Lange Schritte beim dualen Simplex-Verfahren

Diplomarbeit von
Sebastian Sager

Betreuer:
Prof. Dr. Hans Georg Bock

UNIVERSITÄT HEIDELBERG
FAKULTÄT FÜR MATHEMATIK

Erklärung

Hiermit erkläre ich, daß ich diese Arbeit selbständig verfaßt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle Stellen, die dem Wortlaut oder Sinne nach anderen Werken entnommen sind, durch Angabe der Quellen als Entlehnungen kenntlich gemacht habe.

Heidelberg, im Juli 2001

Zusammenfassung

Die vorgelegte Arbeit beschäftigt sich mit dem Lösen sogenannter linearer Programme mit Hilfe des Simplex-Algorithmus. Sie leistet dabei Verschiedenes.

In der gebotenen Ausführlichkeit werden Lösungsalgorithmen hergeleitet, dargestellt und ihre Korrektheit bewiesen. Hierbei wird besonderer Wert auf die Verwendung der Zeilenbasis gelegt, die mathematisch äquivalente Algorithmen hervorbringt, die in der konkreten Implementierung jedoch zu erheblichen Verbesserungen führen können.

Desweiteren wird mit dem Programmpaket SoPlex von Roland Wunderling eine moderne Implementierung eines LP-Solvers in einer objektorientierten Programmiersprache beschrieben. Die prinzipielle Funktionsweise und der strukturelle Aufbau des Programmes werden genauso beleuchtet wie die Details, die zum Großteil dem neuesten Stand der Forschung entsprechen.

Schließlich wird ein bisher wenig beachteter Ansatz, den sogenannten "ratio test" beim dualen Simplex auszuführen, theoretisch und praktisch untersucht. Bei diesem geht es darum, die duale Zulässigkeit bei ungleichungsbeschränkten Variablen durch bestimmte Maßnahmen zu erhalten, um einen zulässigen längeren Schritt ausführen zu können, der nicht über eine Kante zu einer Nachbarecke, sondern durch das Polyeder hindurch zu einer anderen Ecke führt. Dieses Verfahren wird erstmals auf eine Zeilenbasis angewandt, womit es auch für einfügende Algorithmen anwendbar ist — dies erlaubt die Nutzung zusammen mit anderen Methoden, die ausschließlich mit einfügenden Algorithmen funktionieren und die Ausnutzung von Dimensionsvorteilen.
Die Stabilität des Simplex-Algorithmus wird untersucht und es werden stabilisierte Varianten der Methoden angegeben.

Die Verfahren wurden in C++ programmiert und in das Programmpaket SoPlex integriert, um repräsentative Vergleichswerte mit anderen ratio test Methoden zu erhalten. Als Testprobleme dienten ausgewählte lineare Programme aus den Bibliotheken `netlib` und `miplib`.

Es zeigte sich, daß die vorgenommenen Stabilisierungsmaßnahmen Erfolg hatten, von fast 200 Testdurchläufen schlug nur noch einer fehl. Für bestimmte Programme benötigt die neue Methode erheblich weniger Iterationen und Zeit als die implementierte state-of-the-art Methode von SoPlex .

Die Bestimmung günstigerer Stabilisierungsmethoden und schneller Heuristiken zur Vermeidung unnötiger Schritte läßt Spielraum für weitere Forschung.

Inhaltsverzeichnis

Mathematische Symbole **III**

0 Einleitung **1**

1 Lineare Optimierung mit dem Simplex-Algorithmus **5**
1.1 Definitionen und mathematische Grundlagen 5
1.1.1 Lineare Programme 7
1.1.2 Polyeder 8
1.2 Der Simplex-Algorithmus 14
1.3 Dualität 19
1.3.1 Primales und Duales Programm 19
1.3.2 Der duale Algorithmus 22
1.3.3 Vorteile des dualen Algorithmus 24
1.3.4 Wirtschaftliche Deutung der dualen Variablen 25
1.4 Zusammenhang mit nichtlinearer Programmierung 26
1.5 Besonderheiten bei der Implementierung 28
1.5.1 Bestimmung einer Startbasis 28
1.5.2 Pricing Strategien 30
1.5.3 Ratio test Strategien, Stabilität 32
1.5.4 Degeneriertheit 37
1.5.5 Updates 38
1.6 Der Simplex-Algorithmus bei beschränkten Variablen 39
1.7 Die Zeilendarstellung der Basis 45
1.8 Der Algorithmus mit allgemeiner Basis 48

2 Das Softwarepaket SoPlex **52**
2.1 Objektorientierte Programmierung und C++ 52
2.2 Unified Modeling Language 56
2.3 Beschreibung von SoPlex 58
2.3.1 Die Klassenstruktur von SoPlex 58
2.3.2 Elementare Klassen 60
2.3.3 Die Basis 60
2.3.4 Die Vektoren f, g und h 64
2.3.5 Simplex Klassen 64
2.3.6 Parallelisierung 66
2.4 Die Schnittstelle der ratio test Klassen 67

3 Lange Schritte im dualen Algorithmus **68**
3.1 Theoretische Grundlagen 68
3.2 Implementierung im Simplex 74
3.2.1 Der LongStep Algorithmus 75
3.3 Implementierung bei einer Zeilenbasis 77
3.4 Updates 81
3.5 Stabilität 83
3.6 Die Klasse `SPxLongStepRT` 86

4 Tests und Ergebnisse **89**
4.1 `netlib` Tests 90
4.2 `miplib` Tests 99
4.3 Zusammenfassung 104

A Quellcode **105**

Abbildungsverzeichnis **110**

Tabellenverzeichnis **111**

Algorithmenverzeichnis **112**

Literaturverzeichnis **113**

Mathematische Symbole

Schreibweisen

$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{N}$	Menge der natürlichen Zahlen
$\mathbb{Z}$	Menge der ganzen Zahlen
I	Einheitsmatrix, Dimension geht aus Zusammenhang hervor
$\vec{e_q}$	q-ter kanonischer Einheitsvektor, Dimension geht aus Zusammenhang hervor
0	Null, Nullvektor oder Nullmatrix, Dimension geht aus Zusammenhang hervor
$vec_{\{j_1,\ldots,j_k\}}$	Vektor bestehend aus den zu der Indexmenge gehörenden Komponenten von vec, z.B. c_B
$A_{\{j_1,\ldots,j_k\}\cdot}$	Matrix bestehend aus den zu der Indexmenge gehörenden Zeilen von A, z.B. $A_{B\cdot}$
$A_{i\cdot}$	i-te Zeile der Matrix, Zeilenvektor.
$A_{\cdot\{j_1,\ldots,j_k\}}$	Matrix bestehend aus den zu der Indexmenge gehörenden Spalten von A, z.B. $A_{\cdot B}$
$A_{\cdot i}$	i-te Spalte der Matrix, Spaltenvektor, auch a_i.
Y^T	Transponierte der Matrix bzw. des Vektors Y
Y^{-1}	Inverse der Matrix Y
Y^{-T}	Transponierte der Inversen der Matrix Y
ΔY	Differenz zwischen neuem und altem Y, im allgemeinen Richtung des Updates
s.t.	unter der Bedingung daß, subject to
$x = (x_1, x_2)$	$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$

Verwendete Bezeichner

A'	Matrix der Nebenbedingungen im $\mathbb{R}^{m \times n}$
a_i	Vektor, i-te Spalte von A
b	Vektor im $\mathbb{R}^m$, Nebenbedingungsvektor im primalen, Kostenvektor im dualen Programm
c	Vektor im $\mathbb{R}^n$, Kostenvektor im primalen, Nebenbedingungsvektor im dualen Programm
j_i	Index einer Basisvariable, $j_i \in B$
j_q	aus der Basis entfernter (quiting) Index, $j_q \in B, q \in \{1, \ldots, m\}$
J	Indexmenge $\{1, \ldots, n\}$ bzw. $\{1, \ldots, n+m\}$
$J_i^{\leq,\leq}$	und ähnliche, siehe Definition 3.2 auf Seite 70
m	Anzahl der Nebenbedingungs(un)gleichungen im primalen Programm
n	Anzahl der Variablen im primalen Programm
n_e	in die Basis wechselnder (entering) Index, $n_e \in N$, $e \in \{1, \ldots, n-m\}$
n_i	Index einer Nichtbasisvariable, $n_i \in N$
s	Schlupfvariable
v	Vektor im $\mathbb{R}^{n+m}$, zu den Ungleichungen $L \leq x$ gehörende duale Variable
w	Vektor im $\mathbb{R}^{n+m}$, zu den Ungleichungen $U \geq x$ gehörende duale Variable
x	Vektor im $\mathbb{R}^n$, Zustandsvariablen des primalen Programms
y	Vektor im $\mathbb{R}^m$, Zustandsvariablen des dualen Programms
α_i	Steigung der dualen Zielfunktion im Intervall $I = [\Theta^i, \Theta^{i+1}]$ bzw. $I = [\Phi^i, \Phi^{i+1}]$
λ	Vektor im $\mathbb{R}^{2n+3m}$, Zustandsvariablen des dualen Problems bei Schlupfvariablen und Bereichsungleichungen, $\lambda = (y, v, w)$
$\phi(\lambda)$	Duale Zielfunktion $b^T y + U^T w - L^T v$
Φ	Schrittweite des Updates von f
Φ^i	$\Phi^i = \max\{\Phi > \Phi^{i-1} : y_j(\Phi) y_j(\Phi^{i-1}) \geq 0 \, \forall \, j \in J\}$
Θ	Schrittweite des Updates von g und h
Θ^i	$\Theta^i = \max\{\Theta > \Theta^{i-1} : g_j(\Theta) g_j(\Theta^{i-1}) \geq 0 \, \forall \, j \in J\}$

Spaltenbasis

A	Matrix im $\mathbb{R}^{m\times(n+m)}$ der Nebenbedingungen mit Schlupfvariablen
$A_{\cdot B}$	Basismatrix im $\mathbb{R}^{m\times m}$, bestehend aus den zur Spaltenbasis B gehörenden linear unabhängigen Spalten von A
B	Menge der Indizes der Basisvariablen $\{j_1,\ldots,j_m\}$, $B \subseteq J$
B_u, B_l, B_x, B_f	Mengen von Indizes j der Nichtbasisvariablen. Definition Seite 41.
N	Menge der Indizes der Nichtbasisvariablen $J \setminus B$, $N \subseteq J$
N_u, N_l, N_x, N_f	Mengen von Indizes j der Nichtbasisvariablen. Definition Seite 41.
f	Zulässigkeitsvektor des Programms, entspricht $x_B \in \mathbb{R}^m$
g	pricing-Vektor des Programms mit Dimension n bzw. $n+m$, in der Literatur oft mit $\mu, \bar{c}$ oder d bezeichnet
h	Hilfsvektor oder copricing-Vektor des Programms mit Dimension m, in der Literatur oft mit y oder π bezeichnet
l, u	Vektoren im $(\mathbb{R} \cup -\infty)^{n+m}$, Grenzen der Variablen g
L, U	Vektoren im $(\mathbb{R} \cup -\infty)^{n+m}$, Grenzen der Variablen f
R	Vektor im $\mathbb{R}^{n+m}$, gibt den aktuellen Wert der Nichtbasisvariablen an
r	Vektor im $\mathbb{R}^m$, entspricht c_B

Zeilenbasis

A	Matrix im $\mathbb{R}^{n\times(n+m)}$ der Nebenbedingungen mit Schlupfungleichungen
$A_{\cdot B}$	Basismatrix im $\mathbb{R}^{m\times m}$, bestehend aus den zur Zeilenbasis B gehörenden linear unabhängigen Spalten von A
B	Menge der Indizes der Basisvariablen $\{j_1,\ldots,j_n\}$, $B \subseteq J$
B_u, B_l, B_x, B_f	Mengen von Indizes j der Nichtbasisvariablen. Definition Seite 47.
N	Menge der Indizes der Nichtbasisvariablen $J \setminus B$, $N \subseteq J$
N_u, N_l, N_x, N_f	Mengen von Indizes j der Nichtbasisvariablen. Definition Seite 47.
f	Zulässigkeitsvektor des Programms, entspricht $y_B \in \mathbb{R}^m$
g	pricing-Vektor des Programms mit Dimension n bzw. $n+m$, entspricht Schlupfvektor s
h	Hilfsvektor oder copricing-Vektor des Programms mit Dimension n, entspricht x bei Verwendung einer Spaltenbasis
l, u	Vektoren im $(\mathbb{R}\cup-\infty)^{n+m}$, Ungleichungs-Grenzen für $s = g = Ah$
L, U	Vektoren im $(\mathbb{R} \cup -\infty)^{n+m}$, Grenzen der Variablen f
R	Nullvektor um strukturelle Ähnlichkeit hervorzuheben
r	Vektor im $\mathbb{R}^n$, aktuelle rechte Seite der Gleichungen $A_{\cdot B}^T h = r$

Kapitel 0

Einleitung

Die lineare Programmierung beschäftigt sich mit dem Lösen linearer Programme. Dabei geht es um das Optimieren einer linearen Funktion $c^T x$ auf einer durch Gleichungen und Ungleichungen beschriebenen Menge P.
Solche Programme treten in vielen Situationen der realen Welt, insbesondere bei wirtschaftlichen Fragestellungen, auf. Hier spielt die lineare Programmierung oft eine bedeutende Rolle, wie der in [Pad99] wiedergegebene Auszug aus einem Artikel der *New York Times* vom 19. November 1984 verdeutlicht:

> *"The discovery, which is to be formally published next month, is already circulating rapidly through the mathematics world. It has also set off a deluge of inquiries from brokerage houses, oil companies and airlines, industries with millions of dollars at stake in problems known as linear programming."*

Warum geht es hier um soviel Geld? Wirtschaftliche Situationen können durch lineare Programme beschrieben werden, die zulässige Menge P gibt alle möglichen Entscheidungen an. Eine unter Kostenpunkten optimale Lösung ist dann natürlich bares Geld wert. Für komplexere Entscheidungsprozesse ist eine optimale Lösung ohne mathematisches Handwerk allerings nicht mehr zu bestimmen. Wir wollen ein einfachstes Beispiel betrachten um die Ausgangssituation besser zu verstehen.

Ein Alkoholproduzent habe zwei Produkte im Angebot, Bourbon und Scotch. Er möchte planen, wieviel er von beiden für die Winterperiode produzieren soll um möglichst viel Geld zu verdienen. Es gelten die folgenden Beschränkungen:

- Destilliermaschinen inklusive Arbeitskraft stehen 40000 Arbeitsstunden zur Verfügung.
- Das Destillieren eines Liters Bourbon nimmt 3 Stunden Maschinenzeit in Anspruch, für einen Liter Scotch sind 4 Stunden nötig.
- Die Produktionskosten betragen 5 Euro für den Bourbon und 3 Euro 50 für den Scotch.
- Der Verkaufspreis beträgt 9 Euro 50 für den Bourbon und 9 Euro für den Scotch.
- Als Investitionskapital stehen 50000 Euro zur Verfügung.

- Die Verkaufsexperten der Firma schätzen, daß nur je 7500 Liter problemlos abgesetzt werden können.

Die Menge an produzierten Litern Bourbon bezeichnen wir mit der Variable x_1, die produzierte Menge Scotch mit x_2. Der Gewinn in Euro beträgt

$$(9.5 - 5)\, x_1 + (9 - 3.5)\, x_2$$

Die begrenzte Arbeitszeit der Maschinen führt zu der Beschränkung

$$3x_1 + 4x_2 \quad \leq \quad 40000$$

Die Investitionskosten dürfen das vorhandene Kapital nicht überschreiten, daher

$$5x_1 + 3.5x_2 \quad \leq \quad 50000$$

Um den Markt nicht zu überlasten, beschränkt man die Menge auf je 7500 Liter

$$\begin{aligned} 0 \leq \quad x_1 \quad &\leq 7500 \\ 0 \leq \quad x_2 \quad &\leq 7500 \end{aligned}$$

Das zugehörige lineare Programm sieht damit folgendermaßen aus:

$$\begin{array}{lrcr} \max & 4.5x_1 + 5.5x_2 & & \\ \text{s.t.} & 3x_1 + 4x_2 & \leq & 40000 \\ & 5x_1 + 3.5x_2 & \leq & 50000 \\ & x_1, x_2 & \leq & 7500 \\ & x_1, x_2 & \geq & 0 \end{array}$$

s.t. steht hier für "subject to", was soviel wie "unter der Bedingung daß" bedeutet.

Komplexere Beispiele für lineare Programme sind scheduling-Probleme, bei denen es darum geht, vorhandene Ressourcen (Crew, Flugzeuge, Busse, Maschinen, ...) derart einzusetzen, daß der Bedarf gedeckt ist, bestimmte Bedingungen (Wartung, Arbeitszeitbestimmungen, Aufwärmphasen, Ersatz, ...) erfüllt werden und eine Zielfunktion (Kosten, Gewinn, Arbeitszeit der Maschinen, ...) minimiert oder maximiert wird. Eine Variable x_i ist beispielsweise genau dann 1, wenn das Flugzeug A am Tag B von C nach D fliegt und ansonsten null. Der zugehörige Zielfunktionswert c_i gibt die anfallenden Kosten pro Maschine an und die Ungleichungen und Gleichungen decken die Erfordernisse ab.

Das letzte Beispiel zeigt, wie schnell man zu einer recht großen Anzahl von Variablen und Ungleichungen kommen kann. Für eine kleine regionale Airline mit 20 Flugzeugen, 10 angeflogenen Städten, nur einem Flug pro Maschine und Tag und einem Planungshorizont von einem Monat erhält man schon $20*10*10*30 = 60000$ Variablen. Diese großen Programme erfordern spezialisierte Lösungsalgorithmen.

Der erste solche Algorithmus war der Simplex-Algorithmus und wurde 1947 von Dantzig vorgestellt. Er basiert auf der Erkenntnis, daß eine optimale Lösung immer in einer Ecke des zulässigen Polyeders — also der geometrischen Figur, die durch die Nebenbedingungen beschrieben wird — liegt und diese relativ leicht bestimmt werden können. Da eine Berechnung aller Ecken zu aufwendig ist, verfolgt er einen Weg von Ecke zu Ecke, wobei nur Ecken

behandelt werden, die einen besseren Zielfunktionswert aufweisen als die Vorgängerecke. Man kann Beispiele konstruieren, in denen der Algorithmus trotzdem alle Ecken berechnen muß, daher hat der Algorithmus auch eine exponentielle Laufzeit (wenn man den worst case zugrunde legt). Im Durchschnitt gesehen ist die Laufzeit allerdings polynomial.
Der Simplex-Algorithmus wurde über die Jahre immer weiter verbessert und ist heute von der Effizienz immer noch ähnlich gut wie andere Lösungsansätze, vornehmlich die Innere-Punkte-Methoden, die einen Weg durch das Innere des Polyeders verfolgen. Diese Arbeit will dazu beitragen, den Simplex-Algorithmus in seiner praktischen Form weiter zu verbessern. Als Grundlage dient der sogenannte revidierte Simplex-Algorithmus, der die Daten nicht wie in der ursprünglichen Form in Tabellen verändert, sondern auf Matrizen und Vektoren arbeitet. Diese Form ist die in modernen Implementierungen gängige, daher wird in dieser Arbeit der Zusatz revidiert weggelassen.

