hüllenansatzes unsere Gerade immer wieder neu durch Nebenbedingungspunkte ziehen, liegt am Ende mindestens einer dieser Punkte auf der Geraden, die die Lösung liefert. Somit gilt für mindestens einen Weg p die soeben genannte Eigenschaft mit Gleichheit. Deswegen ist $\bar{u}$ der kürzeste Weg bezüglich der transformierten Kosten $\bar{c}_e = c_e - \bar{v} \cdot r_e$.

Da die Werte $\bar{u}$ und $\bar{v}$ durch Maximierung von $\bar{u} + \bar{v} \cdot UBR$ entstanden sind, erscheint die Wahl von $\bar{v}$ für den Wert v in (3.3) also sehr erfolgsversprechend, weswegen sie auch im *CNOP*-Paket verwendet werden.

Die von Ziegelmann durchgeführten Tests lassen nicht erkennen, ob das Einführen solcher Problemreduktionen tatsächlich zu einem niedrigerem Aufwand führt; deswegen werden wir bei unseren Geschwindigkeitsvergleichen die 2-Phasen-Methode mit und ohne Reduktionen testen.

Ermittlung der Lösung

Das *CNOP*-Projekt hat während seiner Entwicklung verschiedene Ansätze zur letztlichen Ermittlung der Lösung des Problems verwendet. Wurde anfangs noch ein *k* Kürzeste-Wege-Algorithmus benutzt, so hat sich inzwischen eine mit skalierten Kosten arbeitende Labeling-Methode als das schnellste Verfahren erwiesen.

Betrachten wir zunächst noch einmal die geometrische Interpretation der Situation nach Vollendung des Hüllenansatzes. Die Lösung befindet sich im Dreieck, welches durch die obere und untere Schranke sowie die durch $r = UBR$ gegebene Gerade definiert ist.

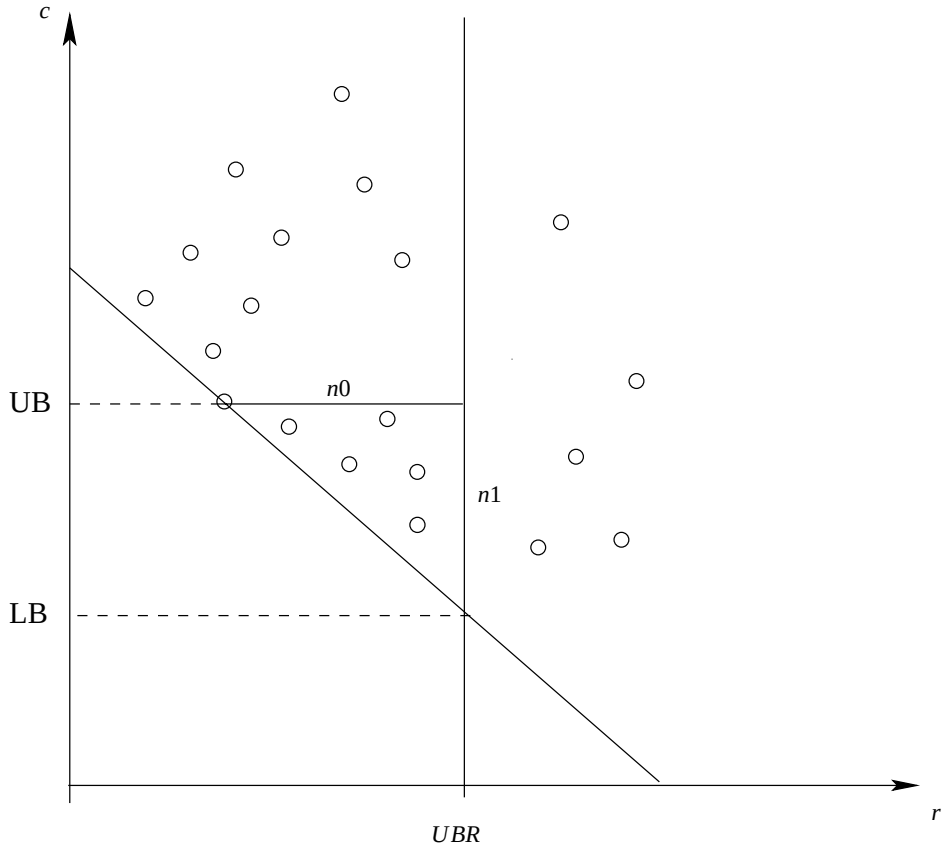

Abbildung 3.6: Ausgangssituation nach dem Hüllenansatz.

Sei nun $n0$ der Ressourcenwert der unteren minus dem der oberen Schranke und $n1$ die Kosten der oberen minus der Kosten der unteren Schranke. Bezüglich der bei der Lösung des Hüllenansatzes ermittelten optimalen Steigung v ergibt sich also die Ungleichung $v := -\frac{n1}{n0}$.

Ziegelmann benutzt nun den gleichen Skalierungsansatz, den wir bei den Kostenreduktionen vorgestellt haben. Die dort benutzten $\bar{c_e}$ werden zum Zwecke der besse-

ren Lesbarkeit noch ein weiteres Mal multipliziert. Es entstehen sogenannte *lower bound completion costs* für alle Kanten e.

$$lb\_cost(e) = n0 \cdot \bar{c_e} = n0 \cdot (c_e + \frac{n1}{no} \cdot r_e) = n0 \cdot c_e + n1 \cdot r_e$$

Ziel dieser Transformation ist es, mit Hilfe einer zu (3.3) analogen Formel möglichst viele Teilwege ausschließen zu können und somit schnell die Lösung zu finden.

Vor dem Start des Labeling-Ansatzes werden zunächst kürzeste Wege zum Zielknoten bezüglich der Kosten, der Ressourcen sowie der *lower bound completion costs* erstellt. Der größte Unterscheid zum normalen Dijkstra-Algorithmus besteht nun darin, dass die Listeneinträge nicht bezüglich der Originalkosten geordnet sind. Es wird für jeden Eintrag ein Wert *lb_cost_to_target* berechnet, der sich aus den *lower bound completion costs* zum betrachteten Knoten und dem kürzesten Restweg zum Zielknoten hinsichtlich dieser Kostenart zusammensetzt. Wie wir sehen werden, ermöglicht eine solche Ordnung ein schnelleres Abbruchkriterium.

Jedes Mal, wenn ein neuer Teilweg $s \rightarrow v$ betrachtet wird, überprüfen wir erst drei Bedingungen, bevor wir ihn in die Liste von v eintragen. Wir erweitern seinen Ressourcenverbrauch um den Verbrauch des ressourcengünstigsten Weges zum Zielknoten und testen, ob der sich ergebende Wert noch unter *UBR* liegt. Ist dies der Fall, so fügen wir die Kosten des kürzesten Weges zum Zielknoten seinem Kostenwert hinzu und stellen fest, ob die obere Kostenschranke *upper_bound_cost* aus der Relaxation einen größeren Wert hat. Ist auch diese Bedingung erfüllt, ermitteln wir den *lb_cost_to_target*-Wert von v. Analog zu (3.3) muss nun folgende Eigenschaft gelten:

$$\frac{(lb\_cost\_to\_target_v - n1 \cdot UBR)}{n0} < upper\_bound\_cost \tag{3.4}$$

Sind all unsere Bedingungen erfüllt, kann der Weg in die Liste eingetragen werden.

Sobald ein Eintrag in t markiert ist, der zulässig ist und bezüglich der Kosten unter der oberen Schranke liegt, wird sein Kostenwert zur neuen oberen Schranke. Somit wird Bedingung (3.4) schwieriger zu erfüllen.

Da wir immer den bezüglich des lb_cost_to_target-Wertes kleinsten Weg markieren, können wir abbrechen, sobald ein gerade markierter Eintrag die Bedingung (3.4) nicht erfüllt.

Der soeben vorgestellte Ansatz ist für unsere Algorithmen nicht nutzbar, da er von der Existenz einer oberen Kostenschranke abhängt. Umgekehrt scheint es auch nicht möglich, diesen durch eine zielgerichtete Transformation noch zu verbessern. Allerdings ist es denkbar, den Labeling-Ansatz bidirektional laufen zu lassen. Da eine obere Kostenschranke existiert, würde auch für den Fall, dass keine zulässige Lösung existiert, schnell ein Resultat geliefert werden.

3.5 Überblick über die vorgestellten Algorithmen

Zusammenfassend stellen wir hier noch einmal alle Algorithmen einschließlich ihrer Eigenschaften in tabellarischer Form dar. Hierbei steht das Kürzel *zg* für einen zielgerichteten Ansatz und *bidir* für bidirektionale Methoden.

		löst		
Index	Algorithmus	CSP	POCP	DB
1	Dijkstra erweitert	ja	ja	ja
2	Dijkstra zg	ja	ja	ja
3	Dijkstra bidir optimal	ja	nein	ja
4	Dijkstra zg bidir optimal	ja	nein	ja
5	CMCF-Labeling	ja	nein	ja
6	CNOP mit/ohne Reduktion	ja	nein	ja
7	Dijkstra bidir	nein	nein	ja
8	Dijkstra zg bidir	nein	nein	ja
9	Dijkstra bidir + Abbruch	nein	nein	ja
10	Dijkstra zg bidir + Abbruch	nein	nein	ja

Tabelle 3.1: Überblick über die Eigenschaften der vorgestellten Algorithmen; *CSP* ist das ressourcenbeschränkte Kürzeste-Wege-Problem, *POCP* das Pareto-optimale ressourcenbeschränkte Wege Problem, *DB* das doppelt beschränkte Problem.

Es ergeben sich hiermit sechs Algorithmen für das *CSP*-Problem. Wir werden keine eigenen Tests für das *POCP*-Problem durchführen, sondern die beiden Ansätze, die diese Fragestellung lösen können, nach der Ermittlung eines ressourcenbeschränkten Weges weiterlaufen lassen, bis sie eine bestimmte Anzahl an Pareto-optimalen Wegen gefunden haben. Dabei ist von Interesse, welcher Mehraufwand entsteht.