Die in dieser Arbeit vorgestellte Methode wird als Variante des Simplex-Algorithmus beschrieben, bei der lediglich der ratio test und das Update des Zulässigkeitsvektors modifiziert werden. Da die Modifizierung dazu führt, daß die Basislösung nicht mehr von Ecke zu Ecke über Verbindungskanten, damit also außen um das Polyeder herum, sondern zu einer beliebigen anderen Ecke des Polyeders mit besserem Zielfunktionswert wechselt, ist die Frage durchaus berechtigt, ob die Definition Simplex-Verfahren noch zutrifft. R. Gabasov, auf den diese Idee zurückgeht, bezeichnete das Verfahren als "Adaptive Methode", da die Auswahl der nächsten Basislösung den kompletten Zielfunktionsvektor c berücksichtigt. Aufgrund der strukturellen Ähnlichkeit und leichten Anwendbarkeit auf die vorgestellten Algorithmen scheint der Begriff "LongStep ratio test" jedoch passender und wird in dieser Arbeit verwendet.

Kapitel 1 widmet sich einer formalen Herleitung des revidierten Simplex-Algorithmus. Nachdem die mathematischen Grundlagen geschaffen sind (1.1), werden der primale und duale Algorithmus in 1.2 und 1.3 hergeleitet und ihre partielle Korrektheit bewiesen. Ein kleiner Exkurs in Abschnitt 1.4 stellt den Zusammenhang zu den Begriffen der nichtlinearen Optimierung her. Der Abschnitt 1.5 behandelt Besonderheiten, die bei einer Implementierung des Algorithmus beachtet werden müssen, will man ein stabiles und effizientes Verfahren programmieren. Der Abschnitt 1.6 beschreibt die Vorgehensweise bei beschränkten Variablen, die zwar keine mathematischen, dafür aber algorithmische Unterschiede aufweist. Die beiden letzten Abschnitte widmen sich einer weiteren Besonderheit, der Zeilenbasis. Mit dieser läßt sich durch einfache Umdefinition einiger Größen das duale Programm mit den vorhandenen Algorithmen lösen, was gerade einen Tausch der beiden Kenngrößen "Anzahl der Variablen" und "Anzahl der (Un-)Gleichungsbeschränkungen" zur Folge hat. Dies ist extrem wichtig für das Laufzeitverhalten.

Das zweite Kapitel ist dem Softwarepaket SoPlex gewidmet. Dieses diente einerseits als Untersuchungsobjekt einer modernen, schnellen und zuverlässigen Implementierung des Simplex-Algorithmus und andererseits als Basis für Tests des neuen ratio tests.
SoPlex ist in der objektorientierten Programmiersprache C++ geschrieben. Abschnitt 2.1 nennt die wichtigsten Konzepte und Vorteile in Abgrenzung zu den prozeduralen Programmiersprachen. Der Rest des Kapitels dient der Beschreibung des Aufbaus von SoPlex im allgemeinen und der ratio test Klassen im besonderen.

Das dritte Kapitel widmet sich einer bisher wenig bis gar nicht beachteten Methodik, den ratio test durchzuführen. Abschnitt 3.1 bietet die theoretische Herleitung des Verfahrens, in 3.2 und 3.3 wird es in die im ersten Kapitel formulierten Algorithmen für Spalten- und Zeilenbasis eingebaut. Die Abschnitte 3.4 und 3.5 widmen sich mit einer Betrachtung der Updates und möglicher Stabilisierungsmaßnahmen den Implementierungsdetails. Abgeschlossen wird das Kapitel durch die Beschreibung der eingebauten ratio test Klasse `SPxLongStepRT`.

Die verschiedenen Stufen der Implementierung wurden, genau wie die vorhandenen ratio test Klassen, anhand gegebener linearer Programme auf ihre Stabilität und Effektivität getestet. Um gute Vergleichswerte zu haben, wurde das Paket SoPlex mit festen Einstellungen verwendet, nur die benutzte Methode für die Ausführung des ratio tests wurde verändert. Die Ergebnisse, eine Zusammenfassung und ein kurzer Ausblick sind in Kapitel vier zusammengetragen um die Arbeit abzuschließen.

Kapitel 1

Lineare Optimierung mit dem Simplex-Algorithmus

In diesem Kapitel sollen die schon seit längerer Zeit bekannten mathematischen und algorithmischen Grundlagen der linearen Optimierung und des Simplex-Algorithmus dargestellt werden, die für das Verständnis des Lösens von linearen Programmen mit Hilfe von Simplex-Algorithmen im allgemeinen und der vorgenommenen Änderungen im besonderen notwendig sind. Es wird die Struktur von sogenannten linearen Programmen untersucht und hieraus der Simplexalgorithmus als Lösungsalgorithmus hergeleitet, zuerst für strukturell einfache Programme ohne Schranken für die Variablen um den geometrischen Zusammenhang deutlicher herausstellen zu können. Nach einem Abschnitt über Dualität und den dualen Algorithmus werden der Zusammenhang mit der nicht-linearen Optimierung, die algorithmische Behandlung von Schranken und Besonderheiten bei der Implementierung beleuchtet. Das Kapitel wird abgeschlossen durch einen Abschnitt über die Verwendung einer Zeilen- und einer allgemeinen Basis als Alternative zur gängigen Spaltenbasis.

1.1 Definitionen und mathematische Grundlagen

In diesem Abschnitt werden die grundlegenden mathematischen Objekte definiert.
Da in der Literatur immer wieder andere Bezeichnungen auftauchen, ist es sinnvoll, die in dieser Arbeit verwendeten kurz zu benennen, siehe hierzu auch die vorangestellte Übersicht der mathematischen Symbole. Dimensionen sollten, wenn sie nicht eindeutig benannt sind, aus dem Zusammenhang erkennbar sein. Beziehungsoperationen zwischen Vektoren sind immer komponentenweise zu verstehen. Die Einheitsmatrix wird mit I, Einheitsvektoren mit $\vec{e_q}$ bezeichnet. In den vorgestellten Algorithmen kommt es zu Updates, in denen ein Gleichheitszeichen vorkommt, z.B. $f = f + \Delta f$. Dieses Gleichheitszeichen entspricht der Zuweisung in Programmiersprachen und nicht einer mathematischen Beziehung zwischen linker und rechter Seite.
Seien im folgenden, soweit nicht anders angegeben, $c, x, l', u', g \in I\!R^n,\ b, y, l'', u'', f, h \in I\!R^m$ und $A \in I\!R^{m \times n}$. Updatevektoren sind durch ein vorgestelltes Δ gekennzeichnet und haben die gleiche Dimension wie der Vektor selber. Sind Schranken für eine Variable oder einen Ausdruck gegeben, so wird vorausgesetzt, daß die untere kleiner gleich der oberen ist, wobei auch Werte $\pm\infty$ möglich sind. Für Vektoren, die in der Form $x = (x_B, x_N)$ beschrieben werden, sei eine entsprechende Umnumerierung der Indizes vorausgesetzt.

Die Vektoren b, c, l', l'', u', u'' und auftauchende Matrizen seien gegeben, während f, g, h, x, y variabel sind. In diesem Sinne ist $\max c^T x$ auch als Maximum aller Werte x zu verstehen, die den nachfolgenden Bedingungen genügen.

An einigen Stellen dieser Arbeit werden Vektor- bzw. Matrixnormen benutzt. Hier soll kurz definiert werden, was damit gemeint ist.

Definition 1.1 *Seien $v, w \in \mathbb{R}^n$. Die Abbildung $\| \cdot \| : \mathbb{R}^n \to \mathbb{R}$ heißt Vektornorm, wenn*

1. $\|v\| = 0 \Longleftrightarrow v = 0$
2. $\|\alpha\, v\| = |\alpha|\, \|v\| \quad \forall\ \alpha \in \mathbb{R}$
3. $\|v + w\| \leq \|v\| + \|w\| \quad \forall\ v, w \in \mathbb{R}^n$

Definition 1.2 *Seien $A, B \in \mathbb{R}^{m \times n}$. Die Abbildung $\| \cdot \| : \mathbb{R}^{m \times n} \to \mathbb{R}$ heißt Matrixnorm, wenn*

1. $\|A\| > 0 \quad \forall\ A \neq 0$
2. $\|\alpha\, A\| = |\alpha|\, \|A\| \quad \forall\ \alpha \in \mathbb{R}$
3. $\|A + B\| \leq \|A\| + \|B\|$

Sie heißt verträglich mit der Vektornorm $\| \cdot \|_a$, wenn

$$\|A\, x\|_a \quad \leq \quad \|A\|\, \|x\|_a \quad \forall\ x \in \mathbb{R}^n$$

Sie heißt submultiplikativ, wenn

$$\|AB\| \quad \leq \quad \|A\|\, \|B\| \quad \forall\ A, B \in \mathbb{R}^{m \times n}$$

Im folgenden bezeichne $\| \cdot \|$ je nach Argument eine Vektornorm oder eine mit dieser gegebenen Vektornorm verträgliche submultiplikative Matrixnorm mit $\| I \| = 1$. In dieser Arbeit werden hierfür nur die euklidische Vektornorm

$$\|x\| \quad := \quad \|x\|_2 = \sqrt{< x, x >}$$

und die Supremumsnorm

$$\|A\| \quad := \quad |||A||| = \sup_{\|x\|_2 = 1} \|Ax\|_2$$

als Matrixnorm benutzt.

1.1.1 Lineare Programme

Definition 1.3 *Lineare Programme (LPs) sind Optimierungsprobleme der Form*

$$\begin{array}{lllll} \max / \min & c_1^T x_1 + c_2^T x_2 + c_3^T x_3 & & \\ s.t. & A_{11}x_1 + A_{12}x_2 + A_{13}x_3 & \leq & b_1 \\ & A_{21}x_1 + A_{22}x_2 + A_{23}x_3 & \geq & b_2 \\ & A_{31}x_1 + A_{32}x_2 + A_{33}x_3 & = & b_3 \\ & x_1 & \geq & l \\ & x_2 & \leq & u \\ & x_3 & & \textit{frei} \end{array}$$

Für $n_1, n_2, n_3, m_1, m_2, m_3 \in \mathbb{N}$ und $n_1 + n_2 + n_3 = n$ sowie $m_1 + m_2 + m_3 = m$ seien hierbei $c_i, x_i \in \mathbb{R}^{n_i}, b_i \in \mathbb{R}^{m_i}, l \in \mathbb{R}^{n_1}, u \in R^{n_2}$ und $A_{ij} \in R^{m_i \times n_j}$.

Die in der Definition angegebene ist die allgemeinste Form, die man oftmals bei der mathematischen Modellierung erhält. Um dieses Programm etwas griffiger zu haben, vereinfacht man die Darstellung durch einige mathematische (nicht: *algorithmische*) Äquivalenzumformungen.
Zu den Umformungsmöglichkeiten des linearen Programms (LP) zählen

- Das Minimieren einer Funktion entspricht dem Maximieren der negativen Funktion, $\min \{c^T x\} = \max \{-c^T x\}$
- Aus beschränkten Variablen kann man durch Erweiterung der Matrix A und des Vektors b formal freie Variablen bekommen. Für $x = (x_1,\ x_2,\ x_3)$ ist

$$\begin{array}{|lllll|} \hline \max & c^T x & & & \\ \text{s.t.} & Ax & \leq & b & \\ & x_1 & \geq & l & \\ & x_2 & \leq & u & \\ & x_3 & & \text{frei} & \\ \hline \end{array} \quad \text{äquivalent zu} \quad \begin{array}{|lll|} \hline \max & c^T x & \\ \text{s.t.} & \begin{pmatrix} & A & \\ 0 & I & 0 \\ -I & 0 & 0 \end{pmatrix} x & \leq \begin{pmatrix} b \\ u \\ -l \end{pmatrix} \\ & x_1, x_2, x_3 & \text{frei} \\ \hline \end{array}$$

- Die Ungleichungen $Ax \leq b$ und $Ax \geq b$ sind der Gleichung $Ax = b$ äquivalent
- $\leq$-Ungleichungen kann man durch Multiplikation mit -1 in $\geq$-Ungleichungen umformen und andersherum
- Aus einer Ungleichung kann man durch Einführen sogenannter Schlupfvariablen eine Gleichung machen: $\{x : Ax \leq b\} = \{x : Ax + s = b \text{ für beliebiges } s \geq 0\}$
- Ist eine Variable x_i durch l_i nach unten beschränkt, so erhält man durch Substitution $x' := x_i - l_i$ und Modifizierung von $b' = b - l_i A_{\cdot i}$ ein äquivalentes Programm mit unterer Schranke $l' = 0$
- Eine freie Variable x_i entspricht der Differenz zweier positiver Variablen: $x_i = x_i' - x_i''$ $x', x'' \geq 0$. Man kann also auch aus freien Variablen vorzeichenbeschränkte erhalten

Für die weiteren theoretischen Untersuchen von linearen Programmen kann man sich daher ohne Beschränkung der Allgemeinheit eine passende Form auswählen, so beispielsweise

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Ax & \leq \quad b \end{array} \tag{1.1}$$

Die Form des zugrundeliegenden linearen Programms wird in dieser Arbeit an verschiedenen Stellen geändert um Eigenschaften des Programmes besser herausarbeiten zu können.

Definition 1.4 *Existiert kein $x \in \mathbb{R}^n$ mit $Ax \leq b$, so heißt das lineare Programm (1.1) unzulässig. Existiert hingegen zu jedem $M \in \mathbb{R}$ ein $x \in \mathbb{R}^n$, so daß $Ax \leq b$ und $c^T x < M$, so heißt (1.1) unbeschränkt.*

1.1.2 Polyeder

Bei der linearen Programmierung geht es also darum, aus einer Menge
$P = \{x : Ax \leq b\}$ ein x auszuwählen, das den Ausdruck $c^T x$ minimiert. Wie im folgenden erläutert wird, handelt es sich bei dieser Menge um ein konvexes Polyeder. Für eine ausführlichere Darstellung der Polyedertheorie siehe beispielsweise [Pad99], dieser Abschnitt ist an [NKT89] angelehnt.
Folgende Definitionen sind bei der Untersuchung der sogenannten zulässigen Menge hilfreich:

Definition 1.5 *Eine Teilmenge M des $\mathbb{R}^n$ heißt konvex, wenn für alle Punkte x und y aus M und beliebiges $0 \leq \lambda \leq 1$ aus $\mathbb{R}$ auch $\lambda x + (1-\lambda)y$ in M liegt.*

Anschaulich bedeutet dies, daß eine Teilmenge M genau dann konvex ist, wenn für zwei beliebige Punkte aus ihr alle Punkte auf der Verbindungsstrecke ebenfalls in M enthalten sind.

Definition 1.6 *x heißt Konvex-Kombination der paarweise verschiedenen Vektoren $x_1, \ldots, x_N$ wenn*

$$x \quad = \quad \sum_{i=1}^{N} \lambda_i \, x_i \qquad \textit{und} \qquad \sum_{i=1}^{N} \lambda_i \quad = \quad 1 \quad \lambda_i \geq 0 \; \forall \; i \in \{1, \ldots, N\}$$

Definition 1.7 *Die Menge $H_1 = \{x \in \mathbb{R}^n : a^T x = b\}$ mit gegebenen $a \neq 0 \in \mathbb{R}^n$ und $b \in \mathbb{R}$ ist eine Hyperebene. Die Menge $H_2 = \{x \in \mathbb{R}^n : a^T x \leq b\}$ ist ein Halbraum. Die zum Halbraum H_2 gehörende Hyperebene H_1 wird auch als begrenzende Hyperebene bezeichnet.*

Definition 1.8 *Die Lösungsmenge von endlich vielen linearen Ungleichungen wird konvexes Polyeder oder auch nur Polyeder genannt. Ist diese Menge beschränkt und nichtleer, so spricht man von einem Polytop.*

Lemma 1.9 *Konvexe Polyeder und Polytope sind konvex.*

Beweis. Offensichtlich sind Halbräume konvex: Sind $x, y \in H = \{x : a^T x \leq b\}$, so folgt für $0 \leq \lambda \leq 1$ unmittelbar $a^T(\lambda x + (1-\lambda)y) = \lambda a^T x + (1-\lambda) a^T y \leq \lambda b + b - \lambda b = b$, daher $\lambda x + (1-\lambda)y \in H$.
Der Durchschnitt zweier konvexer Mengen ist trivialerweise auch wieder konvex: Für alle

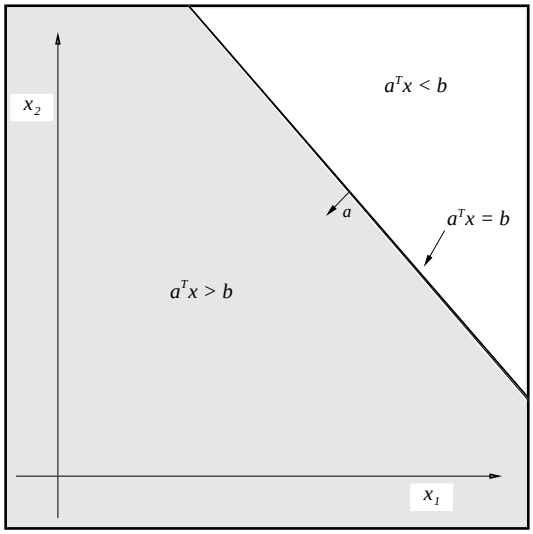


Abbildung 1.1: Hyperebene $a^T x = b$ und zugehörige Halbräume im $\mathbb{R}^2$

Punkte x, y aus dem Durchschnitt liegt jeder Punkt $\lambda x + (1-\lambda)y$ nach Voraussetzug wieder in beiden Mengen und damit auch im Durchschnitt.
Da Polyeder Schnittmengen von endlich vielen Halbräumen sind und Polytope Spezialfälle von Polyedern, folgt die Behauptung. ■

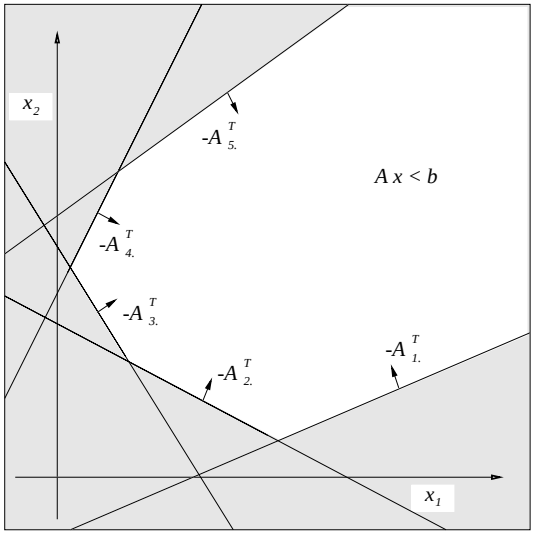


Abbildung 1.2: Unbeschränktes Polyeder $P = \{x : Ax \leq b\}$ im $\mathbb{R}^2$, $A \in \mathbb{R}^{5\times 2}$

Die zulässige Menge $P = \{x : Ax \leq b\}$ ist also ein konvexes Polyeder oder, anders formuliert, die Schnittmenge von endlich vielen Halbräumen. Sucht man in diesem Polyeder nun nach dem Maximum bzw. Minimum, so kann man sich für den zwei- oder dreidimensionalen Fall schnell überlegen, daß das Optimum, wenn es denn überhaupt eines gibt, in

einer Ecke angenommen werden muß. Dies liegt einfach daran, daß man die Hyperebene $c^T x = \beta$ durch die Wahl von β parallel verschieben kann, bis sie keine inneren Punkte des Polyeders mehr berührt, wie es in Abbildung 1.3 angedeutet ist.