Obwohl alle vorgestellten Verfahren das *DB*-Problem prinzipiell lösen können, werden wir uns in diesem Falle auf die letzten vier Algorithmen konzentrieren, da diese im Gegensatz zu den erstgenannten explizit nur einen Weg unter den Schranken suchen. Algorithmen 7 und 8 in unserer Tabelle entsprechen den bidirektionalen und bidirektional zielgerichteten Verfahren, welche zwar nicht über das zusätzliche Abbruchkriterium aus Abschnitt 3.3.5 verfügen, jedoch beim erstgefundenen *s*-*t*-Weg abbrechen, ohne wie bei Algorithmen 3 und 4 Kostenminimalität zu fordern.

Kapitel 4

Implementation und Geschwindigkeitsvergleiche

Sowohl das *CMCF*-Projekt als auch das *CNOP*-Paket haben eine Vielzahl an generierten und realen Verkehrsnetzen zur Verfügung, auf denen ihre Methoden getestet werden können. Wir möchten nun die im letzten Kapitel vorgestellten Algorithmen ebenfalls auf diese Instanzen anwenden, um Rückschlüsse bezüglich der benötigten Laufzeiten schließen zu können. Soweit es möglich ist, wollen wir auch die Ansätze zur Lösung des ressourcenbeschränkten Kürzeste-Wege-Problems von *CMCF* und *CNOP* direkt miteinander vergleichen.

4.1 Angaben zur Implementation

Wir haben als Programmiersprache *C++* eingesetzt, kompiliert wurde mit dem GNU C-Compiler gcc 2.9.5. Die Kompilierung wurde mit der Optimierungsoption -O3 ausgeführt. Getestet wurde auf einem *UltraSparc-II*-Rechner mit 4.0 GB RAM und 448 Mhz.

4.1.1 Verwendete Datenstrukturen

Um einen aussagekräftigen Vergleich mit dem Labeling-Ansatz des *CMCF*-Projektes zu ermöglichen, benutzen wir die gleichen Datenstrukturen zur Verwaltung der Teilwege. Die unmarkierten Listeneinträge werden in einem d-Heap der Größe 4 gespeichert. Zusätzlich wird für jeden Knoten eine lexikographisch geordnete Liste bereitgehalten. Wie bereits in Kapitel 2 gesehen, wäre die Nutzung eines Fibonacci-Heaps um einiges effizienter.

4.1.2 Korrektur von Rechnerungenauigkeiten

Fließkommazahlen lassen sich bekannterweise nicht genau von Computern darstellen. Je nach Rechnerarchitektur variiert die sich ergebende Ungenauigkeit. Da diese jedoch erst viele Stellen nach dem Komma erscheint, sind die Auswirkungen auf einen Labeling-Ansatz zur Lösung des Kürzeste-Wege-Problems mit Nebenbedingung eher gering. Da wir immer nach dem lexikographisch kleinsten unmarkierten Listeneintrag suchen, könnte eine auftretende Rechnerungenauigkeit zur Wahl eines Eintrages führen, der geringfügig schlechter als das eigentliche Minimum ist. Die Wahrscheinlichkeit, dass dies zu einer falschen Lösung führt, wird beim im *CMCF*-Projekt verwendeten Ansatz jedoch als so niedrig eingeschätzt, dass keine Gegenmaßnahmen getroffen worden sind. Die 2-Phasen-Methode des *CNOP*-Pakets lässt ebenfalls Ungenauigkeiten zu, da diese den letztlich ermittelten Optimalwert nur unwesentlich verändern können.

Ungleich gravierender sind die Konsequenzen für zielgerichtete Labeling-Ansätze, da hier transformierte Kosten verwendet werden, welche durch Addition und Substraktion eines Wertes auf die Ausgangskosten entstehen. Liegt die ausgehende Kante (v,w) eines gerade markierten Listeneintrags zum Knoten v genau auf dem kürzesten v-t-Weg, so gilt für ihre Kosten:

$$c'_{(v,w)} = c_{(v,w)} + LBC(w,t) - LBC(v,t) = LBC(v,t) - LBC(v,t) = 0$$

Der in die Liste von w einzufügende Distanzwert entspricht also genau dem gerade gefundenen lexikographisch kleinsten Listeneintrag. Bei der Ermittlung der transformierten Kosten kann die verwendete Substraktion jedoch dazu führen, dass die neuen Kantengewichte einen geringfügig negativen Wert erhalten, woraufhin der einzufügende Wert kleiner als der gerade ermittelte Eintrag wird.

Existiert in der Liste von w ein bereits markierter Eintrag, der den gleichen Kostenwert hat wie unser in diesem Schleifendurchlauf markierter Listeneintrag, so wird beim Einfügen des neuen Wertes versucht werden, diesen aus der Liste der Pareto-optimalen Teilwege zu entfernen, da er aufgrund der Rechnerungenauigkeit als dominiert angesehen wird. In einer Heap-Struktur, die markierte Einträge aus dem Heap entfernt, ist dies gleichbedeutend mit dem versuchten Löschen eines nicht mehr vorhandenen Eintrags, was unweigerlich zu einem Programmabsturz führt.

Es scheint keine Methode zu geben, die dieses Problem vollständig löst. Dennoch gibt es Möglichkeiten, die Fehleranfälligkeit zu minimieren. Dies wird erreicht, indem man die zu betrachtenden Werte nach einer vorgegebenen Anzahl an Nachkommastellen abschneidet. Zu diesem Zwecke werden die Ausgangswerte mit Hilfe der Formel

$$new\_value = (old\_value + 5 \cdot 10^{-(factor+1)}) * 10^{factor}$$

zu ganzzahligen Werten skaliert, wobei man dadurch die Rechnerungenauigkeit bis zur vorgegebenen Nachkommastelle eliminiert. Der Aufrundungsfaktor $5 * 10^{-factor+1}$ sorgt dafür, dass sowohl minimal aufgerundete als auch abgerundete Werte richtig skaliert werden.

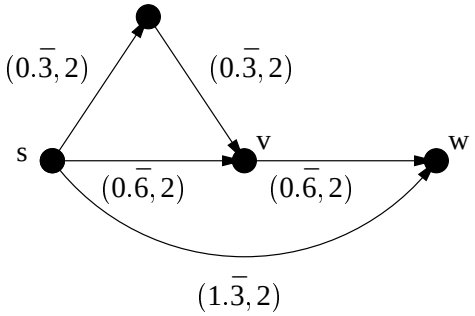

Abbildung 4.1: Auftretende Rechnerungenauigkeiten bei Skalierung der Kantengewichte

Es stellt sich die Frage, zu welchem Zeitpunkt die Werte zu skalieren sind. Macht man dies bereits für die Kantengewichte, so ergibt sich eine erstaunliche Geschwindigkeitsverbesserung von etwa 20%. Dies liegt daran, dass Operationen mit ganzzahligen Werten schneller als bei Fließkommazahlen abgearbeitet werden. Wie wir in Abbildung (4.1) sehen, ist dieser Ansatz jedoch sehr fehleranfällig. Benutzen wir die gerade formulierte Skalierung beispielsweise mit dem Skalierungsfaktor 2, dann scheint der obere *s*-*v*-Weg mit seinem ermittelten Wert von $(0.66, 4)$ Pareto-optimal zu sein, obwohl er eigentlich vom mittleren *s*-*v*-Weg dominiert wird. Versucht man zu skalieren ohne aufzurunden, dann wird für den mittleren *s*-*w*-Weg ein Wert von $(1.32, 4)$ ermittelt, womit auch er Pareto-optimal erscheint, obwohl der untere Weg ihn eigentlich dominiert. Insgesamt liefert dieses Verfahren nur dann korrekte Ergebnisse, wenn alle Kantengewichte eine Höchstanzahl an Nachkommastellen nicht überschreiten. Dies wäre zum Beispiel bei auftretenden Geldbeträgen der Fall. Da wir uns jedoch auf die allgemeine Situation konzentrieren, ist diese Variante nicht einsetzbar.

Um sowohl die Rechnerungenauigkeit als auch den Fall des Löschens eines bereits markierten Eintrages zu vermeiden, müssen wir letztendlich einen etwas umständlichen Weg wählen. Alle Kostenwerte werden weiterhin als Fließkommazahlen gespeichert, bei jeder Vergleichsoperation werden sie jedoch skaliert, um eine richtige lexikographische Ordnung einzuhalten. Außerdem wird eine zusätzliche Abfrage eingebaut, die verhindern soll, dass ein Listeneintrag kleiner als der soeben markierte wird. In diesem Fall wird als Kantengewicht einfach der Wert 0 addiert.

Dieser Ansatz bewirkt eine leichte Verschlechterung der Laufzeit. Wie wir in Tabelle (4.1) sehen, vergrößert sich die Laufzeit um so deutlicher, je genauer man rechnen will. Im Vergleich zum Lösungsansatz ohne Skalierung steigt der Aufwand jedoch nur um wenige Prozente. Aus diesem Grunde benutzen wir diesen

Skalierungsfaktor	Laufzeit in s
-	101.26
2	104.94
4	106.21
6	108.53

Tabelle 4.1: Typische Auswirkung der Skalierung auf die Laufzeit

Ansatz bei unseren Tests.

Es sei jedoch nicht das größte Problem bei Skalierungen verschwiegen. Sind die betrachteten Kosten und Ressourcen schon relativ groß, so kann eine Skalierung dazu führen, dass der maximal zulässige Wert für *long int*-Typen in *C++* überschritten wird. Somit stellt sich also das Problem, dass ein hoher Skalierungsfaktor zwar genaue Ergebnisse verspricht, aber sowohl die Rechengeschwindigkeit verlangsamt als auch im schlimmsten Fall den Algorithmus durch Überschreitung des Maximalwertes fehlerhaft werden lassen kann.

Haben wir eine obere Schranke für die Kosten unserer Lösung zur Verfügung, so können wir den erlaubten Skalierungsfaktor daraus ableiten. Es stellt sich hierbei jedoch die Frage, ob der Aufwand zur Ermittlung dieser Schranke akzeptabel ist oder ob der Nutzer das Restrisiko in Kauf nimmt. Wird zudem eine obere Schranke benutzt, die weit vom eigentlich Kostenwert abweicht, so kann es sein, dass weniger genau skaliert wird, als es durchaus möglich wäre. Zumindest für den Fall des doppelt beschränkten Problems, welches sowohl eine Kosten- als auch eine Ressourcenschranke vorgegeben hat, ist eine solche Vorgehensweise trotz aller Einwände jedoch denkbar.

4.1.3 Einlesen des Verkehrsnetzwerkes

Die zu berechnenden Instanzen werden in separaten Dateien gespeichert, wobei jedes Programm mit einer Einleseroutine ausgestattet worden ist, um diese zu erfassen. Die betrachteten Projekte beziehen sich allerdings auf verschiedene Einleseformate. Während das *CNOP*-Paket *Dat*-Dateien einlesen kann, benötigt *CMCF* das sogenannte *Raw*-Format.

Dat-Dateien beinhalten zu jeder Kante Kosten- und Ressourcenwerte sowie optional graphische Koordinaten zur Darstellung der Knoten. In *CNOP* werden diese Daten in einem *LEDA*-Graphen gespeichert. Dieser Graphentyp bietet mehrere Vorteile; zum Beispiel ermöglicht *LEDA* die Berechnung kürzester Wege (ohne Nebenbedingung) durch im Paket mitgelieferte Algorithmen.

Die in den *Raw*-Dateien gespeicherten Informationen beziehen sich sehr genau auf das *CMCF*-Ausgangsproblem. Insbesondere werden den Kanten keine Kosten mitgeliefert, sondern drei Parameter, welche das ausführende Programm dann mittels einer *Link-Performance*-Funktion in Kantenkosten umwandelt.

Aus diesem Grunde ist es relativ schwierig, diese beiden Ansätze auf den gleichen Daten zu vergleichen. Es scheint nicht möglich, *Dat*-Dateien in *Raw*-Graphen zu verwandeln, da nicht bekannt ist, wie ein Kostenwert in die *Link-Performance*-Parameter umgewandelt werden kann. Umgekehrt können jedoch Instanzen während des *CMCF*-Programmablaufs als *Dat*-Datei gespeichert werden, womit zumindest ein teilweiser Vergleich ermöglicht wird.

Unsere neu erstellten Algorithmen lassen sich hingegen problemlos mit beiden Graphentypen benutzen, da wir die zu verwendende Graphenstruktur per *Traits*-Klasse vom eigentlichen Algorithmus getrennt haben.

4.1.4 Besonderheiten beim Aufruf des *CMCF*-Programms

Das dem *CMCF*-Projekt zugrunde liegende Problem und die genaue Funktionsweise des zur Lösung der Fragestellung verwendeten Algorithmus ist in diversen Veröffentlichungen (siehe z.B. [9], [10]) detailliert dargestellt. In dieser Arbeit soll deswegen nicht das Projekt an sich präsentiert werden; von Interesse ist vielmehr die erreichte Senkung des Aufwands für den Fall, dass ein alternativer Algorithmus bei der Ermittlung der doppelt beschränkten beziehungsweise ressourcenbeschränkten kürzesten Wege verwendet wird.

Aus diesem Grunde gehen wir an dieser Stelle nur kurz auf die wichtigsten Aspekte ein, die für die Nutzung unserer Algorithmen im *CMCF*-Projekt relevant sind. Das Hauptproblem betrachtet üblicherweise einige vorgegebene *Commodities*. Dies sind mit einer vorgegebenen Stärke versehene Verkehrsflüsse, die jeweils von einem Start- zu einem Zielknoten geleitet werden sollen. Es wird zunächst in einer Initialisierungsphase für jede Commodity ein zugehöriges ressourcenbeschränktes Kürzeste-Wege-Problem gelöst. Anschließend werden in jeder Hauptiteration des Problems die Kantenkosten c_e mehrmals neu berechnet. Für jede Commodity ist daraufhin jedes Mal ein neuer ressourcenbeschränkter Weg zu berechnen. Hierbei ändern sich die Anforderungen an die zu ermittelnden Lösungen je nachdem, welche Einstellungen der Nutzer vorgenommen hat. Soll auf jeder Kante eine Kapazitätsbegrenzung für den Fluss gelten, so reicht uns bei jedem Aufruf ein doppelt beschränkter Weg. Ansonsten müssen wir einen zulässigen kostenminimalen Weg suchen.

Ziel des Programms ist es, die Netzbelastung zu minimieren. Der verwendete Algorithmus konvergiert gegen den Minimalwert, verbessert den ermittelten Wert jedoch nach einiger Zeit nur noch sehr wenig. Deswegen besteht die Möglichkeit, den Algorithmus terminieren zu lassen, sobald die momentane Lösung um weniger als einem vorgegebenen Prozentsatz von der Optimallösung abweicht.

Des Weiteren kann der Nutzer den Algorithmus nach einer vorgegebenen Maximalanzahl an Hauptiterationen abbrechen zu lassen. Dies ist insbesondere für den Fall mit nur einer Commodity interessant, denn dann wird unser Algorithmus immer

nur einmal pro Iteration als Unterproblem aufgerufen. Somit ist es möglich, eine kleine Anzahl an Problemen auf immer demselben Graphen und denselben Start- und Zielknoten auszuführen. Die Ausgangsdaten für jede Iteration können dann als *LEDA*-Graph gespeichert werden, womit ein Vergleich mit der 2-Phasen-Methode ermöglicht wird.

Außerhalb des eigentlichen Algorithmus zur Ermittlung der ressourcenbeschränkten Wege werden als Preprocessing die kürzesten Wege ohne Nebenbedingung errechnet. Obwohl diese nicht bei der Lösung des Unterproblems durch unsere neu erstellten Algorithmen benutzt werden, haben wir sie zunächst nicht entfernt. Dies wäre zwar durchaus möglich, würde aber ein Umschreiben des Codes erfordern. Besser wäre es wahrscheinlich, das Preprocessing der neuen Algorithmen zu deaktivieren und die unteren Schranken dafür direkt im *CMCF*-Hauptprogramm zu ermitteln.

Aus dem eben genannten Grund sind unsere Testergebnisse leicht verfälscht. Die Laufzeiten der neuen Algorithmen beinhalten nämlich noch den Aufwand für das Preprocessing des alten Labeling-Ansatzes, sind also sogar noch etwas schneller als eigentlich angegeben.

4.2 Vergleiche

4.2.1 Tests auf *LEDA*-Graphen

Bis auf den *CMCF*-Ansatz sind alle unsere vorgestellten Algorithmen auf *LEDA*-Graphen benutzbar. Wir werden zunächst alle Varianten auf Zufallsgraphen testen und dann die erfolgsversprechendsten zur Lösung von Problemen verwenden, die im *CNOP*-Paket zu Testzwecken auf realistischeren Graphen mitgeliefert worden sind.

Zur Erstellung von *LEDA*-Zufallsgraphen steht uns ein Generator zur Verfügung. Nachdem die Anzahl der gewünschten Knoten angegeben sind, positioniert er diese nach dem Zufallsprinzip in einem 2-dimensionalen Raum. Anhand einer ebenfalls vom Nutzer einstellbaren Dichte berechnet er einen Radius r und fügt für jeden Knoten v Kanten zu allen Knoten, die höchstens um r von v entfernt sind, hinzu. Schließlich wird der Weg vom geographisch südwestlichsten zum nordöstlichsten Weg gesucht. Anhand dieser Problemstellung wird sichergestellt, dass eine möglichst große Zahl an Kanten betrachtet wird.

Wir betrachten zu zwei unterschiedlichen Knotenanzahlen jeweils einen Graphen mit kleiner und einen mit großer Dichte, um sowohl die Auswirkungen der Knoten- als auch der Kantenanzahl auf das Problem zu betrachten. Für jeden Graphen starten wir unsere Algorithmen mit verschiedenen oberen Ressourcenschranken; wir

Graph	Knoten	Kanten	Faktor	Ressourcenschranke	Lösung
gen_graph2a	10000	69454	150%	2788.5	$(785.971, 2142)$
			120%	2230.8	$(785.971, 2142)$
			110%	2044.9	$(787.242, 2042)$
			105%	1951.95	$(793.404, 1951)$
			99%	1840.41	-
gen_graph2b	10000	211594	150%	1135.5	$(713.227, 1090)$
			120%	908.4	$(715.566, 908)$
			110%	832.7	(723.475,832)
			105%	794.85	(733.03,794)
			99%	749.43	-
gen_graph3a	15000	113164	150%	3070.5	(784.975,2443)
			120%	2456.4	(784.975,2443)
			110%	2251.7	(793.466,2251)
			105%	2149.35	(805.343,2149)
			99%	2026.53	-
gen_graph3b	15000	326648	150%	1359	(704.862,1315)
			120%	1087.2	(707.69,1087)
			110%	996.6	(713.535,996)
			105%	951.3	(721.621,951)
			99%	896.94	-

Tabelle 4.2: Eigenschaften der *LEDA*-Zufallsgraphen. Der Faktor stellt die Abweichung in Prozent der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges dar.

setzen dabei die Schranken auf 150%, 120%, 110%, 105% bzw. 99% des ressourcengünstigsten *s*-*t*-Weges ohne Nebenbedingung. Für höhere Schranken sind unsere generierten Graphen nicht geeignet, da unser kostenminimaler Weg dann in den meisten Fällen einen zulässigen Ressourcenverbrauch hat und wir immer die gleiche Lösung bekommen. Indem wir auch den Fall betrachten, dass die Ressourcenschranke knapp kleiner als der Ressourcenverbrauch des ressourcengünstigsten Weges ist (nämlich 99% seines ermittelten Wertes), sind wir auch in der Lage, den bei einer solchen Situation praktizierten Aufwand zu erkennen. Dieses Ergebnis ist für viele Praxisstellungen relevant, unter anderem auch für unser *CMCF*-Projekt.