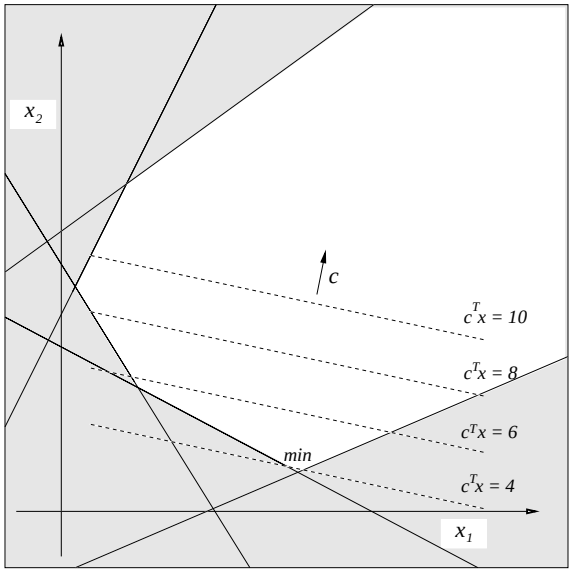


Abbildung 1.3: Höhenlinien der Zielfunktion in zulässiger Menge

Um überprüfen zu können, ob ein ähnliches Resultat auch in höheren Dimensionen gilt, nimmt man die folgenden Definitionen vor.

Definition 1.10 *Gegeben sei eine konvexe Menge C und ein Halbraum $\hat{H}$ mit $C \subseteq \hat{H}$. Gilt für die begrenzende Hyperebene H von $\hat{H}$ nun $C \cap H \neq \emptyset$, so nennt man H eine C unterstützende Hyperebene.*

Definition 1.11 *Sei P ein konvexes Polyeder und H eine unterstützende Hyperebene. Die Schnittmenge $F = P \cap H$ wird Seite oder auch Seitenfläche genannt.*

Definition 1.12 *Die Dimension eines affinen Unterraumes $W = v + U$ ist gleich der Dimension des zugehörigen Untervektorraumes U, also gleich der maximalen Anzahl linear unabhängiger Vektoren.*

Definition 1.13 *Eine Ecke des n-dimensionalen Polyeders P ist eine Seitenfläche der Dimension null. Eine Kante des Polyeders ist eine Seitenfläche der Dimension eins. Seitenflächen der Dimension $n-1$ nennt man Facetten.*

Lemma 1.14 *Folgende Charakterisierungen eines Punktes v innerhalb eines Polyeders P sind äquivalent:*

1. *$v \in P$ ist eine Ecke.*

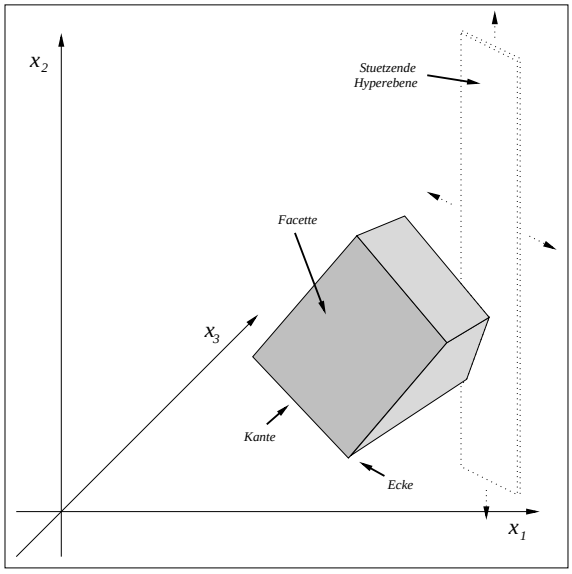


Abbildung 1.4: Dreidimensionales Polytop

2. *$v \in P$ ist ein extremaler Punkt, er kann also nicht als Konvexkombination anderer Punkte aus P gebildet werden.*

Beweis.

- Sei $v \in P$ darstellbar als Konvexkombination $v = \sum_{i=1}^{N} \lambda_i x_i$, wobei die $x_i \in P$, $0 < \lambda_i < 1$, also mindestens ein x_i echt verschieden von v.
  Sei v nun in einer Seitenfläche enthalten und gelte $a^T v = \beta$ für die unterstützende Hyperebene. Da alle x_i im Polyeder liegen, müssen sie auch im Halbraum $\{x \in I\!R^n : a^T x \leq \beta\}$ sein. Aus $\beta = a^T v = a^T(\sum_{i=1}^{N} \lambda_i x_i) \leq \beta$ folgt $a^T x_i = \beta \, \forall i$, die x_i liegen also ebenfalls in der Seitenfläche. Da aber mindestens ein x_i von v verschieden ist, kann die Dimension nicht null und v damit keine Ecke sein.

- Sei andererseits v ein Punkt in P, für den es keine Konvexkombination von Punkten aus P gibt. Dieser Punkt muss in einer Seitenfläche liegen, ansonsten findet man ein ε, so daß $v + \varepsilon e$ und $v - \varepsilon e$ für einen beliebigen Vektor $e \neq v \in P$ noch in P liegen, also eine zulässige Konvexkombination existiert. Gibt es innerhalb der Seitenfläche aber keine Konvexkombination für v, so ist v auch extremaler Punkt dieses Polyeders und muß nach obiger Argumentation in einer Seitenfläche der Seitenfläche liegen. Die Dimension der Seitenfläche verringert sich dabei in jedem Schritt. Nach endlich vielen Schritten liegt v dann in einer Seitenfläche der Dimension null und ist damit eine Ecke.

■

Die Beschreibung einer Ecke des Polyeders als Seite der Dimension null oder als extremaler Punkt ist algorithmisch nicht zu gebrauchen. Auch wird nicht berücksichtigt, daß zwar jede Facette die Schnittmenge des Polyeders mit einer der linearen Ungleichungen ist,

andersherum aber keineswegs jede lineare Ungleichung auch eine Facette beschreiben muß — lineare Ungleichungen können schließlich redundant sein.
Abhilfe schafft hier der folgende Satz, der einen Zusammenhang zwischen einer Ecke und der Matrix A herstellt. Während die oben verwendete Form des Polyeders für geometrische Vorstellungen im $\mathbb{R}^2$ oder $\mathbb{R}^3$ besser geeignet war, greifen wir nun auf die sogenannte Standardform des Polyeders zurück: sei ab sofort $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$.

Satz 1.15 *Ein Punkt $v \in P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$ ist genau dann Ecke des Polyeders, wenn die Spalten von A, die den gleichen Index wie die positiven Komponenten von v haben, linear unabhängig sind.*

Beweis. Sei O.B.d.A. $\bar{v} := (v_1, \ldots, v_p) > 0$, $\tilde{v} := (v_{p+1}, \ldots, v_n) = 0$ und $v = (\bar{v}, \tilde{v})$.
Die Matrix A bestehe entsprechend aus zwei Teilmatrizen: $A = (\bar{A}\ \tilde{A})$, so daß $Av = \bar{A}\bar{v}$ gilt.

- Seien die zugehörigen Spalten von A linear abhängig, also $\bar{A}$ singulär.
  Dann existiert ein $w \neq 0 \in \mathbb{R}^p$ mit $\bar{A}w = 0$ und $\bar{A}(\bar{v} + \varepsilon w) = b$ für beliebiges $\varepsilon \in \mathbb{R}$. Wählt man nun ε so klein, daß sowohl $\bar{v} + \varepsilon w$ als auch $\bar{v} - \varepsilon w$ noch größer null sind, so liegen die beiden Punkte $\begin{pmatrix} \bar{v} \pm \varepsilon w \\ 0 \end{pmatrix}$ ebenfalls in P und ergeben als Konvexkombination mit $\lambda = \frac{1}{2}$ gerade wieder v, das also keine Ecke sein kann.
- Ist v kein extremaler Punkt von P, so existiert eine Konvexkombination $v = \sum_{i=1}^{N} \lambda_i x_i$ aus Vektoren $x_i \in P$. Da $x \geq 0$ für alle Vektoren in P gilt und die letzten $n-p$ Komponenten von v null sind, müssen auch die letzten $n-p$ Komponenten aller x_i null sein.
  Damit folgt für beliebiges i: $A(v - x_i) = \bar{A}(\underbrace{\bar{v} - \bar{x_i}}_{\neq 0}) = b - b = 0$. Also ist A singulär.

■

Bemerkungen:

- Bei Polyedern sind drei verschiedene Fälle zu unterscheiden. Die Anzahl der Variablen kann kleiner, gleich oder größer sein als die Anzahl der Ungleichungen bzw. Gleichungen. Dies spielt bei der theoretischen Untersuchung noch keine bedeutende Rolle, macht aber einen Unterschied bei der Implementierung, wie wir später genauer untersuchen werden.
- Treten nur Gleichungen auf und hat A nicht vollen Zeilenrang, so sind zwei Fälle möglich. Entweder ist die Lösungsmenge die leere Menge $P = \emptyset$, oder aber einige der Gleichungsbeschränkungen sind redundant und können, beispielsweise mit dem Gauß-Algorithmus, eliminiert werden. Hierbei muß natürlich auf Erhaltung von Strukturen, die man eventuell noch ausnutzen möchte, und auf die numerische Stabilität geachtet werden.

Im folgenden habe A den maximalen Zeilenrang m. Die Anzahl der Variablen ist hier mindestens genauso groß wie die Anzahl der Gleichungen, ansonsten wäre das System überbestimmt. Also $n \geq m$.

Um die Beschreibung einer Ecke durch die linear unabhängigen Spalten etwas besser fassen zu können, eignen sich die folgenden Definitionen von Basis und Basisvariablen.

Definition 1.16 *Ein geordnetes Paar $S = (B, N)$ von Mengen $B, N \subseteq \{1, \ldots, n\}$ heißt Spaltenbasis eines gegebenen LPs mit Nebenbedingungsmatrix $A \in \mathbb{R}^{m \times n}$, wenn folgendes gilt:*

1. $B \cup N = \{1, \ldots, n\}$
2. $B \cap N = \emptyset$
3. $|B| = m$
4. $A_{.B}$ *ist regulär*

Definition 1.17 *Eine Variable x_i heißt Basisvariable einer Spaltenbasis S, wenn $i \in B$. Sie heißt Nichtbasisvariable, wenn $i \in N$. Als Basis bezeichnet man je nach Zusammenhang die Indexmenge B oder die Menge der Basisvariablen $\{x_i : i \in B\}$.*
Den Vektor $x = \begin{pmatrix} A_{.B}^{-1} b \\ 0 \end{pmatrix}$ nennt man Basislösungsvektor. Gilt $x \geq 0$, so heißt x zulässiger Basislösungsvektor und S (primal) zulässige Spaltenbasis. Gilt $c^T x \geq c^T y \quad \forall\ y \in P$, so heißt x optimaler Basislösungsvektor und S (primal) optimale Spaltenbasis.

Wir sind unserem Ziel, der Überprüfung der Vermutung, daß sich die Optima von linearen Programmen in den Ecken des zugrundeliegenden Polyeders befinden, schon näher gekommen. Der Vektor x ist also genau dann eine Ecke, wenn er ein zulässiger Basislösungsvektor ist.
Wir brauchen nun noch den sogenannten Darstellungssatz, bevor wir den Abschnitt mit dem Fundamentalsatz der linearen Optimierung beschliessen können.

Definition 1.18 *Ein Vektor $d \neq 0 \in \mathbb{R}^n$ heißt Richtung in P, falls aus $x \in P$ für alle $\lambda \geq 0$ folgt, daß $x + \lambda d \in P$.*

Lemma 1.19 *Das Polyeder $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$ hat endlich viele Ecken.*

Beweis. Es gibt $\begin{pmatrix} n \\ m \end{pmatrix}$ Möglichkeiten, m Spalten aus n vorhandenen auszuwählen, die Anzahl der Ecken kann also maximal diesen Wert annehmen. ∎

Satz 1.20 *Jeder Punkt $x \in P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$ hat eine Darstellung*

$$x \quad = \quad d + \sum_{i \in I} \lambda_i \, v_i$$

wobei $\{v_i : i \in I\}$ die Menge der Ecken des Polyeders ist, d entweder eine Richtung in P oder aber $d = 0$ und alle $\lambda_i \geq 0$ mit $\sum_{i \in I} \lambda_i = 1$ für alle $i \in I$.
Insbesondere besitzt also jedes solches $P \neq \emptyset$ mindestens eine Ecke.

Einen Beweis für diesen Satz findet man beispielsweise bei [NKT89].

Satz 1.21 *Sei $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\} \neq \emptyset$ ein Polyeder. Dann gilt: Entweder hat die Zielfunktion $c^T x$ kein Maximum in P oder aber das Maximum wird in einer Ecke angenommen.*

Beweis. Unterscheide die beiden Fälle:

- es existiert eine Richtung d in P mit $c^T d > 0$:
  Dann wächst $c^T(x + \lambda d) = c^T x + \lambda c^T d \to \infty$ für $\lambda \to \infty$ unbeschränkt.
- es existiert keine Richtung d in P mit $c^T d > 0$:
  Das Maximum liegt im Inneren der konvexen Hülle $x = \sum_{i \in I} \lambda_i v_i$ mit
  $\lambda_i > 0$, $\sum_{i \in I} \lambda_i = 1$ und $\{v_i : i \in I\}$ die Menge der Ecken des Polyeders.
  Es gilt aber $c^T \sum_{i \in I} \lambda_i v_i \leq \sum_{i \in I} \lambda_i \max\{c^T v_i, i \in I\} = \max\{c^T v_i, i \in I\}$

■

1.2 Der Simplex-Algorithmus

Der vorangegangene Abschnitt brachte wichtige Ergebnisse. Das Optimum einer linearen Zielfunktion liegt, wenn es überhaupt existiert, in einer Ecke des zugrundeliegenden Polyeders. Ecken kann man mit Basislösungen identifizieren und berechnen und es gibt nur endlich viele Ecken.
Diese Erkenntnisse reichen schon aus um einen ersten Algorithmus zu formulieren, der ein lineares Programm löst: berechne der Reihe nach alle Ecken, prüfe auf Zulässigkeit und nimm den optimalen Wert als Lösung des LPs. Dieser Algorithmus terminiert zwar, hat aber eine nicht akzeptierbare exponentielle Laufzeit.

Die Idee des Simplex-Algorithmus beruht nun darauf, nicht alle Ecken zu berechnen, sondern einen bestimmten Weg zu verfolgen. Ausgehend von einer Ecke wählt man eine Nachbarecke, die eine Verbesserung des Zielfunktionswertes bringt. Setzt man Nicht-Degeneriertheit voraus — dies bedeutet, daß keine Komponente des Vektors x_B den Wert null hat — so stößt man nach endlich vielen Schritten auf eine Ecke, von der aus keine Verbesserung mehr möglich ist, oder auf eine Ecke, von der aus ein Schritt entlang einer Kante den Zielfunktionswert beliebig verbessert und damit die Unbeschränktheit des Problems zeigt. Dieses Verfahren kann ebenfalls exponentielle Laufzeit haben, wie konstruierte Beispiele zeigen, in denen sogar alle Ecken besucht werden müssen. Im statistischen Mittel ist die Laufzeit allerdings polynomial.
Der Simplex-Algorithmus sucht zuerst in der sogenannten Phase I eine zulässige Basislösung, also eine Ecke des Polyeders. In der Phase II wird dann solange die aktuelle Ecke gegen eine bessere zulässige Ecke ausgetauscht, bis keine Verbesserung mehr möglich ist. Eine Nachbarecke erhält man, indem man einen Index aus der Basis B gegen einen Index aus N tauscht und die neue zugehörige Basislösung x berechnet.

Im weiteren Verlauf werden noch ähnliche Algorithmen untersucht, die sich strukturell nicht sehr unterscheiden und auch sehr ähnlich in SoPlex implementiert sind. Um die Gemeinsamkeiten besser herausarbeiten zu können, bietet sich eine etwas andere Schreibweise an, als sie für gewöhnlich in der Literatur zu finden ist.
Seien der Zulässigkeitsvektor mit $f \in \mathbb{R}^m$ (entspricht hier x_B), wie gehabt die Matrix der

Nebenbedingungsbeschränkungen mit $A \in \mathbb{R}^{m\times n}$, die rechte Seite mit $b \in \mathbb{R}^m$, die zu optimierende Variable mit $x \in \mathbb{R}^n$ und der Zielfunktionsvektor mit $c \in \mathbb{R}^n$ bezeichnet und ein lineares Programm

$$\begin{array}{lrcl} \max & c^T x & & \\ \text{s.t.} & Ax & = & b \\ & x & \geq & 0 \end{array} \tag{1.2}$$

gegeben. Ferner treten ein Hilfs- und ein sogenannter pricing-Vektor auf, die hier mit h (in der Literatur häufig π oder y) bzw. g (sonst $\mu, \bar{c}$ oder d) bezeichnet werden:

Definition 1.22 *Der Vektor $h := A_{.B}^{-T} c_B \in \mathbb{R}^m$ wird copricing-Vektor genannt.*
$g := A^T h \in \mathbb{R}^n$ heißt pricing-Vektor.

Die Frage, wie man von einer gegebenen zulässigen Ecke, also einer zulässigen Spaltenbasis $S = (B, N)$ mit $x_i \geq 0$ für $i \in B$ und $x_i = 0$ für $i \in N$, zu einer anderen Ecke mit besserem Zielfunktionswert kommt, muß noch geklärt werden, bevor wir den Algorithmus formulieren können. Offensichtlich muß bei einem Basistausch eine Variable mit einem neuen Wert $x_{n_e} = \Phi \geq 0,\ n_e \in N$ in die Basis aufgenommen werden. Damit die Lösung zulässig bleibt, muß für $x = (x_B,\ x_N)$ gelten

$$\begin{pmatrix} A_{.B} & A_{.N} \\ 0 & I \end{pmatrix} (x + \Delta\tilde{x}) = \begin{pmatrix} b \\ \Phi\ \vec{e_e} \end{pmatrix} \quad \Longrightarrow \quad \begin{pmatrix} A_{.B} & A_{.N} \\ 0 & I \end{pmatrix} \Delta\tilde{x} = \begin{pmatrix} 0 \\ \Phi\ \vec{e_e} \end{pmatrix}$$
$$\Longrightarrow \quad \Delta\tilde{x} = \begin{pmatrix} A_{.B}^{-1} & -A_{.B}^{-1} A_{.N} \\ 0 & I \end{pmatrix} \begin{pmatrix} 0 \\ \Phi\ \vec{e_e} \end{pmatrix} = -\Phi \begin{pmatrix} A_{.B}^{-1} a_{n_e} \\ -\vec{e_e} \end{pmatrix} \tag{1.3}$$

Modifiziert man also x durch Aufaddieren von

$$-\Phi\Delta x := -\Phi \begin{pmatrix} A_{.B}^{-1} a_{n_e} \\ -\vec{e_e} \end{pmatrix}$$

so bleibt die Gleichung $Ax = b$ weiterhin gültig. Das Kriterium $x \geq 0$ kann man dann durch Wahl eines geeigneten $\Phi \geq 0$ erfüllen. Wählt man Φ so, daß für ein $x_i \in B$ gerade $x_i - \Phi\Delta x_i = 0$ gilt, so kann diese Variable aus der Basis entnommen und in die Menge der Nichtbasisvariablen aufgenommen werden.
Der Zielfunktionswert ändert sich bei einem solchen Update wie folgt:

$$c^T(x - \Phi\Delta x) = (c_B^T\ c_N^T) \left(\begin{pmatrix} x_B \\ x_N \end{pmatrix} - \Phi \begin{pmatrix} A_{.B}^{-1} a_{n_e} \\ -\vec{e_e} \end{pmatrix} \right) = c_B^T x_B - \Phi \left(\underbrace{c_B^T A_{.B}^{-1} a_{n_e}}_{g_{n_e}} - c_{n_e} \right) \tag{1.4}$$

Er ist also zulässig erhöhbar, wenn ein Index e existiert, so daß $g_{n_e} < c_{n_e}$. Der Simplexalgorithmus sieht, wenn man von einer gegebenen zulässigen Basislösung ausgeht, wie folgt aus:

Algorithmus 1.1 (Primaler Simplex für Spaltenbasis)
Gegeben sei eine zulässige Spaltenbasis $S = (B, N)$ von (1.2)

0. INIT

$$\begin{aligned} f &= A_{.B}^{-1} b \\ h &= A_{.B}^{-T} c_B \\ g &= A^T h \end{aligned}$$

1. PRICING

Ist $g_N \geq c_N$, so ist $x = \begin{pmatrix} f \\ 0 \end{pmatrix}$ optimal.
Sonst wähle $n_e \in N$ mit $g_{n_e} < c_{n_e}$

2.

$$\Delta f = A_{.B}^{-1} a_{n_e}$$

3. RATIO TEST

Ist $\Delta f \leq 0$, so ist das Programm unbeschränkt.
Sonst wähle $q \in \arg\min\{\frac{f_i}{\Delta f_i} : \Delta f_i > 0\}$

4.

$$\begin{aligned} \Delta h &= A_{.B}^{-T} \vec{e_q} \\ \Delta g &= A^T \Delta h \end{aligned}$$

5. UPDATE

$$\begin{aligned} B &= B \setminus \{j_q\} \cup \{n_e\} \\ N &= N \setminus \{n_e\} \cup \{j_q\} \\ \Theta &= \frac{c_{n_e} - g_{n_e}}{\Delta g_{n_e}} \\ \Phi &= \frac{f_q}{\Delta f_q} \\ f &= f - \Phi(\Delta f - \vec{e_q}) \\ h &= h + \Theta \Delta h \\ g &= g + \Theta \Delta g \end{aligned}$$

6. Gehe zu Schritt **1.**

Daß dieser Algorithmus wirklich die optimale Lösung von (1.2) liefert, wenn sie existiert, muß natürlich noch gezeigt werden. Dabei macht vor allem die sogenannte *Degeneriertheit* von Ecken Probleme — sie tritt auf, wenn nach einem Basiswechel der Wert Φ null ist, zu einer anderen Basis also der gleiche Basislösungsvektor f gehört. In anderen Worten: am Basislösungsvektor müssen sich mehr als n Stützhyperebenen des zugehörigen Polyeders schneiden. Nur so kann es nämlich sein, daß Algorithmus 1.1 nicht terminiert. Dieser degenerierte Fall und Lösungsstrategien werden im Abschnitt 1.5.4 besprochen. Bis dahin soll vorausgesetzt werden, daß es einen Algorithmus gibt, der das Problem meistert. Deshalb wird im weiteren Verlauf nur von partieller Korrektheit der Algorithmen gesprochen. Bisher ist nicht ganz klar, wie der Index n_e im pricing gewählt werden soll und woher man

eine zulässige Spaltenbasis bekommt, dies wird im Abschnitt 1.5 untersucht.

Hier soll nun die partielle Korrektheit von Algorithmus 1.1 gezeigt werden, was auf zwei Aspekte hinausläuft: der Vektor $x = (f, 0)$ muß eine zulässige Basislösung sein und bei Terminierung im Schritt 2 darf es keine bessere Lösung innerhalb des Polyeders mehr geben. Dazu sind vorab einige Lemmata nötig.
Ein ' kennzeichne im weiteren die Vektoren bzw. Mengen nach dem Update.

Satz 1.23 (Sherman, Morrison, Woodbury)
Sind $u, v \in \mathbb{R}^m$ und gilt $u^T v \neq -1$, so ist $E := I + uv^T$ invertierbar und es gilt

$$E^{-1} = I - \frac{uv^T}{1 + u^T v}$$

Beweis.

$$\begin{aligned} EE^{-1} &= \left(I + uv^T\right)\left(I - \frac{uv^T}{1+u^Tv}\right) = I + uv^T - \frac{uv^T + u(v^Tu)v^T}{1+u^Tv} \\ &= I + uv^T - \frac{(1+v^Tu)\,uv^T}{1+u^Tv} = I \end{aligned}$$

■

Lemma 1.24 *Nach einem Basiswechsel $B' = B \setminus \{j_q\} \cup \{n_e\}$ gilt*

$$\begin{aligned} A_{.B'} &= A_{.B} + (a_{n_e} - a_{j_q})\,\vec{e_q}^T \\ A^{-1}_{.B'} &= \left(I - \frac{(\Delta f - \vec{e_q})\vec{e_q}^T}{\Delta f_q}\right) A^{-1}_{.B} \end{aligned}$$

Beweis. Beim Basiswechsel wird gerade die q-te Spalte a_{j_q} entfernt und durch a_{n_e} ersetzt. Die Darstellung der Inversen ist eine direkte Folgerung der Sherman-Morrison-Formel:

$$\begin{aligned} A_{.B'} &= A_{.B} + (a_{n_e} - a_{j_q})\,\vec{e_q}^T = A_{.B}(I + (A^{-1}_{.B}a_{n_e} - A^{-1}_{.B}a_{j_q})\,\vec{e_q}^T) \\ &= A_{.B}\overbrace{(I + (\underbrace{\Delta f - \vec{e_q}}_{u})\,\underbrace{\vec{e_q}^T}_{v^T})}^{E} \end{aligned}$$

■

Lemma 1.25 *Ist $S = (B, N)$ eine zulässige Spaltenbasis, so gilt nach dem Update*

$$\begin{aligned} f' &= A^{-1}_{.B'}\,b \\ f' &\geq 0 \\ h' &= A^{-T}_{.B'}\,c_{B'} \\ g' &= A^T h' \end{aligned}$$

Beweis.

$$
\begin{aligned}
f' &= f - \Phi(\Delta f - \vec{e_q}) = f - \frac{f_q}{\Delta f_q}(\Delta f - \vec{e_q}) = A_{.B}^{-1}\, b - \vec{e_q}^{\,T} A_{.B}^{-1}\, b \frac{\Delta f - \vec{e_q}}{\Delta f_q} \\
&= A_{.B}^{-1}\, b - \frac{\Delta f - \vec{e_q}}{\Delta f_q} \vec{e_q}^{\,T} A_{.B}^{-1}\, b = \left(I - \frac{(\Delta f - \vec{e_q})\vec{e_q}^{\,T}}{\Delta f_q} \right) A_{.B}^{-1}\, b \overset{Lemma\ 1.24}{=} A_{.B'}^{-1}\, b \\
f_i' &= f_i - \Phi \Delta f_i \geq f_i - \frac{f_i}{\Delta f_i} \Delta f_i = 0 \quad \forall\, 1 \leq\ i \neq q\ \leq m, \quad f_q' = \Phi \geq 0 \\
h' &= h + \Theta \Delta h = A_{.B}^{-T}\, c_B + \frac{c_{n_e} - g_{n_e}}{\Delta g_{n_e}} A_{.B}^{-T}\, \vec{e_q} = A_{.B}^{-T} \left(c_B + \frac{c_{n_e} - \vec{e_{n_e}}^{\,T} A^T h}{\vec{e_{n_e}}^{\,T} A^T \Delta h}\, \vec{e_q} \right) \\
&= A_{.B}^{-T} \left(c_B + \frac{c_{n_e} - \vec{e_{n_e}}^{\,T} A^T A_{.B}^{-T} c_B}{\vec{e_{n_e}}^{\,T} A^T A_{.B}^{-T} \vec{e_q}}\, \vec{e_q} \right) = A_{.B}^{-T} \left(c_B + \frac{c_{n_e} - \Delta f^T c_B}{\Delta f_q}\, \vec{e_q} \right) \\
&= A_{.B}^{-T} \left(c_B + \left(\frac{c_{n_e}}{\Delta f_q} - \frac{\Delta f^T c_B}{\Delta f_q} + \underbrace{\frac{c_{j_q} - c_{j_q}}{\Delta f_q} + (c_{n_e} - c_{j_q}) - \frac{\Delta f_q}{\Delta f_q}(c_{n_e} - c_{j_q})}_{0} \right) \vec{e_q} \right) \\
&= A_{.B}^{-T} \left(c_B \underbrace{- \frac{\vec{e_q}(\Delta f - \vec{e_q})^T}{\Delta f_q} c_B}_{\left(-\frac{\Delta f^T c_B}{\Delta f_q} + \frac{c_{j_q}}{\Delta f_q} \right) \vec{e_q}} + (c_{n_e} - c_{j_q}) \vec{e_q} \underbrace{- \frac{\vec{e_q}(\Delta f - \vec{e_q})^T}{\Delta f_q} (c_{n_e} - c_{j_q}) \vec{e_q}}_{\left(\frac{c_{n_e}}{\Delta f_q} - \frac{c_{j_q}}{\Delta f_q} - \frac{\Delta f_q}{\Delta f_q}(c_{n_e} - c_{j_q}) \right) \vec{e_q}} \right) \\
&= A_{.B}^{-T} \left(I - \frac{(\Delta f - \vec{e_q})\vec{e_q}^{\,T}}{\Delta f_q} \right)^T \left(c_B + (c_{n_e} - c_{j_q}) \vec{e_q} \right) \overset{Lemma\ 1.24}{=} A_{.B'}^{-T}\, c_{B'} \\
g' &= g + \Theta \Delta g = A^T\, h + \Theta A^T \Delta h = A^T h'
\end{aligned}
$$

■

Lemma 1.26 *Mit einer gegebenen zulässigen Basislösung x von (1.2) läßt sich jedes $y \in P = \{x \in I\!R^n : Ax = b, x \geq 0\}$ darstellen als*

$$y = x + \sum_{i=1}^{n-m} y_{m+i}\, \eta_i \qquad \text{mit } y_j \geq 0 \ \ \forall\ m+1 \leq j \leq n$$

wobei $\eta_i = \begin{pmatrix} -A_{.B}^{-1} a_{n_i} \\ \vec{e_i} \end{pmatrix}$ *die $(m+i)$-te Spalte der Matrix* $\begin{pmatrix} A_{.B}^{-1} & -A_{.B}^{-1} A_{.N} \\ 0 & I \end{pmatrix}$ *ist.*

Beweis. Mit $y \in P$ gilt $Ay = b$ und $y = \begin{pmatrix} y_B \\ y_N \end{pmatrix} \geq 0$. Da x zulässige Basislösung ist, also $Ax = b$ und $x_N = 0$, folgt

$$\begin{pmatrix} A_{.B} & A_{.N} \\ 0 & I \end{pmatrix} (y - x) = \begin{pmatrix} 0 \\ y_N \end{pmatrix}$$

und damit

$$y = x + \begin{pmatrix} A_{.B}^{-1} & -A_{.B}^{-1}A_{.N} \\ 0 & I \end{pmatrix} \begin{pmatrix} 0 \\ y_N \end{pmatrix} = x + \sum_{i=1}^{n-m} y_{m+i}\, \eta_i$$

■

Lemma (1.26) sagt aus, daß jeder zulässige Punkt des Polyeders von jedem anderen Punkt des Polyeders, also insbesondere auch von jeder Ecke aus, durch eine positive Linearkombination der Kanten η_i erreichbar ist.
Nun haben wir alles zusammen um den abschliessenden Satz formulieren zu können.

Satz 1.27 *Terminiert Algorithmus 1.1 in Schritt 1, so ist die optimale Lösung von (1.2) gefunden. Terminiert er in Schritt 3, so ist (1.2) unbeschränkt. Terminiert er nicht, so werden degenerierte Updates mit* $\Phi = 0$ *ausgeführt (Kreiseln).*

Beweis.

- Der Algorithmus terminiert in Schritt 1
  Der Vektor $x := (f, 0)$ ist nach (1.25) eine zulässige Basislösung. Es gilt
  $g_N \geq c_N$, nach (1.26) und (1.4) ist keine Verbesserung des Zielfunktionswertes durch ein zulässiges Update Δx bzw. Δf und damit in P möglich.

- Der Algorithmus terminiert in Schritt 3
  Mit $\Delta f \leq 0$ ist $f - \Phi(\Delta f - \vec{e_q}) \geq 0 \ \forall\ \Phi \geq 0, q \in \{1, \ldots, m\}$ und damit für alle $\Phi \geq 0$ zulässig. Nach (1.4) wächst der Zielfunktionswert mit $\Phi \to \infty$ beliebig.

- Der Algorithmus terminiert nicht
  Es gibt maximal $\binom{n}{m}$ verschiedene Basen des LP. Da die Zielfunktion monoton steigt, können Basen nur dann ein zweites Mal besucht werden, wenn alle Updates dazwischen mit $\Phi = 0$ ausgeführt wurden.

■

1.3 Dualität

In diesem Abschnitt soll die Definition des dualen Programms motiviert und der Dualitätssatz der linearen Programmierung formuliert werden. Diese führen dann zum dualen Simplexalgorithmus, der algorithmische Unterschiede zum primalen aufweist, die kurz aufgezeigt werden. Die sogenannten dualen Variablen werden in einem speziellen Abschnitt untersucht.

1.3.1 Primales und Duales Programm

Gegeben sei das lineare Programm (1.2)

$$\begin{array}{lrcl} \max & c^T x & & \\ \text{s.t.} & Ax & = & b \\ & x & \geq & 0 \end{array}$$

Ist x eine zulässige Lösung des Programms, so gilt für alle Zeilen $A_{i\cdot}$ von A

$$A_{i\cdot}\, x \;=\; b_i$$

Jede Linearkombination der Gleichungen bleibt natürlich gültig:

$$\sum_i y_i A_{i\cdot}\, x \;=\; \sum_i y_i b_i$$

Wählt man die Gewichtungsfaktoren y_i der Gleichungen so, daß die neuen Koeffizienten der Variablen x_j in dieser Linearkombination größer gleich den c_j sind, so hat man für nichtnegative x mit dem Wert $y^T b$ auf der rechten Seite eine obere Schranke für das Maximum der Zielfunktion gefunden:

$$c^T x \;\leq\; y^T A x \;=\; y^T b \quad \forall\; x \in P$$

Der Wunsch, unter diesen Schranken die beste auszuwählen, führt schließlich zur Definition des dualen Programmes.

Definition 1.28 *Das lineare Programm*

$$\begin{array}{lrcl} \min & b^T y & & \\ \textit{s.t.} & A^T y & \geq & c \end{array} \tag{1.5}$$

wird das duale Programm zu (1.2) genannt. (1.2) heißt primales Programm.

Bemerkungen:

- Kann man die y_i derart wählen, daß

  $$A^T y \;=\; c$$

  gilt, so ist das Vorzeichen der Variablen x unerheblich. Anders formuliert: freie Variablen x führen zu Gleichungen in den Nebenbedingungen des dualen Programms.
- Gilt $Ax \leq b$, so muß $y \geq 0$ sein, da sich sonst Ungleichungen umdrehen würden und $y^T b$ keine Schranke mehr wäre. Für $Ax \geq b$ entsprechend $y \overset{!}{\leq} 0$.
- Das duale Programm des dualen Programms ist wieder das primale Programm (1.2). Dies sieht man, indem man

  $\begin{array}{|lrcl|} \hline \min & b^T y & & \\ \text{s.t.} & A^T y & \geq & c \\ \hline \end{array}$ äquivalent umformt in $\begin{array}{|lrcl|} \hline \max & (-b)^T y & & \\ \text{s.t.} & (-A)^T y & \leq & -c \\ \hline \end{array}$

  und zu diesem das duale Programm bildet.
- Wie in der Herleitung deutlich wurde, stellt der Funktionswert einer jeden zulässigen Lösung des Programms (1.5) eine obere Schranke des dazu dualen Programms dar. Man spricht hier von schwacher Dualität.

Den Zusammenhang zwischen zulässigen primalen und dualen Lösungen beschreibt der Dualitätssatz der linearen Programmierung von Gale, Kuhn und Tucker von 1951. Für den Beweis setzen wir die Existenz eines korrekten Algorithmus voraus.

Satz 1.29 *Seien* $m, n \in \mathbb{N}, 0 < m \leq n,\ c, x, x^* \in \mathbb{R}^n,\ b, y, y* \in \mathbb{R}^m$ *und* $A \in \mathbb{R}^{m \times n}$. *Dann gilt: Das primale Programm (1.2)*

$$\begin{array}{lrcl} \max & c^T x & & \\ s.t. & Ax & = & b \\ & x & \geq & 0 \end{array}$$

besitzt eine zulässige, optimale Lösung x^* *genau dann wenn das duale Programm (1.5)*

$$\begin{array}{lrcl} \min & b^T y & & \\ s.t. & A^T y & \geq & c \end{array}$$

eine zulässige, optimale Lösung y^* *besitzt. In dem Falle gilt*

$$c^T x^* \ = \ b^T y^*$$

Beweis. Ist S eine zulässige, optimale Spaltenbasis von (1.2) mit zugehörigem Lösungsvektor x^*, den man beispielsweise durch Algorithmus 1.1 erhalten hat, so ist der copricing-Vektor $h = A_B^{-T} c_B$ ein zulässiger, optimaler Lösungsvektor von (1.5):
Da S optimal ist, gilt $g_N \geq c_N$ und nach Konstruktion $g_B = c_B$. Also $A^T h = g \geq c$ und h ist zulässig.
Wegen $b^T h = b^T A_{.B}^{-T} c_B = c_B^T A_{.B}^{-1} b = c_B^T x_B^* = c^T x^*$ und der Beschränkung aller zulässigen dualen Lösungen durch jede zulässige primale (schwache Dualität), ist h auch optimal.
Die Umkehrung für das duale Programm gilt analog. ■

Löst man also das duale Programm (1.5) und erhält eine zulässige, optimale Lösung, so ist der Zielfunktionswert auch der optimale für Programm (1.2) und andersherum. Der optimale Lösungsvektor des dualen Programmes ist leicht aus einer optimalen, zulässigen primalen Basis zu bestimmen, es ist gerade der copricing-Vektor.
Ist eines der beiden Programme unbeschränkt, so kann das andere wegen der schwachen Dualität nur unzulässig sein. Es können aber auch beide Programme gleichzeitig unzulässig sein.
Den strukturellen Zusammenhang zwischen primalem und dualem Programm nutzt man nun beim dualen Algorithmus aus. Bei diesem wird nämlich nicht das duale Programm mit dem primalen Algorithmus gelöst, wie durchaus möglich (siehe 1.3.3), sondern ein modifizierter Algorithmus benutzt, der das gleiche Resultat liefert. Mit der

Definition 1.30 *Eine Spaltenbasis* S *heißt dual zulässig, wenn* $g \geq c$ *gilt. Sie heißt dual optimal, wenn* $f \geq 0$.

kann man eine weitere Folgerung der bisherigen Erkenntnisse formulieren.