Wir lösen zunächst das *CSP*-Problem und ermitteln zusätzlich 10 Pareto-optimale Wege, falls unser Algorithmus dies ermöglicht. Der erweiterte Dijkstra-Algorithmus ohne Beschleunigung erweist sich hierbei erwartungsgemäß als ineffizient gegenüber den anderen Algorithmen. Es fällt allerdings auf, dass weitere Pfade für das *POCP*-Problem praktisch ohne zusätzlichen Aufwand bereitgestellt werden können. Dies liegt daran, dass der ermittelte lexikographisch kleinste Listeneintrag für diesen Algorithmus immer den Wegkosten eines real existierenden *s*-*v*-Weges entspricht. Ist also der erste *s*-*t*-Weg ermittelt, so werden die nächsten Pareto-optimalen Wege schon bald gefunden werden, da sie normalerweise kaum größere Kosten als der zuletzt gefundene Weg haben und damit zu einer der nächsten ermittelten Listeneinträge zählen dürften.

		gen_graph2a		gen_graph2b		gen_graph3a		gen_graph3b	
Ansatz	Faktor	CSP	POCP	CSP	POCP	CSP	POCP	CSP	POCP
Dijkstra	150%	137.9	138.0	517.9	517.9	553.6	553.7	1770.9	1771.1
	120%	147.9	147.9	542.5	542.7	589.2	589.3	1842.3	1842.6
	110%	141.8	141.8	489.9	490	566.5	566.7	1682.7	1682.9
	105%	138.1	138.1	453.5	453.6	546.4	546.5	1560	1560.2
	99%	134.8	134.8	414.8	414.8	526.2	526.2	1428.4	1428.4
zielge-richtet	150%	0.3	0.35	0.7	0.76	0.53	0.59	1.16	1.26
	120%	0.31	0.36	1.89	3.33	0.54	0.61	6.33	8.4
	110%	0.49	0.57	5.4	6.69	3.76	5	16.7	19.24
	105%	1.05	1.36	3.25	4.53	4.39	4.96	8.98	12.14
	99%	0.29	0.29	0.7	0.7	0.54	0.54	1.17	1.17

Tabelle 4.3: Laufzeiten in s auf den LEDA-Zufallsgraphen, Teil 1. POCP wurde mit k=10 getestet. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges.

Des Weiteren fällt auf, dass der Algorithmus fast genauso viel Zeit zur Ermittlung einer Lösung benötigt wie zur Erkenntnis, dass das Problem unzulässig ist. Dies ist nachvollziehbar, da wir in dieser Version reelle Weglängen betrachten und die Ressourcenschranke damit erst sehr spät eingreift: bei einer derart knapp gewählten Schranke wohl kurz vor dem Erreichen des Zielknotens.

Der erweiterte zielgerichtete Dijkstra-Algorithmus erweist sich für diese Testreihe als der schnellste Lösungsansatz. Es zeigt sich, dass die Transformation der Kantengewichte insbesondere für die Fälle, in denen kein zulässiger Weg existiert oder die Ressourcenschranke recht hoch angesetzt ist, zu sehr kurzen Laufzeiten führt. Gibt es hingegen viele Teilwege, die erst im weiteren Verlauf unzulässig werden, so nimmt die Laufzeit recht deutlich zu, bleibt jedoch unter den vergleichbaren Laufzeiten für die anderen auf dem Dijkstra-Algorithmus basierenden Lösungsansätze. Bei der Ermittlung von Pareto-optimalen Wegen ist im Gegensatz zum erweiterten Dijkstra-Algorithmus ohne Beschleunigung eine nennenswerte Erhöhung der Laufzeit erkennbar. Dies ist jedoch nachvollziehbar, da in diesem Fall die Listeneinträge eben nicht reellen Weglängen entsprechen und die auf einen gefundenen *s*-*t*-Weg folgenden Einträge durchaus Teilwegen entsprechen können, die noch sehr weit von einem vollständigen Weg entfernt sind.

Der bidirektionale Ansatz erzielt zwar eine Beschleunigung gegenüber dem erweiterten Dijkstra-Algorithmus, ist aber dennoch weit langsamer als die besten getesteten Algorithmen. Wie erwartet findet er mit weniger Aufwand einen Weg als der erweiterte Dijkstra-Ansatz, falls es ihn gibt, braucht aber länger, um festzustellen, dass kein zulässiger Weg existiert. In Kapitel 2 war für den Fall des Kürzeste-Wege-Problems gezeigt worden, dass die Anzahl der betrachteten Knoten im bidirektionalen Ansatz in etwa der Hälfte der zu bearbeitenden Knoten für den unidirektionalen Fall entsprechen. Was die Laufzeit angeht, so ergeben sich bei unseren Tests größere Unterschiede. Zum einen liegt das daran, dass im Fall der getesteten Zufallsgraphen der bidirektionale Ansatz nur noch ein Drittel der

Ansatz	Faktor	gen_graph2a	gen_graph2b	gen_graph3a	gen_graph3b
bidirektional für CSP	150%	25.4	159.99	101.25	427.04
	120%	26.35	167.45	104.64	444.15
	110%	26.85	180.58	112.54	472.6
	105%	27.23	192.86	123.1	498.19
	99%	768.69	2960.47	2516.48	≥ 7200
zielgerichtet & bidirektional für CSP	150%	0.53	1.27	0.97	2.06
	120%	0.52	5.21	0.98	17.32
	110%	0.85	9.21	8.13	25.94
	105%	1.71	6.8	9.31	15.13
	99%	0.56	1.34	1.15	2.2

Tabelle 4.4: Laufzeiten in s auf den LEDA-Zufallsgraphen, Teil 2. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges.

Anzahl an Knoten durchläuft, die im unidirektionalen Algorithmus abgearbeitet werden. Dies könnte auf mehrere Gründe zurückzuführen sein, zum Beispiel auf die spezielle Graphstruktur, die Beschleunigung durch ein gutes Abbruchkriterium oder das Auftreten der Ressourcenschranke. Außerdem sind aufgrund der gesunkenen Anzahl an zu betrachtenden Knoten sowohl die Operationen auf dem Heap für unmarkierte Listeneinträge als auch das Einfügen eines neuen Eintrags in die Liste eines Knotens weniger aufwändig, was den weiteren Beschleunigungseffekt erklärt.

Der bidirektional zielgerichtete Algorithmus ist zwar auch langsamer als der unidirektional zielgerichtete Ansatz, scheint aber der einzige unserer Labeling-Ansätze zu sein, der durchaus für gewisse Problemstellungen schneller eine Lösung finden könnte. Für die Fälle mit hoher oder 99%iger Ressourcenschranke benötigt er fast das Doppelte der Laufzeit des zielgerichteten Ansatzes. Dies bedeutet, dass für diese Ausgangssituationen die Algorithmen schon nach sehr wenigen Hauptschleifendurchläufen terminieren und der Aufwand insbesondere durch die Ermittlung der unteren Schranken entsteht, wobei dieser für bidirektionale Verfahren doppelt so hoch ist. Ist die obere Schranke jedoch so angelegt, dass die Algorithmen nicht sofort die Lösung finden, so werden zwar durch die Verbindung von Bidirektionalität und zielgerichtetem Suchen weiterhin langsamere Endlaufzeiten erzielt. Die in Tabelle (4.6) dargestellten Aufwände, die dieser Ansatz zur Lösung des *DB*-Problems benötigt, sind jedoch insbesondere bei den schwierigen Instanzen kleiner, womit klar ist, dass bei gewissen Graphen oder gar mit einem besseren Abbruchkriterium dieser Ansatz dem unidirektionalen Algorithmus überlegen sein dürfte.

Die 2-Phasen-Methode des *CNOP*-Projekts ist auf den Zufallsgraphen in den meisten Fällen dem zielgerichteten und teilweise sogar dem bidirektional zielgerichteten Ansatz unterlegen. Obwohl der Hüllenansatz der ersten Phase sehr gute Schranken für den minimalen Kostenwert liefert, fällt auf, dass der Algorithmus eine gewisse Grundlaufzeit zu benötigen scheint, die dann aber für jedes Problem un-

	Faktor	gen_graph2a	gen_graph2b	gen_graph3a	gen_graph3b
obere und untere Kostenschranken der 2-Phasen-Methode	150%	-	-	-	-
	120%	-	716.86/715.48	784.98/784.97	707.73/707.64
	110%	787.24/781.17	723.7/723.28	807.28/793.36	713.55/713.38
	105%	794.66/793.07	733.03/732.78	807.28/805.06	728.38/721.41
	99%	-	774.26/774.26	-	-
Laufzeiten in s für 2-Phasen-Methode mit Reduktionen	150%	1.17	2.82	1.9	4.52
	120%	1.17	7.37	5.3	12.65
	110%	3	6.98	4.93	10.68
	105%	2.48	4.86	3.42	8.09
	99%	0.91	2.24	1.43	3.58
Laufzeiten in s für 2-Phasen-Methode ohne Reduktionen	150%	0.46	0.93	0.65	1.35
	120%	0.46	5.81	4.2	9.86
	110%	2.77	6.73	3.77	9.77
	105%	2.61	5.72	3.78	9.01
	99%	1.34	5.21	2.3	4.1

Tabelle 4.5: Ergebnisse der 2-Phasen-Methode auf den LEDA-Zufallsgraphen. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. Sind keine Kostenschranken angegeben, so wurde die Lösung schon während der ersten Phase gefunden.

abhängig von der Ressourcenschranke in etwa gleich bleibt. Die kleineren Laufzeiten für besonders hohe oder niedrige obere Schranken folgen aus geschickten Abbruchkriterien, die dafür sorgen, dass der Algorithmus nicht in seiner Gesamtheit durchlaufen werden muss und schon während des Hüllenansatzes terminiert. Sie stellen also keinen Widerspruch zu unserer Aussage dar. Insgesamt scheint der Gesamtaufwand eher von der Größe des Eingabegraphen als von der gewählten Schranke abhängen. Deswegen ist zu erwarten, dass diese Methode bei komplexeren Problemen schneller eine Lösung finden wird, wie es für den generierten Graphen *gen_graph3b* teilweise schon der Fall ist.

Die Frage, ob der Einsatz von Reduktionen für diesen Algorithmus vorteilhaft ist, lässt sich nach dieser Testreihe nicht genau beantworten. Zwar sorgen diese Reduktionen offensichtlich für einen Mehraufwand, jedoch scheint sich dieser zumindest bei den schwierigeren Problemen auszuzahlen. So werden bei einer 105%-igen Schranke die Lösungen durchweg schneller ermittelt.