Lemma 1.31 *Eine Spaltenbasis ist zulässige und optimale Lösung von (1.2) und (1.5), wenn sie sowohl primal als auch dual zulässig ist.*

Beweis. Einfache Folgerung der Sätze (1.27) und (1.29). ■

1.3.2 Der duale Algorithmus

Primale Zulässigkeit entspricht also in gewisser Weise dualer Optimalität und umgekehrt. Die Idee des dualen Algorithmus beruht nun darauf, daß man von dualer Zulässigkeit ausgeht und primale Zulässigkeit zu erreichen sucht. Daher ändern sich gerade die Rollen von g und h einerseits und f andererseits beim pricing und beim ratio test. Der duale Algorithmus sieht dann folgendermaßen aus.

Algorithmus 1.2 (Dualer Simplex für Spaltenbasis)
Gegeben seien eine optimale Spaltenbasis $S = (B, N)$ von (1.2)

0. INIT

$$\begin{aligned} f &= A_{.B}^{-1} b \\ h &= A_{.B}^{-T} c_B \\ g &= A^T h \end{aligned}$$

1. PRICING

Ist $f \geq 0$, so ist $x = \begin{pmatrix} f \\ 0 \end{pmatrix}$ optimal.
Sonst wähle $j_q \in B$ mit $f_q < 0$

2.

$$\begin{aligned} \Delta h &= A_{.B}^{-T} \vec{e_q} \\ \Delta g &= A^T \Delta h \end{aligned}$$

3. RATIO TEST

Ist $\Delta g \geq 0$, so ist das Programm unzulässig.
Sonst wähle $n_e \in \arg\min\{\frac{c_i - g_i}{\Delta g_i} : \Delta g_i < 0\}$

4.

$$\Delta f = A_{.B}^{-1} a_{n_e}$$

5. UPDATE

$$\begin{aligned} B &= B \setminus \{j_q\} \cup \{n_e\} \\ N &= N \setminus \{n_e\} \cup \{j_q\} \\ \Theta &= \frac{c_{n_e} - g_{n_e}}{\Delta g_{n_e}} \\ \Phi &= \frac{f_q}{\Delta f_q} \\ f &= f - \Phi(\Delta f - \vec{e_q}) \\ h &= h + \Theta \Delta h \\ g &= g + \Theta \Delta g \end{aligned}$$

6. Gehe zu Schritt **1.**

Lemma 1.32 *Ist $S = (B, N)$ eine dual zulässige Spaltenbasis, so gilt nach dem Update*

$$\begin{aligned} f^{'} &= A_{.B'}^{-1}\, b \\ h^{'} &= A_{.B'}^{-T}\, c_{B'} \\ g^{'} &= A^T h^{'} \\ g^{'} &\geq c \end{aligned}$$

Beweis. Die Formeln für $f^{'}, g^{'}$ und $h^{'}$ wurden schon in Lemma 1.25 gezeigt. Die Beibehaltung der dualen Zulässigkeit folgt aus

$$\begin{aligned} g_{B'}^{'} &= (A^T h^{'})_{B'} = (A^T A_{.B'}^{-T}\, c_{B'})_{B'} = c_{B'} \\ g_i^{'} &= g_i + \Theta \Delta g_i = g_i + \frac{c_{n_e} - g_{n_e}}{\Delta g_{n_e}} \Delta g_i \geq c_i \quad \forall\, i \in N^{'} \end{aligned}$$

■

Satz 1.33 *Terminiert Algorithmus 1.2 in Schritt 1, so ist die optimale Lösung von (1.2) gefunden. Terminiert er in Schritt 3, so ist (1.2) unzulässig. Terminiert er nicht, so werden degenerierte Updates mit $\Theta = 0$ ausgeführt (Kreiseln).*

Beweis.

- Der Algorithmus terminiert in Schritt 1
  Nach (1.32) und wegen $f \geq 0$ ist die Spaltenbasis B primal und dual zulässig.

- Der Algorithmus terminiert in Schritt 3
  Für $\Delta g \geq 0$ und beliebiges positives Θ bleibt die Spaltenbasis dual zulässig. Der duale Zielfunktionswert ändert sich wie folgt:

  $$b^T h^{'} = b^T (h + \Theta \Delta h) = b^T h + \Theta b^T A_{.B}^{-T} \vec{e_q} = b^T h + \Theta f_q$$

  Wegen $f_q < 0$ kann der Funktionswert also mit $\Theta \to \infty$ beliebig fallen, aus der schwachen Dualität folgt die Unzulässigkeit von Programm (1.2).

- Der Algorithmus terminiert nicht
  Es gibt maximal $\binom{n}{m}$ verschiedene Basen des primalen LPs. Da die duale Zielfunktion monoton fällt, können Basen nur dann ein zweites Mal besucht werden, wenn alle Updates dazwischen mit $\Theta = 0$ ausgeführt wurden.

■

1.3.3 Vorteile des dualen Algorithmus

Der duale Algorithmus wirkt sich so aus, als löste man das duale Programm mit dem Algorithmus 1.1. Da die Problemdimensionen n und m im allgemeinen unterschiedlich groß sind, vermutet man schnell, daß es hier einen Vorteil gibt. Das Lösen von linearen Gleichungssystemen nimmt den größten Anteil der benötigten Rechenzeit in Anspruch und die Basismatrix ist beim primalen Algorithmus im $\mathbb{R}^{m\times m}$, beim Lösen des dualen Programms dagegen im $\mathbb{R}^{n\times n}$. In der Praxis wird diese Feststellung aber dadurch relativiert, daß oft noch Schlupfvariablen eingeführt werden müssen um die Formen (1.2) bzw. (1.5) zu erhalten. Die Algorithmen 1.1 und 1.2 arbeiten beide auf einer Basis der Größe m. Will man mit einer Basis der Dimension n arbeiten, so sieht SoPlex die Zeilenbasis vor, vergleiche Abschnitt 1.8.

Der eigentliche Unterschied zwischen primalen und dualen Algorithmen liegt darin, daß erstere auf einer zulässigen, letztere dagegen auf einer optimalen Basis aufbauen und diese auch als Input erwartet wird (siehe Abschnitt 1.5.1). Dies macht einen entscheidenden Unterschied, wenn das LP modifiziert wird.
Gehen wir davon aus, daß wir eine primal zulässige Lösung haben, also $f \geq 0$ und $A_{.B}f = b$. Wird nun eine weitere Ungleichung hinzugefügt, so ändert sich $A_{.B}$ und damit auch der Lösungsvektor f — es muß eine neue zulässige Basis bestimmt werden um primale Zulässigkeit zu gewährleisten.
Haben wir eine dual zulässige, also primal optimale Lösung mit $g = A^T h \geq c$, läßt sich eine neue Ungleichung leicht zulässig einbauen. Man behält den dual zulässigen Lösungsvektor h bei und fügt eine neue Variable $h_{m+1} = 0$ hinzu, der Wert von g ändert sich nicht und man kann mit der bekannten Lösung weitermachen.

Anders sieht es aus, wenn eine neue Variable, also eine neue Spalte a_{n+1} und ein Wert c_{n+1}, hinzugefügt (oder entfernt) wird. Nun ändert sich $A_{.B}$ nicht und mit $f_{n+1} = 0$ bleibt die primale Zulässigkeit erhalten. Die duale Zulässigkeit dagegen nicht, das System $A^T h \geq c$ hat eine Zeile mehr und muß vollständig neu gelöst werden.

In der Praxis tritt der Fall, daß ein LP modifiziert und erneut gelöst wird, vor allem bei der Lösung von ganzzahligen linearen Programmen (IP, *integer programming*) oder gemischt ganzzahligen linearen Programmen (MIP, *mixed integer programming*) auf. IPs sind Probleme der Form (1.2), bei denen der zulässige Bereich auf ganze Zahlen $x \in \mathbb{Z}^n$ eingeschränkt ist, bei MIPs gilt dies nur für einen Teil der Variablen. Da die Variablenzahl oft sehr groß ist und Enumerierungen und auch kombinatorische Methoden im allgemeinen zu aufwendig sind, benutzt man spezielle Methoden, in denen Relaxierungen eine Rolle spielen um eine Lösung zu finden. Man nennt ein lineares Programm relaxiert, wenn die zulässige Menge eine Obermenge derer des Ausgangsprogrammes ist. In diesem Zusammenhang wird das Programm bis auf die Einschränkung der Ganzzahligkeit übernommen. Hat man eine optimale Lösung für das relaxierte Problem, so ist dies eine obere (bei Minimierung: untere) Schranke für die gesuchte ganzzahlige Lösung, da der zulässige Bereich größer ist.

Es gibt zwei erfolgreiche Ansätze, die durch das Hinzufügen von Zeilen bzw. Spalten zu einem Ausgangs-LP funktionieren: *branch-and-cut* und *branch-and-price*. Der erste fügt sogenannte Schnittebenen zu den Schranken der Variablen hinzu, bis eine gefundene optimale Lösung im zulässigen Bereich liegt. Der zweite wählt Variablen aus der großen Menge aller

Variablen aus und fügt diese zu einem kleinen Ausgangs-LP hinzu. Für einen Überblick und weitere Verweise eignen sich [NKT89], [JRT95] und [BJN93].

Der duale Algorithmus findet Anwendung bei branch-and-cut-Algorithmen und anderen Methoden, die eine gefundene Lösung als Startbasis eines durch Hinzufügen oder Entfernen von Nebenbedingungen modifizierten LPs benutzen. Der primale Algorithmus wird dagegen eher bei branch-and-price-Algorithmen verwendet.

1.3.4 Wirtschaftliche Deutung der dualen Variablen

Der copricing-Vektor h kann für bestimmte Probleme wirtschaftlich gedeutet werden. Er spielt in der primalen Zielfunktion genau dann eine Rolle, wenn die rechte Seite b nicht fix gegeben, sondern veränderlich ist, wie in folgendem linearen Programm.

$$\begin{array}{lrcl} \max & c^T x & & \\ \text{s.t.} & Ax & = & b + \Delta b \\ & x & \geq & 0 \end{array} \tag{1.6}$$

Interpretiert man b als vorhandene Ressourcen, wie dies in den meisten wirtschaftlichen Anwendungen der Fall ist, so gibt der copricing-Vektor an, wie lohnend eine Aufstockung bzw. Verringerung dieser Ressourcen sein würde. Um dies genauer auszudrücken, formulieren wir das folgende

Lemma 1.34 *x^* sei die nicht-degenerierte, zulässige und optimale Basislösung von (1.2) und $\Delta b \in \mathbb{R}^n$ ein Vektor mit $\Delta b \geq -(1 \dots 1)^T A_{\cdot B}\, \varepsilon$, wobei $\varepsilon := \min_{j \in B}\{x_j^*\}$ wegen der Nicht-Degeneriertheit echt größer null ist.*
Dann ist

$$x^* + \Delta x = \begin{pmatrix} A_{\cdot B}^{-1}(b + \Delta b) \\ 0 \end{pmatrix}$$

eine zulässige Basislösung

Beweis. $x^* + \Delta x$ ist Basislösung von 1.6.

Wegen $x^* + \Delta x = x^* + \begin{pmatrix} A_{\cdot B}^{-1}\Delta b \\ 0 \end{pmatrix} \geq x^* - \begin{pmatrix} \varepsilon \\ \vdots \\ \varepsilon \end{pmatrix} \geq 0$ ist sie auch zulässig.

■

Diese zulässige Basislösung $x + \Delta x$ ist für kleine Δb auch wieder die optimale Ecke — dies ist eine Folge davon, daß die Lösung x lokal differenzierbar von b abhängt. Der optimale Zielfunktionswert verändert sich damit natürlich auch. Wenn h der zugehörige copricing-Vektor ist, so gilt

$$c^T(x^* + \Delta x) = c^T x^* + c_B^T A_{\cdot B}^{-1} \Delta b = c^T x^* + \Delta b^T h$$

Dieses Einflusses auf die Zielfunktion wegen nennt man die h_i auch *Schattenpreise*. Ist h_i größer null, so ist es profitabel, die Ressource b_i zu erhöhen.

Bemerkung:

- Eine nette Überlegung ist in diesem Zusammenhang die Betrachtung der zugehörigen Einheiten, wie es sonst eher in der Physik üblich ist. Beispiel:

$$\begin{aligned}
[x_j] &= \text{Einheit von Produkt } j \\
[c_j] &= \frac{\text{Preis in Euro}}{\text{Einheit von Produkt } j} \\
[a_{ij}] &= \frac{\text{Einheit von Ressource } i}{\text{Einheit von Produkt } j} \\
[b_i] &= \text{Einheit von Ressource } i
\end{aligned}$$

  Damit folgt zwangsläufig

$$\begin{aligned}
[g_j] &= \frac{\text{Preis in Euro}}{\text{Einheit von Produkt } j} \\
[h_i] &= \frac{\text{Preis in Euro}}{\text{Einheit von Ressource } i}
\end{aligned}$$

  g_j gibt also den Zugewinn pro produzierter Einheit an, h_i beziffert den Zugewinn pro zusätzlicher Einheit der Ressource i.

1.4 Zusammenhang mit nichtlinearer Programmierung

Die lineare Programmierung ist nur ein Spezialfall der nichtlinearen Programmierung, für die besondere Methoden entwickelt wurden um auch große Dimensionen behandeln zu können. Es kann aber durchaus sinnvoll sein, lineare Programme aus dem Blickwinkel der nichtlinearen Programmierung zu betrachten um einige Zusammenhänge besser zu verstehen und allgemeingültige Ergebnisse zu deuten. Dies wollen wir in diesem Abschnitt unternehmen. Man beachte dabei, daß die Namenskonvention sich für diesen Abschnitt ändert, g und h sind nun z.B. Funktionen und nicht mehr pricing- bzw. copricing-Vektor.

Definition 1.35 *Ein allgemeines Optimierungsproblem hat die Form*

$$\begin{array}{lrcl}
\max & f(x) & & \\
s.t. & g(x) & = & 0 \\
 & h(x) & \geq & 0
\end{array}$$

mit $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$, $g : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ *und* $h : D \subseteq \mathbb{R}^n \to \mathbb{R}^k$.

Definition 1.36 *Ein* $x \in P := \{x \mid g(x) = 0, h(x) \geq 0\}$ *wird zulässig genannt. Gilt in einer Umgebung* $U \subseteq P$ *des Punktes* x *nun* $f(x) \geq f(y) \; \forall \, y \in U$, *so wird* x *als lokales Maximum bezeichnet.*

Ein wichtiges Ergebnis ist, daß es einen strukturellen Unterscheid macht, ob der Wert einer Funktion $h_i(x)$ null oder echt größer null ist, ähnlich wie bei der Bestimmung von Basis- und Nichtbasisvariablen, vergleiche Definition 1.17. Daher die folgende

Definition 1.37 *Für* $x \in S$ *definiere*

- $I(x) := \{i \mid h_i(x) = 0\} = \{i_1, \ldots, i_s\}$ *heißt Indexmenge der aktiven Ungleichungen.*
- $I^{\perp}(x) := \{i \mid h_i(x) > 0\}$ *heißt Indexmenge der inaktiven Ungleichungen.*

- $\tilde{h} := (h_{i_1}, \ldots, h_{i_s})^T \qquad \tilde{h} : D \subseteq I\!R^n \to I\!R^s, s \leq k$
- $\tilde{g} := (g, \tilde{h})^T \qquad \tilde{g} : D \subseteq I\!R^n \to I\!R^{m+s}$
- $x \in S$ *heißt regulär, wenn* $\tilde{g}_x(x) = \nabla \tilde{g}(x)^T$ *vollen Rang* $m + s \leq n$ *hat.*
- $L(x, \lambda, \mu) := f(x) - \lambda^T g(x) - \mu^T h(x)$ *mit* $\lambda \in I\!R^m, \mu \in I\!R^k$ *heißt Lagrangefunktion.*

Einer der elementarsten Sätze der nichtlinearen Optimierung ist der folgende. Einen Beweis und weiterführende Untersuchungen findet man beispielsweise in [NKT89].

Satz 1.38 (Karush-Kuhn-Tucker-Bedingungen)
Sei x^* *regulär und lokales Maximum. Dann existieren* μ^* *und* λ^* *mit*

- $\nabla_x L(x^*, \lambda^*, \mu^*) = 0$
- $\mu^* \geq 0$
- ${\mu^*}^T h(x^*) = 0$ *(Komplementaritätsbedingung)*
- $p^T \nabla_x^2 L(x^*, \lambda^*, \mu^*)\, p \leq 0 \quad \forall\, p \in T(x^*) := \{p \mid \tilde{g}_x(x^*)p = 0\}$

Betrachten wir den Spezialfall des Optimierungsproblems (1.2) mit den Abbildungen

- $f : D \subseteq I\!R^n \to I\!R \qquad x \to f(x) = c^T x$
- $g : D \subseteq I\!R^n \to I\!R^m \qquad x \to g(x) = Ax - b$
- $h : D \subseteq I\!R^n \to I\!R^k \qquad x \to h(x) = x$

Uns interessiert nun die Aussage des Satzes 1.38 über eine optimale zulässige Basislösung x. Zuerst prüfen wir nach, daß aus der Zulässigkeit von x die Regularität folgt. Es gilt $\nabla g = A^T$ und $\nabla h = I \Rightarrow \nabla \tilde{h} = W$, wobei $W \in I\!R^{n \times s}$ eine Matrix ist, deren i-te Zeile aus dem j-ten Einheitsvektor im $I\!R^s$ besteht, wenn $x_{i_j} = 0$ und aus dem Nullvektor, wenn $x_{i_j} > 0$. Es folgt$\nabla \tilde{g}(x) = \nabla(g, \tilde{h}) = (A^T \;\; W)$. Da $A_{.B}$ regulär ist und alle anderen Zeilen mit einem Einheitsvektor erweitert wurden, hat die Matrix vollen Rang und x ist regulär. Der Begriff der Optimalität überträgt sich ebenfalls, die Bedingungen von Satz 1.38 sind erfüllt und für die Basislösung x gilt:

- $\nabla_x L(x, \lambda, \mu) = 0 \implies \nabla_x(f(x) - \lambda^T g(x) - \mu^T h(x)) = 0 \implies c = A^T\lambda + \mu$
- $\mu \geq 0$
- $\mu_i > 0 \implies x_i = 0$ und $x_i > 0 \implies \mu_i = 0$
- Die hinreichende Krümmungsbedingung interessiert uns hier nicht weiter, die Hessematrix ist identisch null.

Die Gleichung $c = A^T\lambda + \mu$ sieht sehr bekannt aus, vergleiche das duale Programm (1.5):

$$\begin{array}{ll} \min & b^T y \\ \text{s.t.} & A^T y \;\; \geq \;\; c \end{array}$$

Offensichtlich entspricht das Lagrange-Multiplikator genannte λ dem copricing-Vektor y und μ gibt an, wieweit der pricing-Vektor $A^T y$ über c liegt. Die Ungleichung gibt gerade die Bedingung für die duale Zulässigkeit an.
Die Komplementaritätsbedingung (*englisch: complementary slackness*) hätte für die lineare Optimierung auch direkt gezeigt werden können. Diese Beziehung zwischen primalen und dualen Variablen ist für eine effiziente Implementierung des Simplexverfahrens sehr wichtig und wir werden uns in Abschnitt 2.3.3 noch einmal darauf beziehen.

Bemerkung:

- Ändert man das lineare Programm derart ab, daß x von zwei Seiten beschränkt ist, wie es in Abschnitt 1.6 untersucht wird, so ändert sich die Bedingung

$$h(x) = x \geq 0 \quad \text{in} \quad \begin{pmatrix} h_1(x) \\ h_2(x) \end{pmatrix} = \begin{pmatrix} x - L \\ U - x \end{pmatrix} \geq 0.$$

  In der Ableitung verschwindet der konstante Term aber wieder und nur das negative Vorzeichen bei $h_2(x)$ bleibt erhalten, so daß die dualen Variablen $\mu_2 \leq 0$ sein müssen. Dies entspricht den Beobachtungen in Abschnitt 1.3.

1.5 Besonderheiten bei der Implementierung

Wir haben zwei Algorithmen formuliert, die ein gegebenes lineares Programm lösen, und deren partielle Korrektheit gezeigt. Ein paar Fragen sind allerdings noch offen geblieben: Woher kommt die zulässige Startbasislösung, wie kann man Zyklen bei degenerierten Problemen verhindern, nach welchen Kriterien werden Indizes ausgewählt und wie werden die Updates der Matrix und der Vektoren realisiert? Diese Fragen sollen in diesem Abschnitt beantwortet werden. Die Darstellung lehnt sich dabei an [Wun96] an, da das Programmpaket SoPlex die Grundlage dieser Arbeit darstellt und die dort verwendeten Techniken besonders herausgestellt werden sollen. Dort sind auch Beweise und weiterführende Erläuterungen zu finden.

1.5.1 Bestimmung einer Startbasis

Voraussetzung für die vorgestellten Algorithmen ist eine gegebene primal oder dual zulässige Lösung. Wir betrachten das LP (1.2)

$$\begin{array}{lrcl} \max & c^T x & & \\ \text{s.t.} & Ax & = & b \\ & x & \geq & 0 \end{array}$$

und suchen zu diesem eine primal zulässige Lösung x. Es gelte $b \geq 0$, ansonsten multipliziere man die entsprechenden Zeilen mit -1. Um eine Lösung zu bestimmen, werden in der Standardliteratur (siehe beispielsweise [Pad99]) im allgemeinen zwei Verfahren angegeben, die beide auf der Einführung zusätzlicher Schlupfvariablen beruhen.

- **Big M Ansatz**
  Ein theoretisch aus der Codierungslänge C des Programms (1.2) herleitbares und genügend großes $M \in \mathbb{N}^+$ sei bestimmt. Damit formuliert man das lineare Programm

$$\begin{array}{lrcl} \max & c^T x - M 1^T s & & \\ \text{s.t.} & Ax + s & = & b \\ & x, s & \geq & 0 \end{array} \tag{1.7}$$

Für dieses Programm ist $x = 0, s = b$ eine zulässige Startbasis. Wendet man nun Algorithmus 1.1 an, so kann man aus dem Ergebnis Rückschlüsse auf das LP (1.2) ziehen. Ist (1.7) unbeschränkt, so auch (1.2). Enthält die optimale Lösung ein $s_i > 0$, so ist (1.2) wegen der Wahl von M unzulässig — eine zulässige Lösung x von (1.2) hätte mit $s = 0$ einen höheren Zielfunktionswert. Eine Lösung mit $s = 0$ ist schließlich eine zulässige und optimale Lösung auch von (1.2).

- **Zwei-Phasen-Methode**
  In einer sogenannten Phase I wird ein LP formuliert, für das eine zulässige Basis direkt angegeben werden kann. Aus der optimalen Lösung läßt sich dann eine zulässige Basis für das Ausgangs-LP konstruieren. Für die einfachste Variante betrachten wir das LP

$$\begin{array}{lrcl} \max & -1^T s & & \\ \text{s.t.} & Ax + s & = & b \\ & x, s & \geq & 0 \end{array} \tag{1.8}$$

  Wieder löst man das LP (1.8) mit Algorithmus 1.1. Eine optimale Basislösung mit $s \neq 0$ zeigt die Unzulässigkeit des Ausgangsprogramms. Ansonsten ist die optimale Basislösung x eine zulässige Lösung von (1.2). Das Lösen dieses Programms nennt man dann die Phase II.

Der *Big M Ansatz* wird in der Praxis nicht eingesetzt. Aufgrund der Größe von M kommt es zu numerischen Problemen (Auslöschung). Die angegebene Zwei-Phasen-Methode ist zwar numerisch stabil, benötigt aber viele Iterationen — die komplette Basis muß mindestens ausgetauscht werden. In der Praxis werden Verfahren eingesetzt, die direkt auf den Variablen des Ursprungs-LPs arbeiten. Hier werden zwei solche Methoden vorgestellt.

- **Composite Simplex-Verfahren** [Wol65]
  Gegeben sei eine beliebige Spaltenbasis S von (1.2). Das zugehörige Phase I-Problem wird definiert als

$$\begin{array}{lrcl} \max & c(B)^T x_B & & \\ \text{s.t.} & Ax & = & b \\ & x & \geq & 0 \end{array}$$

  wobei $c(B)_i = 1$ für $x_i < 0$ und $c(B)_i = 0$, sonst.
  Es gibt verschiedene Varianten des Verfahrens, in denen bestimmte Details des Algorithmus verändert werden.

- **SoPlex -Verfahren** [Wun96]
  S sei eine beliebige Basis von (1.2). Das zugehörige Phase I-Problem wird definiert als

$$\begin{array}{lllll} \max & c^T x & & \\ \text{s.t.} & Ax & = & b \\ & x_B & \geq & l \\ & x_N & \geq & 0 \end{array}$$

Die Schranke l_i nimmt den Wert x_i an, wenn die zugehörige Basisvariable negativ ist. Ansonsten ist $l_i = 0$. Damit ist die Basis S nach Konstruktion zulässig und der Algorithmus 1.1 kann gestartet werden (bzw. der noch aufzustellende Algorithmus für gleichungsbeschränkte Variablen). Terminiert dieser mit einer optimalen Lösung, so ist diese auch für (1.2) noch **dual** zulässig. Es kann nun also der duale Simplex mit der gefundenen Basis gestartet werden, um das eigentliche Problem zu lösen.
Entsprechend kann auch ein Phase I–LP konstruiert werden, das dual zulässig ist und dessen optimale Lösung man als primal zulässige Startbasis von (1.2) nutzt.
Es ist aber immer notwendig, sowohl den primalen als auch den dualen Algorithmus zu verwenden, nur die Reihenfolge ist anders.

Das erste Verfahren ist recht effizient und wurde nach [Bix94] — zumindest in älteren Versionen — in CPLEX eingesetzt. Das zweite Verfahren scheint dem ersten überlegen zu sein; es arbeitet nicht nur ausschließlich auf den richtigen Variablen, sondern berücksichtigt auch die Zielfunktion — oftmals terminiert die Phase I schon mit der optimalen Lösung.

Als Startbasis kann nun jede beliebige Basis dienen, beispielsweise $B = \{1, \ldots, m\}$, $N = \{m+1, \ldots n-m\}$. Es ist allerdings sinnvoll, schon hier eine Heuristik einzusetzen, die den Spalten bzw. Zeilen einen bestimmten Gewichtungsfaktor zuweist und die Startbasis anhand einer Sortierung dieser Faktoren auswählt. Die in SoPlex implementierte Crash-Prozedur berücksichtigt drei Aspekte: die Art der Schranken (vergleiche Abschnitt 1.6) — je geringer die Spanne $u - l$, umso größer die Wahrscheinlichkeit, daß die zugehörige Spalte oder Zeile in der Basis ist —, den möglichst kleinen Winkel der Normalenvektoren zum Zielfunktionsvektor und numerische Eigenschaften (Vektoren mit wenigen Nichtnulleinträgen sind oft numerisch gutartiger).

1.5.2 Pricing Strategien

Im Schritt 1 der Simplex Algorithmen soll ein Index ausgewählt werden, für den eine Ungleichung $f_q \geq 0$ bzw. $g_{n_e} - c_{n_e} \geq 0$ verletzt ist. Gibt es mehrere solcher Indizes, ist Spielraum für Varianten des Algorithmus, die sogenannten pricing-Strategien. Es ist keine Strategie bekannt, die für alle gegebenen LPs das beste Verhalten zeigt (und es darf auch bezweifelt werden, daß es eine solche gibt), daher bieten gute Implementierungen des Simplex-Verfahrens die Wahl zwischen mehreren pricing-Strategien. Einige sollen hier vorgestellt werden.

- **Most violation pricing**
  Diese pricing Methode ist die ursprünglich von Dantzig vorgeschlagene. Beim primalen Algorithmus wird

  $$n_e = \arg\min \{g_i - c_i \ : \ g_i < c_i\}$$

  gewählt. Beim dualen Algorithmus ist

  $$j_q = \arg\min \{f_i \ : \ f_i < 0\}$$

der die Basis verlassende Index. Da n_e den Index mit den niedrigsten reduzierten Kosten bezeichnet, ist die pricing-Strategie auch als reduced-cost pricing bekannt.

- **Partial pricing**
  Beim partial pricing wird die Menge der Variablen in mehrere kleinere Teilmengen aufgeteilt, so daß man beim pricing nur einen Teil der Variablen berücksichtigt. Erst wenn in der aktuellen Gruppe kein Index mehr gefunden werden kann, wechselt man zur nächsten Teilmenge. Auch der pricing-Vektor g wird dann natürlich nur partiell berechnet. Dieses Verfahren ist nur für den primalen Algorithmus einsetzbar, beim dualen Algorithmus wird der vollständige Vektor g benötigt.
  Wie groß die Teilmengen sind, in welcher Reihenfolge sie abgearbeitet werden und nach welchem Kriterium man den Index aus der Teilmenge bestimmt, läßt natürlich Spielraum für Varianten.

- **Multiple pricing**
  Beim multiple pricing werden alle wählbaren Indizes in eine Menge aufgenommen. Diese Menge bleibt nun über einige Iterationsschritte hinweg konstant und man spart sich die Arbeit der Neubestimmung in der Hoffnung, daß die Variablen auch im nächsten Schritt noch geeignete Kandidaten sind. Dies reduziert den Rechenaufwand zur Berechnung von g.

- **Partial multiple pricing**
  Das partial multiple pricing stellt eine Kombination aus partial pricing und multiple pricing dar. Es werden Teilmengen bestimmt und diese über einige Iterationsschritte hinweg konstant gehalten. In der Praxis wird zusätzlich ein Teil des Restmatrix durchsucht und geeignete Indizes werden der Restmenge hinzugefügt.

- **Steepest edge pricing**
  Die Variablen können verschieden skaliert sein, was sich auch auf die möglichen Richtungsvektoren $\Delta x^{(i)}$ auswirkt. Um trotzdem den Vektor auszuwählen, der den kleinsten Winkel zum Zielfunktionsvektor $-c$ aufweist, wird die Änderung des Zielfunktionswertes durch die Norm der Richtungsvektoren geteilt und der maximale Wert bestimmt:

$$n_e = \arg\max \left\{ \frac{|c^T \Delta x^{(i)}|}{\|\Delta x^{(i)}\|} \right\}$$

  Die Werte, über die das Maximum gebildet wird, sind dem Cosinus der Winkel, die zwischen den Richtungsvektoren und dem Vektor $-c$ liegen, proportional. Der Index n_e bezeichnet also den Richtungsvektor, der den kleinsten Winkel zu $-c$ hat.
  Das steepest edge pricing führt bei den meisten linearen Programmen zu einer erheblichen Reduzierung der Iterationen. Allerdings ist das Berechnen der Normen $\|\Delta x^{(i)}\|_2$ sehr teuer, daher wird es erst verwendet, seit Goldfarb und Reid Update-Formeln formulierten, siehe [GR77].

- **Devex pricing**
  Das devex pricing ist eine Approximation des steepest edge pricings. Die Werte

$$\frac{|c^T \Delta x^{(i)}|}{\|\Delta x^{(i)}\|}$$

werden nicht exakt bestimmt, sondern approximiert um den Rechenaufwand zu verringern. Das Verfahren geht auf Harris [Har73] zurück.

- **Weighted pricing**
  Beim weighted pricing werden am Programmanfang statisch Präferenzwerte vorgegeben, anhand derer die Auswahl erfolgt. Dies kann beispielsweise so erfolgen, wie es bei der Bestimmung der Startbasis gemacht wird: Berücksichtigung von Stabilität, Schranken und Winkel zum Zielfunktionsvektor.

- **Hybrid pricing**
  Die Nutzung mehrerer verschiedener pricing-Strategien bezeichnet man als hybrid pricing. Abhängig vom aktuellen LP, der Basisdarstellung und der Art des Algorithmus wird ein pricing Verfahren ausgewählt, von dem man sich besonders gute Fortschritte erhofft. Dies kann entweder statisch oder anhand von Laufzeitdaten geschehen, in SoPlex ist nur ein statisches hybrid pricing implementiert. Für den entfernenden Algorithmus wird das steepest edge pricing gewählt, für den einfügenden mit $m \ll n$ bzw. $n \ll m$ partial multiple pricing und devex pricing, wenn die Größen vergleichbar sind.

1.5.3 Ratio test Strategien, Stabilität

Führt man mathematische Berechnungen mit Hilfe von Computern durch, so wird man nur in den seltensten Fällen gänzlich fehlerfrei arbeiten. Im allgemeinen sind schon die Eingabedaten mit einem Fehler behaftet, der verschiedene Ursachen haben kann. So können die Daten aus unsicherer Quelle kommen, beispielsweise aus Messungen oder Schätzungen. Oder ein reeller Wert wie beispielsweise die Zahl π muß mit endlicher Gleitkommaarithmetik dargestellt und deswegen gerundet werden. Im Verlauf des Algorithmus können auch Fehler entstehen, beispielsweise durch Approximationen oder Rundungen.
Operiert man nun auf diesen fehlerbehafteten Daten, so bildet man auch den Fehler ab und erhält ein verfälschtes Resultat, man spricht hier von Fehlerfortpflanzung. In bestimmten Fällen kann die Verstärkung von Fehlern so gravierend sein, daß das erhaltene Ergebnis keine sinnvollen Aussagen mehr über die tatsächliche Lösung des Problems zuläßt.
Um untersuchen zu können, welche Rolle die Fehlerfortpflanzung im Simplexalgorithmus spielt und was bei einer stabilen Implementierung beachtet werden muß, sind einige grundlegene Definitionen nötig, die in jedem Einführungswerk in die Numerik zu finden sind, vergleiche z.B. [Sto94].

Gegeben sei ein Vektor $e \in I\!R^n$ mit Eingabedaten und eine Abbildung φ, die diesen auf einen Ergebnisvektor abbildet:

$$\varphi : e \to \varphi(e)$$

Definition 1.39 *Die (relative) Kondition κ eines Problems (φ, e) ist die kleinste positive Zahl κ, so daß*

$$\frac{\|\varphi(\tilde{e}) - \varphi(e)\|}{\|\varphi(e)\|} \leq \kappa \, \frac{\|\tilde{e} - e\|}{\|e\|} \qquad \text{für } \tilde{e} \to e.$$

Ist $\kappa \approx 1$, so heißt der Algorithmus bzw. das Problem gut konditioniert, für $\kappa \gg 1$ heißt es schlecht konditioniert. Die Kondition einer regulären Matrix wird definiert durch das

Produkt

$$\kappa(A) := \|A\| \, \|A^{-1}\|$$

Den relativen Fehler einer Größe x bezeichnet man mit

$$\varepsilon(x) = \frac{\|\Delta x\|}{\|x\|}$$

wobei Δx den absoluten Fehler angibt.

Die Definition der Matrixkondition führt zu folgendem Satz, mit dem der Einfluss der Fehler ΔA und Δb auf die Lösung x des linearen Gleichungssystems $Ax = b$ beschrieben wird.

Satz 1.40 *Gegeben sei ein lineares Gleichungssystem $Ax = b$ mit regulärer Matrix $A \in \mathbb{R}^{n \times n}$. Mit Fehlern ΔA und Δb in der Eingabe sei eine Lösung $\tilde{x} = x + \Delta x$ bestimmt, so daß $(A + \Delta A)(x + \Delta x) = b + \Delta b$ gilt. Falls $\kappa(A)\, \varepsilon(A) < 1$ ist, gilt dann*

$$\varepsilon(x) \leq \frac{\kappa(A)}{1 - \kappa(A)\, \varepsilon(A)} (\varepsilon(A) + \varepsilon(b))$$

Beweis.

$$\varepsilon(x) \quad = \quad \frac{\|\Delta x\|}{\|x\|} = \frac{\|A^{-1}\left((b + \Delta b) - Ax - \Delta A x - \Delta A \Delta x\right)\|}{\|x\|}$$

Mit $Ax = b$, der Verträglichkeit der Normen und der Dreiecksungleichung folgt

$$\frac{\|\Delta x\|}{\|x\|} \quad \leq \quad \frac{\|A^{-1}\| \, \left(\|\Delta b\| + \|\Delta A\| \, \|x\| + \|\Delta A\| \, \|\Delta x\|\right)\|}{\|x\|}$$

Durch Umstellen und Erweitern erhält man

$$(1 - \|A^{-1}\| \, \|\Delta A\|) \, \frac{\|\Delta x\|}{\|x\|} \quad \leq \quad \frac{\|A^{-1}\| \, \|\Delta b\| \, \|A\|}{\|x\| \, \|A\|} + \frac{\|A^{-1}\| \, \|\Delta A\| \, \|A\|}{\|A\|}$$

Nach Voraussetzung ist $\kappa(A)\, \varepsilon(A) = \|A^{-1}\| \, \|\Delta A\| < 1$, daher

$$\frac{\|\Delta x\|}{\|x\|} \quad \leq \quad \frac{1}{1 - \kappa(A)\, \varepsilon(A)} \, (\kappa(A)\, \varepsilon(b) + \kappa(A)\, \varepsilon(A))$$

■

Für gut konditionierte Matrizen gibt $\kappa(A)$ eine gute Approximation der Kondition des Problems, das lineare Gleichungssystem $Ax = b$ zu lösen, da für $\varepsilon(A) \ll 1$ gerade $\varepsilon(x) \leq \kappa(A)(\varepsilon(A) + \varepsilon(b))$ gilt.