Wir untersuchen nun die Algorithmen für das *DB*-Problem auf den Zufallsgraphen. Wir betrachten die bidirektionalen sowie bidirektional zielgerichteten Ansätze, die bei der ersten ermittelten Lösung abbrechen, und die zielgerichteten bzw. bidirektional zielgerichteten Ansätze, welche um das zusätzliche Abbruchkriterium erweitert wurden. Wie erwartet liefern die ersten beiden Ansätze qualitativ gute Ansätze mit relativ hoher Laufzeit, während das zusätzliche Abbruchkriterium für Lösungen mit hohem Kostenwert sorgen, diese jedoch sehr schnell berechnen.

Wie schon bei den Algorithmen für das *CSP*-Problem erweist sich der reine bidirektionale Ansatz als schwächste aller Beschleunigungsmethoden, obwohl durch

Ansatz	Faktor	gen_graph2a	gen_graph2b	gen_graph3a	gen_graph3b
bidirektional	150%	23.75 (786,2142)	158.87 (713.5,1077)	98.34 (786.1,2456)	423.42 (706.3,1239)
	120%	24.57 (786,2142)	166.2 (718.7,906)	101 (786.1,2456)	438.92 (707.7,1087)
	110%	24.52 (787.8,2044)	178.21 (727.7,832)	108.15 (794.2,2251)	451.44 (713.8,996)
	105%	25.11 (793.7,1951)	191.71 (735.1,794)	117.9 (806.1,2149)	489.81 (723.7,951)
	99%	25.39 -	170.23 -	105.7 -	467.55 -
bidirektional & zielgerichtet	150%	0.52 (786,2142)	1.22 (713.2,1090)	0.96 (784.9,2443)	2.06 (704.8,1315)
	120%	0.52 (786,2142)	2.9 (715.6,908)	0.94 (784.9,2443)	10.91 (714.1,1087)
	110%	0.71 (787.6,2044)	4.21 (724.2,832)	4.17 (793.9,2251)	12.49 (714.7,696)
	105%	1.06 (793.4,1951)	3.29 (733.6,794)	4.18 (812.9,2149)	7.7 (722.4,951)
	99%	0.51 -	1.24 -	1.03 -	2.13 -

Tabelle 4.6: Laufzeiten in s und ermittelte Lösungen für das DB-Problem auf den LEDA-Zufallsgraphen, Teil 1. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges.

die Einführung einer oberen Kostenschranke die Laufzeit für den Fall, dass es keine zulässige Lösung gibt, erheblich sinkt. Die ermittelten Lösungen sind zwar nahe an denen der kostenminimalen ressourcenbeschränkten Wege, die Verbindung des bidirektionalen mit dem zielgerichteten Ansatz sorgt jedoch für ähnliche Lösungen mit viel besseren Laufzeiten. Diese sind zwar um einiges langsamer als die mit dem zusätzlichen Kriterium erzielten Zeiten, jedoch lässt sich aufgrund der besseren Lösungen zunächst noch nicht sagen, welche Methoden zu bevorzugen sind.

Bei den Algorithmen mit zusätzlichem Abbruchkriterium fällt auf, dass sich bei relativ simplen Graphen die Laufzeit genau genommen auf das Preprocessing zur Ermittlung der unteren Schranken beschränkt. Aus diesem Grunde benötigt die Verbindung des zielgerichteten mit dem bidirektionalen Ansatz eine doppelt so hohe Zeit wie der unidirektionale Algorithmus, da sie auch die unteren Schranken zum Startknoten berechnen muss. Da wir zu diesem Zeitpunkt noch nicht wissen, wie sich die beiden Algorithmen bei schwierigeren Problemen verhalten, ist es auch hier noch nicht möglich zu sagen, welcher Ansatz der bessere ist.

Abschließend kommen wir nun zu denen im *CNOP*-Paket bereitgestellten Graphen. Es handelt sich hierbei um einen Straßengraphen unbekannter Herrkunft sowie um einen schottischen Straßengraphen, welche jeweils als kleine und große Instanz existieren. Die obere Ressourcenschranke ist fest vorgegeben, wobei nicht

Ansatz	Faktor	gen_graph2a	gen_graph2b	gen_graph3a	gen_graph3b
bidirektional & zielgerichtet mit zusätzlichem Abbruchkriterium	150%	0.56 (859.9,1859)	1.3 (777.7,765)	1.02 (848.2,2057)	2.18 (747.5,916)
	120%	0.56 (859.9,1859)	1.29 (777.7,765)	1.01 (848.2,2057)	2.2 (747.5,916)
	110%	0.55 (859.9,1859)	1.28 (777.7,765)	1.02 (848.2,2057)	2.19 (747.5,916)
	105%	0.59 (825.13,1924)	1.35 (746.58, 794)	1.28 (823.69, 2149)	2.18 (739.43,948)
	99%	0.57 -	1.29 -	1.04 -	2.18 -
zielgerichtet mit zusätzlichem Abbruchkriterium	150%	0.29 (859.9,1859)	0.72 (777.7,765)	0.54 (848.3,2057)	1.18 (747.5,916)
	120%	0.28 (859.9,1859)	0.72 (777.7,765)	0.53 (848.3,2057)	1.18 (747.5,916)
	110%	0.29 (859.9,1859)	0.72 (777.7,765)	0.54 (848.3,2057)	1.19 (747.5,916)
	105%	0.34 (813.3,1951)	0.76 (746.6,794)	0.59 (823.7,2149)	1.19 (739.4,948)
	99%	0.3 -	0.72 -	0.59 -	1.19 -

Tabelle 4.7: Laufzeiten in s und ermittelte Lösungen für das DB-Problem auf den LEDA-Zufallsgraphen, Teil 2. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges.

bekannt ist, nach welchem Kriterium sie ausgewählt worden ist. Insgesamt scheint es sich bei den zu suchenden Wegen auf den *road_graph*-Graphen um relativ einfache Problemstellungen zu handeln. Dies gilt insbesondere für die kleinere Version, bei der der ressourcengünstigste Weg die Lösung ist.

Graph	Knoten	Kanten	UBR	LBR	Lösung
road_graph_small	24086	50826	816.987	0	$(875.846, 0)$
road_graph_big	77059	171536	1324.96	0	(1281.03,1280)
scotland_small	16384	65024	2810.5	2093	(2966,2809)
scotland_big	63360	252432	5897.1	5361	(5848,5897)

Tabelle 4.8: Eigenschaften der *LEDA*-Beispielgraphen des CNOP-Projekts. *UBR* ist die vorgegebene Ressourcenschranke, *LBR* der Verbrauch des ressourcengünstigsten Weges.

Wir testen diesmal nur die erfolgsversprechendsten Algorithmen für das *CSP*-Problem. Hierbei scheinen sich nun einige Eigenschaften zu bestätigen. Unsere Labeling-Ansätze sind bei der Ermittlung der einfachen *road_graph*-Lösungen schneller als die 2-Phasen-Methode, sind jedoch auf den *scotland*-Graphen weit unterlegen. Hingegen ergeben für diese Problemstellungen weder der Einsatz der Bidirektionalität für den zielgerichteten Ansatz noch das Ausführen der Reduktionen bei der 2-Phasen-Methode eine Verbesserung der Laufzeit.

Ansatz	road_graph_small	road_graph_big	scotland_small	scotland_big
zielgerichtet	0.45	3.53	33.01	> 3600
zg + bidir	0.88	6.06	102.55	> 3600
CSP mit Red.	1.11	13.75	2.69	17.09
CSP ohne Red.	2.25	9.81	2.69	13.09

Tabelle 4.9: Laufzeiten in s auf den CNOP-Beispielgraphen

4.2.2 Tests auf *Raw*-Graphen

Da wir die 2-Phasen-Methode des *CNOP*-Paketes nicht direkt in das *CMCF*-Projekt eingebaut haben, starten wir zunächst einige Testprobleme auf *CMCF* ohne Kapazitätsbeschränkung mit nur einer Commodity und der Einschränkung, dass sie nach zehn Iterationen abbrechen; wir erstellen für jede dieser Iterationen einen zugehörigen *LEDA*-Graphen. Somit können wir den zu erwartenden Aufwand für die 2-Phasen-Methode abschätzen, und es lässt sich erkennen, wie gut alle vorgestellten Algorithmen auf demselben Problem abschneiden.

Graph	Knoten	Kanten	Faktor	Flussstärke
berlin3	6690	10937	110%	2300
berlin5	8891	14468	110%	1850
berlin7	12100	19570	110%	2300

Tabelle 4.10: Ausgangsdaten für unsere erste Testreihe auf dem *CMCF*-Problem. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. Es wird eine vorgegebene Flussstärke auf einen Weg geschickt.

Wir untersuchen Problemstellungen auf drei verschiedenen Teilgraphen des Berliner Verkehrsnetzes. Diese unterscheiden sich zwar bezüglich der Knoten- und Kantengröße nicht allzu sehr, jedoch ist die Größe des Ausgangsgraphen in diesem Falle nicht mehr so relevant wie bei den *LEDA*-Graphen, da hier auch andere Parameter wie der Gesamtverkehrsfluss die Schwierigkeit der zu lösenden *CSP*-Probleme beeinflusst. Insgesamt sind die entstehenden *CSP*-Unterprobleme des ersten Graphes sehr schnell zu lösen, die des zweiten schon etwas schwerer, und die des dritten lassen sich nur mit größerem Aufwand ermitteln.

Algorithmus	berlin3	berlin5	berlin7
Dijkstra	16	118	1775
zielgerichtet	2	14	126
bidirektional	7	163	429
bidir-zg	2	22	65
CMCF-Labeling	10	83	534
2-Phasen mit Red.	7	16	20
2-Phasen ohne Red.	7	17	21

Tabelle 4.11: Laufzeiten in s für 10 Iterationen auf den Testgraphen

Wir geben aus Gründen der Übersichtlichkeit nicht die einzelnen Laufzeiten an, die bei der jeweiligen Ermittlung der ressourcenbeschränkten Wege entstanden sind, sondern beschränken uns auf die Gesamtlaufzeit, welche hauptsächlich bei den 11 Aufrufen (Initialisierung plus zehn Hauptiterationen) des *CSP*-Problems entsteht. Bei der 2-Phasen-Methode werden hierbei Schätzungen geliefert, da wir nur den Aufwand zur Lösung der Unterprobleme berechnen können.

Auf dem leichtesten Problem entsprechen die Ergebnisse in etwa denen, die bei der Untersuchung auf *LEDA*-Graphen beobachtet wurden. Der erweiterte Dijkstra-Algorithmus ohne Beschleunigung ist weiterhin der langsamste Ansatz. Das bidirektionale Verfahren ist mehr als doppelt so schnell, bleibt allerdings den zielgerichteten Ansätzen deutlich unterlegen. Hierbei erreichen der unidirektionale und der bidirektionale Ansatz die gleiche Gesamtlaufzeit. Die 2-Phasen-Methode benötigt bei einer derart einfachen Problemstellung erneut mehr Zeit als die schnellsten Labeling-Methoden und scheint für solche Fälle nicht geeignet zu sein.

Erstmals ist es nun möglich, die Geschwindigkeit des bisher im *CMCF*-Projekt verwendeten Labeling-Ansatzes mit den neuen Verfahren zu vergleichen. Auf dem kleinsten Graphen ist dieser Algorithmus nur dem erweiterten Dijkstra-Ansatz ohne Beschleunigung überlegen; selbst der bidirektionale Ansatz erscheint leicht schneller zu sein. Auf unseren drei Problemen stellt der zielgerichtete Ansatz gar eine vier- bis sechsfache Beschleunigung zum bisher verwendeten Algorithmus dar.

Betrachten wir unser Problem auf dem *berlin*5-Graphen, so fällt insbesondere auf, dass der rein bidirektionale Ansatz hier nur sehr langsam arbeitet und selbst der erweiterte Dijkstra-Algorithmus ohne Beschleunigung schneller ist. Dies ist jedoch wahrscheinlich auf die spezielle Problemstruktur zurückzuführen, denn auf dem *berlin*7-Graphen stehen diese beiden Ansätze wieder im bewährten Verhältnis zueinander. Die 2-Phasen-Methode ist inzwischen praktisch genauso schnell wie die zielgerichteten Verfahren, sie ermittelt die gewünschten Wege bei den letzten, schwierigeren Hauptiterationen sogar eher, woraus wir schließen könen, dass sie wohl bei einer größeren Anzahl an Iterationen den kleinsten Aufwand erzeugen würde. Der Einsatz von Reduktionen sorgt hierbei für eine zusätzliche Beschleunigung.

Auf dem *berlin*7-Graphen sind dann die 2-Phasen-Methoden endgültig die schnellsten Verfahren. Ihre Laufzeit erhöht sich nur unmerklich und scheint eher durch die Größe des Ausgangsgraphen als durch die Schwierigkeit des Problems beeinflusst. Es zeigt sich erneut, dass dieser Ansatz bei aufwändigeren Fragestellungen den Labeling-Verfahren überlegen ist. Die zielgerichteten Methoden zeichnen sich dabei erneut als die schnellsten auf dem Dijkstra-Algorithmus basierenden Ansätze aus. In diesem Fall ist der bidirektionale Algorithmus sogar schneller als der unidirektionale. Ob dies generell für kompliziertere Probleme gilt oder nur auf diesen Beispielgraphen zutrifft, wird bei den nächsten Tests zu untersuchen sein.

Instanz	Graph	Knoten	Kanten	Commodities	Flussstärken
bs24	berlin2	1616	2476	1000	0.1-28
bs45	berlin4	4638	7631	2000	0.1-28
bs64	berlin6	10491	17038	2500	1-240

Tabelle 4.12: Ausgangsdaten für unsere ausführlicheren Tests auf dem *CMCF*-Problem

Als Nächstes untersuchen wir nun Problemstellungen, bei denen das Hauptprogramm erst terminiert, wenn die zuletzt ermittelte Netzbelastung nur noch um einen vorgegebenen Prozentsatz von der optimalen Belastung abweicht. Dabei verzichten wir auf die Algorithmen, die bei den bisherigen Ergebnissen sich als offensichtlich zu langsam herausgestellt haben. Da die 2-Phasen-Methode bei einer derart großen Zahl von Aufrufen unseres Unterproblems nicht mehr zum Vergleich herangezogen werden kann, reduzieren sich unsere Tests für den Fall ohne Kapazitätsbeschränkungen auf das bisher verwendete Labeling-Verfahren und unsere zielgerichteten Ansätze.

Algorithmus	Faktor	Abbruch	CSP-Aufrufe	Wert	Laufzeit
CMCF-Labeling ohne Kapazitäts-beschränkung	120%	1.5%	6000	3453475.1	27
		1%	6000	3453475.1	27
		0.5%	8000	3435066.7	35
	110%	1.5%	5000	3470737.2	23
		1%	6000	3460840.7	27
		0.5%	9000	3452535.9	40
zielgerichtet ohne Kapazitäts-beschränkung	120%	1.5%	6000	3453475.1	29
		1%	6000	3453475.1	30
		0.5%	8000	3435066.7	39
	110%	1.5%	5000	3470737.2	24
		1%	6000	3460840.7	28
		0.5%	9000	3452535.9	43
zielgerichtet & bidirektional ohne Kapazitäts-beschränkung	120%	1.5%	6000	3453475.1	52
		1%	6000	3453475.1	52
		0.5%	8000	3435066.7	69
	110%	1.5%	5000	3470737.2	43
		1%	6000	3460840.7	53
		0.5%	9000	3452535.9	78

Tabelle 4.13: Ergebnisse der Tests auf der *bs*24 Instanz. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. *Abbruch* ist die maximale Abweichung, die der momentane Netzbelastungswert haben darf, wenn das Hauptprogramm abbricht. *Wert* ist der Netzbelastungswert zum Zeitpunkt des Abbruchs. *CSP-Aufrufe* ist die Anzahl der Aufrufe des Unterprogrammes. Die Laufzeiten sind in s.

Wir betrachten zunächst drei Instanzen, die bei vorangegangenen Experimenten für das *CMCF*-Problem verwendet wurden und untersuchen den Fall, in dem die Kantenkapazitäten ignoriert werden [9]. Es handelt sich dabei um Probleme, die tausen-

de Commodities mit relativ kleinen Flussstärken betrachten. Wie wir an den ermittelten Ergebnissen erkennen können, sind die jeweiligen ressourcenbeschränkten Wege sehr einfach zu finden. Es werden teilweise hunderte von ihnen pro Sekunde ermittelt. Dabei liefert der bisherige Labeling-Ansatz scheinbar sehr gute Ergebnisse und ist meistens sogar schneller als die zielgerichteten Algorithmen. Dieses Ergebnis ist jedoch zu relativieren. Zum Einen wird wie gesagt immer das Preprocessing des ehemaligen Ansatzes ausgeführt, selbst wenn ein neuer Algorithmus benutzt wird. Schwerwiegender ist jedoch die Tatsache, dass unsere zielgerichteten Verfahren bei jedem Aufruf die ressourcengünstigsten Wege zum Zielknoten neu errechnen. Wie wir gesehen haben, ändern sich die Ressourcenwerte jedoch nie; somit würde eine einmalige Ermittlung der Wege genügen. Angesichts dieser Tatsachen dürfte zumindest der rein zielgerichtete Ansatz bei Anpassung an die neue Problemstellung eine Beschleunigung darstellen. Ein weiterer Grund für die guten Resultate des *CMCF*-Labeling-Ansatzes liegt darin, dass er darauf ausgerichtet ist, schnell zu terminieren, wenn die Probleme derart einfach sind. Bei komplexeren Problemstellungen ist er, wie bereits gesehen, unterlegen.

Algorithmus	Faktor	Abbruch	CSP-Aufrufe	Wert	Laufzeit
CMCF-Labeling ohne Kapazitäts-beschränkung	120%	1.5%	8000	6542602.7	104
		1%	10000	6516659.6	129
		0.5%	14000	6493119	188
	110%	1.5%	8000	6530006.9	111
		1%	8000	6530006.9	110
		0.5%	8000	6530006.9	111
zielgerichtet ohne Kapazitäts-beschränkung	120%	1.5%	8000	6542192.8	143
		1%	10000	6516091.6	186
		0.5%	14000	6492849.4	254
	110%	1.5%	8000	6529759.4	144
		1%	8000	6529759.4	143
		0.5%	8000	6529759.4	144
zielgerichtet & bidirektional ohne Kapazitäts-beschränkung	120%	1.5%	8000	6542192.8	234
		1%	10000	6516091.6	306
		0.5%	14000	6492849.4	411
	110%	1.5%	8000	6529759.4	242
		1%	8000	6529759.4	234
		0.5%	8000	6529759.4	235

Tabelle 4.14: Ergebnisse der Tests auf der *bs*45 Instanz. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. *Abbruch* ist die maximale Abweichung, die der momentane Netzbelastungswert haben darf, wenn das Hauptprogramm abbricht. *Wert* ist der Netzbelastungswert zum Zeitpunkt des Abbruchs. *CSP-Aufrufe* ist die Anzahl der Aufrufe des Unterprogrammes. Die Laufzeiten sind in s.

Des Weiteren fällt auf, dass das *CMCF*-Problem je nach verwendetem Algorithmus bei unterschiedlichen Gesamtlösungswerten abbricht. Dies liegt erneut an den auftretenden Rechnerungenauigkeiten. Durch das Skalieren stimmen in weniger als 1% der Fälle die von den zielgerichteten Verfahren ermittelten Wege nicht mit

denen überein, die vom *CMCF*-Labeling-Ansatz geliefert werden. Obwohl ihre Kostenwerte bis auf mehrere Stellen hinter dem Komma gleich sind, hat die Auswahl einer anderen Wegroute signifikante Auswirkungen auf das Hauptproblem, da der Fluss nunmehr über einen anderen Weg geleitet wird. Dies erklärt die teilweise recht großen Abweichungen zwischen den Ergebnissen.