Die Kondition gibt eine Eigenschaft an, die dem Problem anhaftet und unabhängig vom verwendeten Algorithmus ist. Um die Fehler zu untersuchen, die ein solcher produziert und

fortpflanzt, gebraucht man den Begriff der Stabilität. Es gibt zwei Ansätze zur Untersuchung der Stabilität von Algorithmen, die Vorwärts- und die Rückwärtsanalyse. Erstere untersucht, wie weit ein Resultat eines fehlerproduzierenden Algorithmus $\tilde{\varphi}(\tilde{e})$ vom korrekten Ergebnis $\varphi(\tilde{e})$ entfernt sein kann — die Größen der beiden Mengen werden miteinander verglichen, man spricht von einem stabilen Algorithmus, wenn das Verhältnis nahe bei eins liegt.
Letztere geht von einem Fehler im Resultat aus und modelliert die Entstehung dieses Fehlers durch einen Fehler in den Eingabedaten, der von einem korrekten Algorithmus fortgepflanzt wird. Es wird die Menge aller $\hat{e} \in I\!R^n$ gebildet, für die $\varphi(\hat{e}) = \tilde{\varphi}(\tilde{e})$ für ein fehlerbehaftetes $\tilde{e}$ gilt und die Größe der beiden Fehlermengen verglichen.
Wir werden nur die Rückwärtsanalyse verwenden, es folgt die formale Definition der Stabilität in diesem Sinne.

Definition 1.41 *Sei (φ, e) ein numerisches Problem und F ein Lösungsalgorithmus. Mit einer Fehlermenge $E \subset I\!R^n$ und einer Menge $\hat{E} := \{\hat{e} \ : \ \varphi(\hat{e}) = \tilde{\varphi}(\tilde{e}), \tilde{e} \in E, \tilde{\varphi} \in F\}$ heißt F stabil im Sinne der Rückwärtsanalyse, wenn*

$$\sup_{\hat{e}\in\hat{E},\tilde{e}\in E} \frac{\|\hat{e}-\tilde{e}\|}{\|\tilde{e}\|} \approx \sup_{\tilde{e}\in E} \frac{\|\tilde{e}-e\|}{\|e\|}$$

Wir wollen nun das Lösen von linearen Programmen mit dem Simplex Algorithmus betrachten. In einer Basis treten nur ganzzahlige Werte auf, hier kommt es also zu keinen Fehlern. Der Bereich, in dem am meisten Fehler mit den schwerwiegendsten Folgen auftreten, ist das Lösen der anfallenden linearen Gleichungssysteme. Wie diese auf stabile Art gelöst werden können, wird im Abschnitt 1.5.5 besprochen.
Beim Simplex Algorithmus werden laufend neue Basismatrizen bestimmt. Nach Satz 1.40 ist es sehr wichtig, daß sie dabei gut konditioniert bleiben. Dies wollen wir mit dem folgenden Satz untersuchen.

Satz 1.42 *Nach einem Basisupdate in Algorithmus 1.1 oder 1.2 gilt*

$$\kappa(A_{.B'}) \ \leq \ \frac{1}{|\Delta f_q|}\,(1+\|\Delta f\|)\,(1+\|\Delta f\|+|\Delta f_q|)\ \kappa(A_{.B})$$

Beweis. Nach Lemma 1.24 gilt

$$\begin{aligned} A_{.B'} &= A_{.B}\,V := A_{.B}\left(I + (A_{.B}^{-1}a_{n_e} - \vec{e_q})\ \vec{e_q}^{\,T}\right) \\ A_{.B'}^{-1} &= V^{-1}\,A_{.B}^{-1} := \left(I - \frac{(\Delta f - \vec{e_q})\vec{e_q}^{\,T}}{\Delta f_q}\right) A_{.B}^{-1} \end{aligned}$$

Wegen $\kappa(A_{.B'}) = \|A_{.B'}\|\ \|A_{.B'}^{-1}\| \leq \|A_{.B}\|\ \|A_{.B}^{-1}\|\ \|V\|\ \|V^{-1}\| = \kappa(A_{.B})\ \|V\|\ \|V^{-1}\|$ müssen $\|V\|$ und $\|V^{-1}\|$ abgeschätzt werden.

$$\begin{aligned} \|V\| &= \sup_{\|x\|=1} \|Vx\| \\ &= \sup_{\|x\|=1} \|\left(x + (A_{.B}^{-1}a_{n_e} - \vec{e_q})\ x_q\right)\| \end{aligned}$$

$$
\begin{aligned}
&\leq \sup_{\|x\|=1} \|x - x_q \vec{e_q}\| + \sup_{\|x\|=1} \|A_{.B}^{-1} a_{n_e}\| \\
&\leq 1 + \|A_{.B}^{-1} a_{n_e}\| = 1 + \|\Delta f\| \\
\|V^{-1}\| &= \sup_{\|x\|=1} \|V^{-1} x\| \\
&= \sup_{\|x\|=1} \| \left(x - \frac{(\Delta f - \vec{e_q}) x_q}{|\Delta f_q|} \right) \| \\
&\leq \sup_{\|x\|=1} \|x\| + \frac{1}{|\Delta f_q|} \left(\sup_{\|x\|=1} \|\Delta f\| + \sup_{\|x\|=1} \|x_q \vec{e_q})\| \right) \\
&\leq 1 + \frac{1}{|\Delta f_q|} \left(\|\Delta f\| + 1 \right) \\
&= \frac{1}{|\Delta f_q|} \left(|\Delta f_q| + \|\Delta f\| + 1 \right)
\end{aligned}
$$

■

Die Kondition der Basismatrizen ist also insbesondere abhängig vom Wert

$$\Delta f_q = \vec{e_q}^T A_{.B}^{-1} a_{n_e} = \vec{e_{n_e}}^T A^T A_{.B}^{-T} \vec{e_q} = \Delta g_{n_e}$$

Stabile Implementierungen des ratio tests zielen nun darauf ab, einen Index mit möglichst großem $|\Delta f_q|$ bzw. $|\Delta g_{n_e}|$ zu wählen um eine besser konditionierte neue Basismatrix zu erhalten. Mathematisch ist der die Basis verlassende Index eindeutig bestimmt bis auf Indizes, die gleichzeitig ihre Schranken erreichen — wird eine größere Schrittweite gewählt, verliert man schließlich die Zulässigkeit, auf der der Algorithmus arbeitet und die Monotonie der Zielfunktion ist nicht mehr gewährleistet. Vom numerischen Standpunkt her ist eine Relaxierung dieser Schranken allerdings erforderlich. Diese Idee wurde erstmals von Harris [Har73] formuliert.
Betrachten wir verschiedene Implementierungen des ratio tests für den Algorithmus 1.2. Die Änderungen für den primalen Algorithmus sind offensichtlich.

- **Normaler ratio test**
  Die Schrittweite Θ wird so bestimmt, daß die duale Zulässigkeit $g \geq c$ gewahrt bleibt:

$$\Theta = \min \left\{ \frac{c_i - g_i}{\Delta g_i} : \Delta g_i < 0 \right\}$$

- **"Textbook ratio test"**
  Der textbook ratio test läßt eine Verletzung der Schranken um einen Wert $\delta \geq 0$ zu: $g_i \geq c_i - \delta \quad \forall\, i$. Setzt man diese Schranke ein, so erhält man

$$
\begin{aligned}
n_e &\in \arg\min \left\{ \frac{c_i - g_i - \delta}{\Delta g_i} : \Delta g_i < 0 \right\} \\
\Theta &= \frac{c_{n_e} - g_{n_e}}{\Delta g_{n_e}}
\end{aligned}
$$

  Die Schrittweite wird genau berechnet um wieder eine Basis zu erhalten, nur die Auswahl des Index n_e unterscheidet sich durch die Relaxierung der Schranken. Der

Vorteil liegt darin, daß Komponenten mit betragsmäßig großem Δg_i den zusätzlichen Weg δ schneller zurücklegen, also bevorzugt ausgewählt werden. Dadurch ist die Kondition der nächsten Basismatrix im allgemeinen besser als bei der Wahl eines Index mit kleinerem Δg_i. Dafür gibt es allerdings zwei Nachteile:

1. Θ wird mit den ursprünglichen Schranken gebildet, daher kann $\Theta < 0$ gelten, was einem Rückschritt im Zielfunktionswert entspricht. Dadurch kann es zum Kreiseln kommen und der Algorithmus terminiert nicht. Das Problem wird in Abschnitt 1.5.4 wieder aufgegriffen.
2. Für die Indizes i kann nach einigen Iterationen $g_i = c_i - \delta$ gelten, dann gibt es keinen Vorteil mehr gegenüber dem mathematischen ratio test. Dieser Fall scheint aber nicht so häufig aufzutreten.

- **Harris ratio test**
  Der von Harris vorgeschlagene ratio test besteht aus zwei Phasen. In der ersten wird analog zum textbook ratio test eine Schrittweite

$$\Theta_{max} \quad = \quad \arg\min \left\{ \frac{c_i - g_i - \delta}{\Delta g_i} : \Delta g_i < 0 \right\}$$

  bestimmt, für die g die um δ relaxierte Schranke erfüllt. In der zweiten Phase wird nun aus allen Indizes, die zu einem $\Theta \leq \Theta_{max}$ führen, derjenige Index i ausgewählt, für den die Komponente $|\Delta g_i|$ maximal ist:

$$n_e \quad \in \quad \arg\max \left\{ |\Delta g_i| \ : \ \frac{c_i - g_i}{\Delta g_i} \leq \Theta_{max}, \Delta g_i < 0 \right\}$$

- **SoPlex ratio test**
  Der standardmäßig in SoPlex eingesetzte ratio test ist ebenfalls ein Zwei-Phasenverfahren. Es unterscheidet sich vom Harris ratio test nur in der ersten Phase dadurch, daß er mehrmalige Verschärfungen der Schranken um δ zuläßt. Setze

$$\delta_i \quad = \quad \begin{cases} \delta & \text{falls } g_i \geq c_i \\ c_i - g_i - \delta & \text{sonst} \end{cases}$$

  In der Phase I wird Θ_{max} bestimmt:

$$\Theta_{max} \quad = \quad \arg\min \left\{ \frac{c_i - g_i - \delta_i}{\Delta g_i} : \Delta g_i < 0 \right\}$$

  In der Phase II wird analog zum Harris Test der stabilste zulässige Index ausgewählt. Damit ist für $g_{n_e} < c_{n_e}$

$$g'_{n_e} \quad \geq \quad g_{n_e} - \delta$$

  die Schranken können also in jedem Schritt um einen zusätzlichen Betrag δ verletzt werden. Dies verhindert offensichtlich das oben geschilderte Problem Nummer zwei, verschärft dafür aber das erste. Da die gröbere Verletzung der Schranken einen substantiellen Rückgang in der Zielfunktion bewirken kann, **shiftet** man die Variablengrenzen auf den gerade aktuellen Wert. Dies wird genauer im Abschnitt 1.5.4

erklärt, da shifting ein probates Mittel gegen Degeneriertheit ist und auch bei der in Abschnitt 1.5.1 beschriebenen Zwei-Phasen Methode zur Bestimmung einer zulässigen und optimalen Basis eingesetzt wird.

Eine zusätzliche Stabilisierung wird durch die Einführung eines Parameters `minstab` erreicht, der einen minimalen Wert für $|\Delta f_q| = |\Delta g_{n_e}|$ vorgibt. Führt der ratio test zu einem Index, der die Ungleichung $|\Delta g_{n_e}| \geq$ `minstab` verletzt, wird kein Basisupdate ausgeführt um die Basismatrix gut konditioniert zu halten. Stattdessen wird das pricing erneut ausgeführt, wobei verhindert wird, daß der gleiche Index erneut ausgewählt wird. Die Parameter δ und `minstab` werden dynamisch angepaßt um der numerisch kritischen Situation in der aktuellen Ecke gerecht zu werden.

1.5.4 Degeneriertheit

Wie wir in Abschnitt 1.4 gesehen haben, bedingt die complementary slackness, daß entweder die primale oder die zugehörige duale Variable den Wert null annimmt. Allerdings ist es auch möglich, daß beide null sind. Dies entspricht einer zusätzlichen Hyperebene durch die primale oder duale Ecke, die nicht zu ihrer Bestimmung notwendig ist. Man spricht hier von Degeneriertheit.
Die Folge ist offensichtlich: wählt man gerade diese Hyperebene als neue Richtung des Updates aus, kann der Fall eintreten, daß nur ein Update mit $\Theta = 0$ bzw. $\Phi = 0$ möglich ist, da ansonsten die Zulässigkeit verloren ginge. In dem Fall bleibt der Lösungsvektor und damit der Zielfunktionswert gleich, während sich die Basismatrix ändert. Man kann Beispiele konstruieren, für die es eine Schleife solcher degenerierter Basiswechsel gibt, so daß die vorgestellten Algorithmen nicht terminieren. Man spricht hier vom Kreiseln.
In der Literatur wird zum Teil angeführt, daß es aufgrund von Rundungsfehlern extrem unwahrscheinlich sei, daß ein solcher Fall wirklich auftritt und man die Behandlung vernachlässigen könne. Dieses Argument gilt allerdings nicht für viele gut konditionierte Nebenbedingungsmatrizen von Problemen aus der kombinatorischen Optimierung, in denen nur die Werte $-1, 0$ und 1 vorkommen.
Als Abhilfe wird oft vorgeschlagen, die Eingabedaten bei Beginn des Algorithmus zufällig zu stören. Dies hat allerdings Nachteile. Zum einen kann die sparse-Struktur der Matrix verloren gehen und der Rechenaufwand pro Iteration erhöht sich. Zum zweiten kann man ja auch Glück haben — der Algorithmus muß nicht kreiseln, wenn es degenerierte Ecken gibt. In solchen Fällen erhöht man durch eine anfängliche Permutation nur die Anzahl der Iterationen, da man die degenerierte Ecke in eine Schar von Ecken aufspaltet, die unter Umständen alle abgearbeitet werden müssen.

Erfolgversprechender scheint der Ansatz zu sein, in einer Variablen mitzuführen, wieviele degenerierte Schritte es hintereinander gegeben hat und erst bei Überschreiten eines gewissen Wertes die betroffenen Schranken zu verändern. Für Algorithmus 1.2 bedeutet dies bei mehr als `maxcycle` degenerierten Schritten

$$c_i := c_i - \text{rand}(I) \iff g_i < c_i + \delta \text{ und } \Delta g_i < 0$$

rand(I) bezeichnet dabei eine Zufallszahl aus einem Intervall I. Bei SoPlex (vgl. Kapitel 2) wurde $I = [100\ \delta, 1000\ \delta]$, `maxcycle` $= 150$ und $\delta = 10^{-6}$ gewählt. Das Ändern der Schranken wird *shifting* genannt.

Die Frage ist natürlich, wie man die Lösung eines Problems mit geshifteten Schranken nutzen kann um die Lösung des Originalproblems zu bestimmen, diese stimmen ja im allgemeinen nicht überein. Dazu benutzt man die einfache Feststellung, daß die optimale Lösung des geshifteten Problems nach Lemma 1.31 primal (bzw. dual) zulässig ist. Die geshifteten Schranken haben keinen Einfluss auf diese Zulässigkeit, sie bleibt also bei Beibehaltung des Lösungsvektors und der Basis gültig, wenn die Änderungen an den Schranken rückgängig gemacht werden. Man startet den primalen (bzw. dualen) Algorithmus mit der gefundenen Lösung neu. Geometrisch interpretiert man das so, daß der gefundene Lösungsvektor aus der ursprünglichen zulässigen Menge herausgetreten ist, die Lösung aber optimal ist, also einen besseren Zielfunktionswert besitzt, als die optimale des ursprünglichen Problems. Handelt es sich um ein Phase-I-Problem (vergleiche Abschnitt 1.5.1), sind die Schranken sowieso schon geshiftet.
Dieses Vorgehen iteriert man so lange, bis man ein Problem ohne shifting lösen kann.

Für ein solches Vorgehen gibt es keinen Terminierungsbeweis. Allerdings sind in der Praxis keine Fälle aufgetreten, bei denen es zu unendlichen Wechseln zwischen den beiden Algorithmusarten gekommen wäre. Die zufällige Bestimmung der Schrankenmodifikation läßt auch keine konstruierten Gegenbeispiele zu.

1.5.5 Updates

Wie in Lemma 1.25 bewiesen wurde, entspricht das Aufaddieren der Updatevektoren $\Delta f, \Delta g$ und Δh der Lösung der linearen Gleichungssysteme mit neuer Basis. Der Updatevektor der primalen oder dualen Variablen wird für die Durchführung des ratio tests benötigt und muß von daher sowieso berechnet werden. Der andere wird aber eigentlich nicht benötigt. Anstatt Δf zu berechnen und mit der gefundenen Schrittweite zu f zu addieren, könnte man auch das Gleichungssystem $f = A_{\cdot B'}^{-1} b$ lösen. Dies ist allerdings mit Ausnahme des partial multiple pricing mit einem größeren Rechenaufwand verbunden.

Die Nebenbedingungsmatrizen von linearen Programmen sind oftmals dünnbesetzt (sparse), was bedeutet, daß die Anzahl der Einträge von Nullen die von Nichtnullelementen (NNE) deutlich überwiegt. Wie in den folgenden Abschnitten ausgeführt wird, ist die Einführung von Schlupfvariablen nötig um Ungleichungsbeschränkungen behandeln zu können. Diese erweitern die Basismatrix um Einheitsvektoren $a_i = \vec{e_i}$ und verstärken daher noch die Dünnbesetztheit. Die dünnbesetzen Spalten- und Zeilenvektoren führen zu einer vereinfachten Berechnung der Updatevektoren. $\Delta f = A_{\cdot B}^{-1} a_{n_e}$ entspricht für in die Basis eintretende Schlupfvariablen gerade der i-ten Spalte von $A_{\cdot B}^{-1}$. Diese Matrix wird allerdings nicht berechnet, es gibt nur eine Lösungsvorschrift zum Lösen der Gleichungssysteme, beispielsweise durch Vorwärts- und Rückwärtssubstitution bei einer sogenannten LU-Zerlegung der Basismatrix. Doch fallen weniger Rechnungen an, wenn die rechte Seite aus einem dünnbesetzten Vektor a_{n_e} anstelle eines dichtbesetzten Vektors b besteht.
Aus diesem Grund wird in den meisten effektiven Implementierungen ein Update der Variablen gebildet.