Algorithmus	Faktor	Abbruch	CSP-Aufrufe	Wert	Laufzeit
CMCF-Labeling ohne Kapazitäts-beschränkung	120%	1.5%	20000	10298822	75
		1%	22500	10282162.1	85
		0.5%	32500	10258667.9	122
	110%	1.5%	15000	10421843.7	70
		1%	22500	10421843.7	106
		0.5%	32500	10367924	151
zielgerichtet ohne Kapazitäts-beschränkung	120%	1.5%	20000	10298822	88
		1%	22500	10282162.1	100
		0.5%	32500	10258668	143
	110%	1.5%	15000	10421842.7	68
		1%	22500	10367925.6	100
		0.5%	32500	10351453.5	145
zielgerichtet & bidirektional ohne Kapazitäts-beschränkung	120%	1.5%	20000	10298822	178
		1%	22500	10282162.1	203
		0.5%	32500	10258668	290
	110%	1.5%	15000	10421842.7	138
		1%	22500	10367925.6	204
		0.5%	32500	10351453.5	293

Tabelle 4.15: Ergebnisse der Tests auf der *bs*64 Instanz. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. *Abbruch* ist die maximale Abweichung, die der momentane Netzbelastungswert haben darf, wenn das Hauptprogramm abbricht. *Wert* ist der Netzbelastungswert zum Zeitpunkt des Abbruchs. *CSP-Aufrufe* ist die Anzahl der Aufrufe des Unterprogrammes. Die Laufzeiten sind in s.

Da wir für das Hauptproblem nur einen Wert suchen, der um einen vorgegebenen Prozentsatz vom eigentlichen Optimalwert abweicht, lässt sich auch nur schwer sagen, inwiefern diese Ungenauigkeiten Auswirkungen auf die Korrektheit des Ergebnisses haben. Es ist jedoch offensichtlich, dass ein Nutzer, der frühere Ergebnisse des Hauptproblems noch einmal durchgehen möchte, besser den alten Labeling-Ansatz verwenden sollte, um zu den gleichen Werten zu kommen.

Instanz	Graph	Knoten	Kanten	Commodities	Flussstärken
bs4new	berlin4	4618	7631	8	2500
bs5new	berlin5	8491	14468	5	2000-3000
bs6new	berlin6	10491	17038	5	2500
bs7new	berlin7	12100	19570	3	2000

Tabelle 4.16: Ausgangsdaten für weitere ausführlichere Tests auf dem *CMCF*-Problem. Aller Aufrufe werden ohne Kapazitätsbeschränkungen ausgeführt.

Da die Commodities in den vorangegangenen Beispielen nur kleine Flussstärken hatten, sind die daraus resultierenden Unterprobleme zu einfach geworden. Deswegen haben wir noch einige weitere Probleminstanzen erstellt. Diese zeichnen sich dadurch aus, dass sie zwar nur wenige Commodities beinhalten, dafür aber sehr viel Fluss auf jeden Weg geschickt wird. Somit benötigt jede Ermittlung eines ressourcenbeschränkten Weges mehr Zeit.

Algorithmus	Faktor	Abbruch	CSP-Aufrufe	Wert	Laufzeit
CMCF-Labeling	120%	2%	296	378535942.9	289
	110%	2%	400	401293154.8	370
zielgerichtet	120%	2%	296	378535942.9	43
	110%	2%	400	401293154.8	71
zielgerichtet & bidirektional	120%	2%	296	378535942.9	149
	110%	2%	400	401293154.8	155

Tabelle 4.17: Ergebnisse der Tests auf der *bs*4*new* Instanz. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. *Abbruch* ist die maximale Abweichung, die der momentane Netzbelastungswert haben darf, wenn das Hauptprogramm abbricht. *Wert* ist der Netzbelastungswert zum Zeitpunkt des Abbruchs. *CSP-Aufrufe* ist die Anzahl der Aufrufe des Unterprogrammes. Die Laufzeiten sind in s.

Für die Instanz *bs*4*new* benötigt das bisherige Verfahren zum Beispiel fast eine Sekunde pro Aufruf des Labeling-Ansatzes. Die zielgerichteten Ansätze sind vielfach schneller, und insbesondere die unidirektionale Methode senkt die Gesamtlaufzeit extrem. Zudem gibt es in diesem Fall auch kein Auftreten von Rechnerungenauigkeiten.

Algorithmus	Faktor	Abbruch	CSP-Aufrufe	Wert	Laufzeit
CMCF-Labeling	120%	2%	190	187206857	1474
	110%	2%	100	374672199.4	551
zielgerichtet	120%	2%	190	187206857.1	456
	110%	2%	100	374672199.1	128
zielgerichtet & bidirektional	120%	2%	190	187206857.1	718
	110%	2%	100	374672199.1	171

Tabelle 4.18: Ergebnisse der Tests auf der *bs*5*new* Instanz. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. *Abbruch* ist die maximale Abweichung, die der momentane Netzbelastungswert haben darf, wenn das Hauptprogramm abbricht. *Wert* ist der Netzbelastungswert zum Zeitpunkt des Abbruchs. *CSP-Aufrufe* ist die Anzahl der Aufrufe des Unterprogrammes. Die Laufzeiten sind in s.

Das gleiche Bild bietet sich bei Betrachtung der Ergebnisse für die *bs*5*new*-Instanz. Der rein zielgerichtete Ansatz sorgt hier für eine drei- bis vierfache Verbesserung des Aufwandes. Der bidirektional zielgerichtete Ansatz ist auch hier nicht ganz so schnell, wenngleich auch er schneller terminiert als die bisherige Methode.

Algorithmus	Faktor	Abbruch	CSP-Aufrufe	Wert	Laufzeit
CMCF-Labeling	120%	5%	270	24756136.9	2721
	110%	5%	460	28978248.4	5351
zielgerichtet	120%	5%	270	24756128.2	945
	110%	5%	495	28945863.6	2546
zielgerichtet & bidirektional	120%	5%	270	24756128.2	1343
	110%	5%	495	28945863.6	851

Tabelle 4.19: Ergebnisse der Tests auf der *bs*6*new* Instanz. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. *Abbruch* ist die maximale Abweichung, die der momentane Netzbelastungswert haben darf, wenn das Hauptprogramm abbricht. *Wert* ist der Netzbelastungswert zum Zeitpunkt des Abbruchs. *CSP-Aufrufe* ist die Anzahl der Aufrufe des Unterprogrammes. Die Laufzeiten sind in s.

Die *bs*6*new*-Instanz läuft auf dem gleichen Graphen, auf dem bei unseren ersten Vergleichen für das *CMCF*-Programm der bidirektional zielgerichtete Algorithmus dem rein zielgerichteten überlegen war. Offensichtlich kommt ihm die Graphenstruktur zugute, denn auch dieses Mal liefert er gute Ergebnisse. Im Gegensatz zu den anderen beiden Methoden ermittelt er die Lösung sogar schneller, wenn die Ressourcenschranke gesenkt wird. Somit ist er in dem Fall praktisch dreimal so schnell wie der unidirektionale Ansatz. Liegt die Schranke bei 120% des ressourcengünstigsten Weges, so ist er allerdings weiterhin langsamer.

Algorithmus	Faktor	Abbruch	CSP-Aufrufe	Wert	Laufzeit
CMCF-Labeling	120%	1.5%	231	11365539.3	740
	110%	1.5%	288	11800927.8	6338
zielgerichtet	120%	1.5%	246	11362856.8	127
	110%	1.5%	309	11798484.1	1863
zielgerichtet & bidirektional	120%	1.5%	246	11362856.8	349
	110%	1.5%	309	11798484.1	1088

Tabelle 4.20: Ergebnisse der Tests auf der *bs*7*new* Instanz. Der Faktor entspricht der prozentualen Abweichung der Ressourcenschranke vom Verbrauch des ressourcengünstigsten Weges. *Abbruch* ist die maximale Abweichung, die der momentane Netzbelastungswert haben darf, wenn das Hauptprogramm abbricht. *Wert* ist der Netzbelastungswert zum Zeitpunkt des Abbruchs. *CSP-Aufrufe* ist die Anzahl der Aufrufe des Unterprogrammes. Die Laufzeiten sind in s.

Diese Beobachtungen wiederholen sich auch bei den Tests für *bs*7*new*. Ist zunächst der zielgerichtete Ansatz schneller, so wird bei Senkung der oberen Ressourcenschranke der bidirektional zielgerichtete Algorithmus immer besser. Der bisherige Labeling-Ansatz ist bei all diesen Vergleichen weiterhin um ein Vielfaches langsamer.

Trotz der teilweise überzeugenden Ergebnisse für den bidirektional zielgerichteten Ansatz bedeutet das nicht, dass dieser bei schwierigen Problemen automatisch

schneller ist. Genau genommen ist die unidirektionale Methode bei den letzten zwei Instanzen selbst für eine Ressourcenschranke von 110% für die meisten Commodities schneller. Jedoch tritt in beiden Fällen ein zu berechnender Weg auf, der mit dem bidirektionalen Verfahren viel besser zu lösen ist. Da wir nur eine einstellige Anzahl an Commodities gewählt haben, wirkt sich ein solcher Fall besonders stark auf die Laufzeit aus. Es ist zu erwarten, dass bei einer größeren Auswahl an Commodities die Gesamtlaufzeit nicht mehr so stark durch einzelne Wege beeinflusst wird. Somit könnte der rein zielgerichtete Algorithmus wiederum einen ähnlichen Aufwand wie der bidirektional zielgrichtete aufweisen.

Nachdem wir das Verhalten unserer Algorithmen für den Fall des *CMCF*-Problems ohne Kantenkapazitäten untersucht haben, wäre es nun von besonderem Interesse, den Fall mit Kapazitätsbeschränkungen zu betrachten. Somit könnten wir nicht nur die Eignung unserer das doppelt beschränkte Problem lösenden Algorithmen für dieses Projekt testen, sondern wir wären auch in der Lage darzustellen, welche der beiden *CMCF*-Aufrufvarianten am meisten von unseren Beschleunigungsmethoden profitieren.

Bei der Durchführung unserer Tests stellte sich jedoch heraus, dass das *CMCF*-Programm beim Beachten der Kapazitäten nicht mehr terminierte. Dies lag wahrscheinlich an einer fehlerhaften Einbindung von Softwarepaketen, die der Hauptalgorithmus benötigt. Der Fehler war bis zum Einreichen dieser Arbeit noch nicht behoben worden, womit keine Tests durchgeführt werden konnten. Diese sollten allerdings nachgeholt werden, sobald das Programm wieder einwandfrei funktioniert.

Während der Implementation der Algorithmen wurden einige nicht weiter dokumentierte Tests auf dem damals noch einwandfrei laufenden *CMCF*-Programm durchgeführt. Insbesondere der zielgerichtete Ansatz mit zusätzlichem Abbruchkriterium sorgte hierbei für eine erhebliche Verringerung des Gesamtaufwandes. Es ist also zu erwarten, dass auch in diesem Fall eine Verbesserung der Laufzeit auftreten wird.

Kapitel 5

Zusammenfassung

Ein System, welches ein Fahrzeug möglichst schnell durch ein Netzwerk von seinem Start- zu seinem Zielort leiten soll, kann einen solchen Weg effizient berechnen. Es benutzt hierzu einen Algorithmus, der das Kürzeste-Wege-Problem löst, und erhält auf diese Weise in polynomialer Zeit eine Lösung.

Benutzen jedoch eine Vielzahl an Fahrzeugen ein solches System, so erfordert die Problemstellung, dass der Weg nicht nur schnell ist, sondern mehrere Eigenschaften gleichzeitig erfüllt. Ein solcher kürzester Weg mit Nebenbedingung ist ungleich schwerer zu lösen; er gehört zur Klasse der *schwach NP-vollständigen* Probleme.

Wir haben ausgehend von einem Algorithmus für das klassische Kürzeste-Wege-Problem Methoden entwickelt, die Wege mit Nebenbedingung durch eine Vielzahl an Beschleunigungsansätzen mit möglichst wenig Aufwand finden. Außerdem haben wir einen neu entwickelten Ansatz dargestellt, der an der Universität des Saarlandes entstanden ist und ebenfalls sehr schnell Resultate liefert.

Schließlich haben wir unsere Algorithmen sowohl direkt auf einzelnen Graphen als auch als wiederholt aufgerufenes Unterproblem eines Verkehrsnavigationssystems getestet.

5.1 Interpretation der Ergebnisse

Eine simple Erweiterung des Dijkstra-Algorithmus, der das Kürzeste-Wege-Problem effizent löst, auf unsere neue Fragestellung sorgt für einen Algorithmus, der sehr hohe Laufzeiten aufweist. Alle getesteten Beschleunigungsmethoden sorgen für signifikante Verbesserungen. Trotzdem sind sie nicht alle für den weiteren Gebrauch zu empfehlen. Der rein bidirektionale Ansatz benötigt für den Fall, dass kein zulässiger Weg existiert, eine noch größere Laufzeit als der erweiterte Dijkstra-Algorithmus. Existiert hingegen ein Weg, so ist der bidirektionale Ansatz

dennoch weit langsamer als die zielgerichteten Methoden. Dabei erweist sich der unidirektional zielgerichtete Ansatz als derjenige, der konstant die besten Resultate liefert. Eine Verbindung von zielgerichtetem und bidirektionalem Ansatz liefert zwar vereinzelt noch schneller Lösungen, benötigt aber besonders aufgrund eines schlechten Abbruchkriteriums für die meisten Problemstellungen noch zuviel Zeit.

Insgesamt stellen beide zielgerichteten Ansätze eine signifikante Verbesserung zum bisher bei einem Straßennavigationssystem der *TU Berlin* verwendeten Algorithmus dar. Ihr Einsatz im entsprechenden Unterprogramm würde die Gesamtlaufzeit des Pakets spürbar senken.

Die an der Universität des Saarlandes entwickelte 2-Phasen-Methode ist insbesondere bei schwierigen Instanzen, die eine große Laufzeit erwarten lassen, unseren Algorithmen weit überlegen. Bei kleinen Problemen benötigt sie jedoch länger, da sie etwas komplizierter aufgebaut ist.

Suchen wir nur einen doppelt beschränkten Weg, der sowohl bezüglich seiner Kosten als auch der Nebenbedingung unter vorgegebenen Schranken bleibt, so scheint nach den bisher durchgeführten Vergleichen der zielgerichtete Ansatz mit zusätzlichem Abbruchkriterium am geeignesten für den Einsatz im Navigationssystem der *TU Berlin* zu sein. Obwohl seine ermittelten Lösungen Kostenwerte aufweisen, die mitunter recht weit vom Optimalwert abweichen, ist er dermaßen schnell, dass er die Gesamtlaufzeit am meisten senkt.

5.2 Empfehlung für das Route-Guidance-Projekt an der TU Berlin

Der bisher verwendete Ansatz liefert Wege mit akzeptablem Aufwand. Aufgrund der durch die Rechnerarchitektur bedingten Ungenauigkeiten würde der Einsatz von einem anderen Verfahren zu unterschiedlichen Resultaten führen. Deswegen sollte diese Methode zumindest optional noch verwendbar sein, um Vergleiche mit älteren Tests zu ermöglichen.

Kurzfristig empfiehlt sich der Einsatz des unidirektional zielgerichteten Verfahrens zur Ermittlung der ressourcenbeschränkten kürzesten Wege. Es liefert durchweg schnellere Laufzeiten als der bisherige Ansatz und ist deswegen zu bevorzugen. Durch eine optimale Anpassung an die Problemstellung des Route-Guidance-Projektes kann eine weitere Beschleunigung erzielt werden. Die dafür am Programmcode vorzunehmenden Änderungen scheinen relativ gering zu sein.

Zur Lösung des doppelt beschränkten Problems empfehlen wir vorläufig das zielgerichtete Verfahren mit zusätzlichem Abbruchkriterium. Allerdings sind hier noch weitere Tests durchzuführen, bevor eine endgültige Entscheidung getroffen werden kann.

Langfristig erscheint eine Einbindung der 2-Phasen-Methode in das Projekt sinnvoll. Sie ist zwar für die meisten bisher durchgeführten Tests noch ungeeignet, da sie eine gewisse Grundlaufzeit benötigt, dürfte aber spätestens bei der Nutzung von noch größeren Graphen für sehr starke Beschleunigungseffekte sorgen. Zusätzlich könnte die Ermittlung der Lösung in der zweiten Phase durch ein bidirektionales Vorgehen noch schneller emöglicht werden. Der Einsatz von Problemreduktionen vor dem zweiten Schritt ist nicht zwingend notwendig, da dies nicht zwangsweise den Aufwand senkt. Außerdem müssen bei einer Weiterverwendung des Graphen alle entfernten Knoten und Kanten wieder eingefügt werden.

Der bidirektional zielgerichtete Ansatz ist für große Problemstellungen teilweise besser geeignet als der rein zielgerichtete, in diesen Fällen ist jedoch die 2-Phasen-Methode noch schneller. Sein Ansatz wäre also nur dann gerechtfertigt, wenn möglichst schnell ein besseres Abbruchkriterium gefunden würde.

Schließlich bietet der zielgerichtete Ansatz noch einen weiteren Vorteil: Er ist in der Lage, mehrere Pareto-optimale Wege bei einem einzigen Durchlauf zu ermitteln. Es wäre eine Überlegung wert, ob vielleicht einige dieser Wege vom Hauptprogramm benutzt werden könnten, um die Laufzeit noch weiter zu verringern.

Liste der Algorithmen

Literaturverzeichnis

[1] R. Ahuja, T. Magnanti und J. Orlin. *Network Flows*. Prentice-Hall, New Jersey, 1993.

[2] J. Beasley und N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19:379–394, 1989.

[3] G. Beccaria und A. Bolelli. Modelling and assessment of dynamic route guidance: the MARGOT project. In *Vehicle Navigation and Information Conference (VNIS' 92)*, Seiten 117–126. IEEE, Oslo, 1992.

[4] R. Borndörfer und A. Löbel. Scheduling duties by adaptive column generation. Technischer Bericht ZIB-01-02, Konrad-Zuse-Zentrum für Informationstechnk, 2001.

[5] M. Desrochers und F. Soumis. A generalized permanent labeling algorithm for the shortest path problem with time windows. *INFORMS*, 26:191–212, 1988.

[6] A. Elimam und D. Kohler. Two engineering applications of a constrained shortest path model. *European Journal of Operational Research*, 103:426–438, 1997.

[7] M. Garey und D. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H.Freeman, New York, 1979.

[8] G. Handler und I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10:293–310, 1980.

[9] O. Jahn. Multicommodity Flow-Modelle und Algorithmen zur dynamischen Lenkung von Verkehrsströmen, 1998. Diplomarbeit Technische Universität Berlin.

[10] O. Jahn, R. Möhring und A. Schulz. System-Optimal Routing of Traffic Flows with User Constraints in Networks with Congestion. Technischer Bericht 658-1999, July 2000. Extended abstract in: Proceedings of the Symposium on Operations Research, Magdeburg, 1999, (SOR 99).

[11] V. Jimenez und A. Marzal. Computing the *k* shortest paths. A new algorithm and an experimental comparison. In *Proc. 3rd Workshop on Algorithm Engineering (WAE99)*, Band 25, Seiten 15–29. LNCS 1668, Springer, Berlin, 1999.

[12] H. Joksch. The shortest route problem with constraints. *Journal of Mathematical Analysis and Application*, 14:191–197, 1966.

[13] M. Lübbecke und U. Zimmermann. Computer aided scheduling of switching engines. *CASPT2000*, 2000.

[14] R. Möhring. Verteilte Verbindungssuche im öffentlichen Personenverkehr: Graphentheoretische Modelle und Algorithmen. Technischer Bericht 624/99, TU Berlin, 1999. Appeared in: Patrick Horster (ed.) Angewandte Mathematik - insbesondere Informatik, Beispiele erfolgreicher Wege zwischen Mathematik und Informatik, Vieweg Verlag, 1999, pages 192-220.

[15] A. Orda. Routing with end to end QoS guarantees in broadband networks. *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Seiten 27–34, 1998.

[16] R. Sedgewick und J.S. Vitter. Shortest Paths in Euclidean Graphs. *Algorithmica*, 1:31–48, 1986.

[17] C. Skiscim und B. Golden. Solving *k*-shortest and constrained shortest path problems efficiently. *Annals of Operations Research*, 20:249–282, 1989.

[18] K. Stroetmann. The constrained shortest path problem: A case study in using ASMs. *Journal of Universal Computer Science*, 3(4):304–319, 1997.

[19] C. Witzgall und A.J. Goldmann. Most profitable routing before maintenance. *ORSA Bulletin*, 13, 1965.

[20] J. Yen. Finding the *k* shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

[21] M. Ziegelmann. Constrained Shortest Paths and Related Problems, 2001. Dissertation Universität des Saarlandes; Presented at: ESA 2000 (LNCS 1879) Springer.