Um ein Gleichungssystem $Ax = b$ zu lösen, wie es in jedem Durchlauf des Simplexalgorithmus auftritt, gibt es verschiedenste Ansätze und Methoden. Iterative Methoden verbessern eine Approximation des Lösungsvektors iterativ, bis eine bestimmte Genauigkeit erreicht wird. Direkte Methoden transformieren die Basismatrix in ein Produkt von Matrizen, mit

deren Hilfe das Gleichungssystem direkt gelöst werden kann. Für diese Methoden existieren Updateformeln, die mehr oder weniger direkt auf Lemma 1.24 basieren. Die Speicherung und Multiplikation der Updatematrizen benötigt allerdings zusätzliche Zeit und verschlechtert die Kondition des Matrixprodukts, daher wird nach einer bestimmten Anzahl von Schritten wieder eine neue Produktform berechnet.
Eine Beschreibung möglicher Methoden, insbesondere der in Simplex Algorithmen hauptsächlich eingesetzten direkten, findet man beispielsweise bei [BG69], [DER89], [FT72], [GMS87] oder [Suh89].

1.6 Der Simplex-Algorithmus bei beschränkten Variablen

Nach Abschnitt 1.1.1 sind die verschiedenen Darstellungsformen von linearen Programmen einander mathematisch äquivalent. So ist beispielsweise ein gegebenes Programm

$$\begin{array}{lccccc} \max & c'^T x' & & & & \\ \text{s.t.} & l'' & \leq & A'x' & \leq u'' \\ & l' & \leq & x' & \leq u' \end{array} \tag{1.9}$$

mit Schlupfvariablen s_i leicht in die Form

$$\begin{array}{lrcl} \max & c'^T x' & & \\ \text{s.t.} & A'x' - s_1 & = & l'' \\ & A'x' + s_2 & = & u'' \\ & Ix' + s_3 & = & u' \\ & x' \geq l' & & s_i \geq 0 \end{array}$$

überführbar und kann nach Umbenennung von A und x mit Algorithmus 1.1 gelöst werden. Algorithmisch ist diese Lösungsmethode allerdings nicht allzu geschickt, da die neue Matrix

$$A = \begin{pmatrix} A' & -I & 0 & 0 \\ A' & 0 & I & 0 \\ I & 0 & 0 & I \end{pmatrix} \in \mathbb{R}^{(2m+n)\times(2m+2n)}$$

ziemlich groß ist, die Berechnung der Inversen beziehungsweise der Updates in jedem Simplexschritt also sehr teuer wird. Stattdessen formuliert man (1.9) um in

$$\begin{array}{lccccccc} \max & c^T x & & & & & & \\ \text{s.t.} & & & A'x' & - & s & = & 0 \\ & l' & \leq & x' & & & \leq & u' \\ & l'' & \leq & & & s & \leq & u'' \end{array}$$

und definiert $A = (A' \ -I), c = \begin{pmatrix} c' \\ 0 \end{pmatrix}, x = \begin{pmatrix} x' \\ s \end{pmatrix}, L = \begin{pmatrix} l' \\ l'' \end{pmatrix}, U = \begin{pmatrix} u' \\ u'' \end{pmatrix}$.
Man erhält dadurch ein LP, das nur noch Schlupfvariablen für x benötigt und das das *zu (1.9) gehörende Spalten-LP* genannt wird:

$$\begin{array}{lccccc} \max & c^T x & & & & \\ \text{s.t.} & & & Ax & = & 0 \\ & L & \leq & x & \leq & U \end{array} \tag{1.10}$$

Optimale zulässige Lösungen x' von (1.9) und (x', s) von (1.10) lassen sich durch einfache Transformationen $x = (x', A'x')$ bzw. $x = x'$ zu einer optimalen zulässigen Lösung des anderen Programms mit gleichem Zielfunktionswert machen.

Die Anzahl der Variablen ist von n auf $n + m$ angewachsen. Allerdings können die m Schlupfvariablen s algorithmisch anders behandelt werden als normale Variablen. Dies wird in Abschnitt 2.3.3 genauer behandelt. Die Dimensionen sind nun $A \in \mathbb{R}^{m\times(n+m)}$ und $x, c, L, U \in \mathbb{R}^{n+m}$.

Der Vektor x ist immer noch durch zwei Ungleichungen beschränkt. Die hierfür eigentlich notwendigen Schlupfvariablen werden komplett eingespart durch eine implizite Verwaltung. Der Wert einer solchen Schlupvariablen kann jederzeit aus dem aktuellen Wert von x abgeleitet werden, daher wird er gar nicht erst gespeichert. Für den Algorithmus ist er nicht weiter wichtig, da er nicht in die Zielfunktion eingeht — entscheidend ist nur, ob die zugehörige Variable x_i von den beiden Schranken L_i und U_i verschieden ist. Daher liegt die Idee nahe, einfach mitzuführen, welche Variable aktuell welchen Status hat. Der Basisvektor x_B wird explizit gespeichert — x_N dagegen, nur wenn es notwendig ist, aus dem Status der Variablen und den jeweiligen Schranken bestimmt.

Welche der beiden Schranken erreicht wurde, ist allerdings entscheidend, da der Wert einer Nichtbasisvariablen, die an einer ihrer Schranken festgehalten wird, in die Berechnung von $f = x_B$ eingeht:

$$A_{\cdot B} f \quad = \quad -A_{\cdot N} x_N$$

Außerdem ist die Interpretation der reduzierten Kosten $g - c$ anders: Konnte eine Nichtbasisvariable von null aus nur erhöht werden um in den zulässigen Bereich zu gelangen, ist nun auch eine Reduzierung einer an der oberen Schranke festgehaltenen Variablen möglich. Das gewünschte Vorzeichen um eine Erhöhung des Zielfunktionswertes zu erlangen, dreht sich dann natürlich genau um: Setze $x_{n_e}(\Phi) = L_{n_e} + \Phi$ mit $\Phi \geq 0$ oder $x_{n_e}(\Phi) = U_{n_e} + \Phi$ mit $\Phi \leq 0$, vergleiche Seite 15. Dies bewirkt

$$c^T(x - \Phi\Delta x) = (c_B^T\ c_N^T)\left(\begin{pmatrix} x_B \\ x_N \end{pmatrix} - \Phi \begin{pmatrix} A_{\cdot B}^{-1} a_{n_e} \\ -\vec{e_e} \end{pmatrix}\right) = c^T x - \Phi\,(g_{n_e} - c_{n_e})$$

Die Auswirkungen sind in Tabelle 1.1 zusammengefasst.

	$x_{n_e} = L_{n_e},\ \Phi \geq 0$	$x_{n_e} = U_{n_e},\ \Phi \leq 0$
$g_{n_e} > c_{n_e}$	Zielfunktion fällt	Zielfunktion wächst
$g_{n_e} < c_{n_e}$	Zielfunktion wächst	Zielfunktion fällt

Tabelle 1.1: Zusammenhang zwischen g und c

Um den Status der einzelnen Variablen unterscheiden zu können, transformiert man die Indexmenge N in Mengen N_l und N_u, in der die Indizes derjenigen Variablen enthalten sind, deren Wert der unteren (lower) bzw. oberen (upper) Schranke entspricht. Da auch fixierte ($L_i = U_i$) und freie ($L_i = -\infty$ und $U_i = \infty$) Variablen erlaubt sind und

für diese unnötige Berechnungen ausgelassen werden können, werden zusätzliche Mengen N_x und N_f eingeführt, so daß die Nichtbasisvariablenmenge dargestellt werden kann als $N = N_f \uplus N_x \uplus N_l \uplus N_u$. Für die Basisvariablen ist eine solche Unterteilung eigentlich nicht nötig. Sie bietet allerdings einige algorithmische Vorteile, so daß auch B weiter untergliedert wird. Die Darstellung soll ähnlich Definition 1.17 formalisiert werden.

Definition 1.43 *Ein geordnetes Paar $S = (B, N)$ von Mengen $N = N_f \uplus N_x \uplus N_l \uplus N_u$, $B = B_f \uplus B_x \uplus B_l \uplus B_u \uplus B_n$ heißt Spaltenbasis von LP 1.10, wenn folgendes gilt:*

1. $B \cup N = \{1, \ldots, n+m\}$
2. $B_f, B_l, B_u, B_b, B_n, N_f, N_l, N_u$ *und* N_x *sind paarweise disjunkt*
3. $|B| = m$
4. $A_{.B}$ *ist regulär*
5. $$\begin{array}{lcccl} i \in N_l \cup B_u & \Rightarrow & -\infty < L_i & \neq & U_i = \infty \\ i \in N_u \cup B_l & \Rightarrow & -\infty = L_i & \neq & U_i < \infty \\ i \in B_b & \Rightarrow & -\infty < L_i & \neq & U_i < \infty \\ i \in N_f \cup B_n & \Rightarrow & -\infty = L_i & \textit{und} & U_i = \infty \\ i \in N_x \cup B_f & \Rightarrow & -\infty < L_i & = & U_i < \infty \end{array}$$

Neu in der Definition sind neben der geänderten Dimension die Spezifizierungen der Nichtbasis- bzw. Basisvariablenmengen. Die Abkürzungen bedeuten **l**ower, **u**pper, **f**ree, fi**x**ed, **b**oth, und **n**ot defined. Im wesentlichen handelt es sich hierbei um die Unterscheidung zwischen N_u, N_l und B, die anderen Mengen sind lediglich für Ausnahmefälle (Schranken unendlich, Variablen fixiert) und für Wechsel von B nach N und umgekehrt gedacht. Dies wird im Abschnitt 2.3.3 etwas deutlicher gemacht.

Definition 1.44 *Das Residuum um das die rechte Seite bei der Berechnung von f modifiziert werden muß, ist der wie folgt definierte Vektor $R \in \mathbb{R}^{n+m}$.*

$$R_i = \begin{cases} 0 & i \in N_f \\ L_i & i \in N_l \cup N_x \\ U_i & i \in N_u \\ 0 & i \in B \end{cases}$$

Definition 1.17 muß entsprechend angepaßt werden:

Definition 1.45 *Eine Variable x_i heißt Basisvariable einer Spaltenbasis S, wenn $i \in B$. Sie heißt Nichtbasisvariable, wenn $i \in N$. Als Basis bezeichnet man je nach Zusammenhang die Indexmenge B oder die Menge der Basisvariablen $\{x_i : i \in B\}$.*
Den Vektor

$$x = \begin{pmatrix} -A_{.B}^{-1} A R \\ 0 \end{pmatrix} + R$$

nennt man Basislösungsvektor. Gilt $L \leq x \leq U$, so heißt x zulässiger Basislösungsvektor und S primal zulässige Spaltenbasis. Gilt $c^T x \geq c^T y \ \forall \ y \in P$, so heißt x optimaler Basislösungsvektor und S primal optimale Spaltenbasis.

Um der veränderten Situation für den pricing-Vektor g Rechnung zu tragen, werden für ihn Schranken $l, u \in \mathbb{R}^{n+m}$ definiert. Durch sie ist es einfacher abzufragen, ob duale Zulässigkeit vorliegt oder nicht.

Definition 1.46 *Die Vektoren l und u bezeichnen die untere respektive obere Schranke für den pricing-Vektor g.*

$$l_i = \begin{cases} -\infty & \text{wenn } i \in N_u \\ c_i & \text{wenn } i \in N_l \\ -\infty & \text{wenn } i \in N_x \\ c_i & \text{sonst} \end{cases} \qquad u_i = \begin{cases} c_i & \text{wenn } i \in N_u \\ \infty & \text{wenn } i \in N_l \\ \infty & \text{wenn } i \in N_x \\ c_i & \text{sonst} \end{cases}$$

Eine Spaltenbasis heißt dual zulässig genau dann wenn $l \leq g \leq u$.

Analog zu den vorigen Abschnitten formulieren wir den folgenden Satz.

Satz 1.47 *Eine Spaltenbasis S ist zulässige und optimale Lösung von (1.10), wenn sie sowohl primal als auch dual zulässig ist.*

Beweis.
Zulässigkeit entspricht per Definition primaler Zulässigkeit. Sei x eine primal und dual zulässige Basislösung mit Zielfunktionswert $c^T x$ und pricing-Vektor $l \leq g \leq u$. Angenommen, dieses x sei nicht die optimale Lösung, so muß es eine Nichtbasisvariable x_{n_e} geben, die den Zielfunktionswert bei Eintritt in die Basis erhöht. Dieser beträgt $c^T x - \Phi(g_{n_e} - c_{n_e})$.

- Nach Tabelle 1.1 steigt er für $n_e \in N_l$ bei $g_{n_e} < c_{n_e}$ und für $n_e \in N_u$ bei $g_{n_e} > c_{n_e}$. Für eine dual zulässige Spaltenbasis ist also keine Erhöhung möglich.
- $n_e \in N_x \;\Rightarrow\; \Phi \stackrel{!}{=} 0 \;\Rightarrow$ Zielfunktion konstant
- $n_e \in N_f \;\Rightarrow\; g_{n_e} = c_{n_e} \;\Rightarrow$ Zielfunktion konstant

■

Mit diesen Vorüberlegungen ist es uns nun möglich, einen Algorithmus anzugeben, der die Schranken von (1.9) berücksichtigt. Dies wird hier nur exemplarisch für den dualen Algorithmus gemacht, da dieser im Blickpunkt dieser Arbeit steht.

Algorithmus 1.3 (Dualer Simplex, Spaltenbasis, gleichungsbeschränktes LP)
Sei S eine dual zulässige Spaltenbasis von (1.10)

0. INIT

Setze
Initialisiere R, l, u gemäß Definition.

$$\begin{aligned} f &= A_{\cdot B}^{-1}(0 - AR) \\ h &= A_{\cdot B}^{-T} c_B \\ g &= A^T h \end{aligned}$$

1. PRICING

Ist $L_B \leq f \leq U_B$, so ist $x = \begin{pmatrix} f \\ 0 \end{pmatrix} + R$ optimal.

Sonst wähle $j_q \in B$ mit
- Fall a) $f_q > U_{j_q}$
- Fall b) $f_q < L_{j_q}$

2.

Setze

$$\begin{aligned} \Delta h &= A_{\cdot B}^{-T} \vec{e_q} \\ \Delta g &= A^T \Delta h \end{aligned}$$

Fall a) $\Theta_0 = -\infty$, $\rho = U_{j_q}$, $l_{j_q} = -\infty$
Fall b) $\Theta_0 = +\infty$, $\rho = L_{j_q}$, $u_{j_q} = \infty$

3. RATIO TEST

Falls $\Theta_0 > 0$ setze

$$\begin{aligned} \Theta_+ &= \tfrac{u_{n_{e_+}} - g_{n_{e_+}}}{\Delta g_{n_{e_+}}} \text{ mit } n_{e_+} \in \arg\min_i \{\tfrac{u_i - g_i}{\Delta g_i} : \Delta g_i > 0\} \\ \Theta_- &= \tfrac{l_{n_{e_-}} - g_{n_{e_-}}}{\Delta g_{n_{e_-}}} \text{ mit } n_{e_-} \in \arg\min_i \{\tfrac{l_i - g_i}{\Delta g_i} : \Delta g_i < 0\} \\ \Theta &= \min\{\Theta_+, \Theta_-, \Theta_0\} \end{aligned}$$

Falls $\Theta_0 < 0$ setze

$$\begin{aligned} \Theta_+ &= \tfrac{l_{n_{e_+}} - g_{n_{e_+}}}{\Delta g_{n_{e_+}}} \text{ mit } n_{e_+} \in \arg\max_i \{\tfrac{l_i - g_i}{\Delta g_i} : \Delta g_i > 0\} \\ \Theta_- &= \tfrac{u_{n_{e_-}} - g_{n_{e_-}}}{\Delta g_{n_{e_-}}} \text{ mit } n_{e_-} \in \arg\max_i \{\tfrac{u_i - g_i}{\Delta g_i} : \Delta g_i < 0\} \\ \Theta &= \max\{\Theta_+, \Theta_-, \Theta_0\} \end{aligned}$$

Ist $\Theta = \pm\infty$, so ist das LP unzulässig.
Sonst setze $n_e = n_{e_+} \iff \Theta = \Theta_+$ und $n_e = n_{e_-} \iff \Theta = \Theta_-$

4.

Falls $n_e \neq j_q$ setze

$$\Delta f = A_{.B}^{-1} a_{n_e}$$

5. UPDATE

$$\begin{aligned} h &= h + \Theta \Delta h \\ g &= g + \Theta \Delta g \end{aligned}$$

Falls $n_e \neq j_q$ setze

$$\begin{aligned} \Phi &= \tfrac{f_q - \rho}{\Delta f_q} \\ f &= f - \Phi \Delta f + (R_{n_e} + \Phi - \rho)\, \vec{e_q} \\ B &= B \setminus \{j_q\} \cup \{n_e\} \\ N &= N \setminus \{n_e\} \cup \{j_q\} \end{aligned}$$

(wobei die Einordnung der Indizes gemäß Definition erfolgt)

Aktualisiere die Vektoren l, u, R.

6. Gehe zu Schritt **1.**

Wieder soll die partielle Korrektheit von Algorithmus 1.3 gezeigt werden. Dies geschieht in zwei Schritten: ein Lemma, das sicherstellt, daß die Updates korrekt ausgeführt werden, und ein Satz, der die Terminierungs-Schlußfolgerungen untersucht.

Lemma 1.48 *Ist $S = (B, N)$ eine dual zulässige Spaltenbasis, so gilt nach dem Update*

$$\begin{aligned} f' &= -A_{.B'}^{-1}\, AR' \\ h' &= A_{.B'}^{-T}\, c_{B'} \\ g' &= A^T h' \\ l' &\leq g' \leq u' \end{aligned}$$

Beweis. Ist $n_e \in B$ so gilt wegen $\Delta g_B = \vec{e_q}$ zwangsläufig $j_q = n_e$. Die Basis, R und f ändern sich nicht, die erste Gleichung ist daher weiterhin korrekt. Für $n_e \notin B$ gilt mit $R' = R - R_{n_e} \vec{e_{n_e}} + \rho \vec{e_{j_q}}$, $\rho = f_q - \Phi \Delta f_q$ und $A_{.B'} = A_{.B} + (a_{n_e} - a_{j_q}) \vec{e_q}^T$

$$\begin{aligned} A_{.B'} f' &= \left[A_{.B} + (a_{n_e} - a_{j_q}) \vec{e_q}^T\right] (f - \Phi \Delta f + (R_{n_e} + \Phi - \rho)\, \vec{e_q}) \\ &= A_{.B} f - \Phi A_{.B} \Delta f + (R_{n_e} + \Phi - \rho) a_{j_q} \\ &\quad + f_q (a_{n_e} - a_{j_q}) - \Phi \Delta f_q (a_{n_e} - a_{j_q}) + (R_{n_e} + \Phi - \rho)(a_{n_e} - a_{j_q}) \\ &= -AR - \Phi a_{n_e} + (R_{n_e} + \Phi - \rho) a_{j_q} \\ &\quad + f_q (a_{n_e} - a_{j_q}) - (f_q - \rho)(a_{n_e} - a_{j_q}) + (R_{n_e} + \Phi - \rho)(a_{n_e} - a_{j_q}) \\ &= -AR + (-\Phi + f_q + \rho - f_q + R_{n_e} + \Phi - \rho)\, a_{n_e} \\ &\quad + (R_{n_e} + \Phi - \rho + f_q - \rho - f_q - R_{n_e} - \Phi + \rho)\, a_{j_q} \\ &= -AR + R_{n_e} a_{n_e} - \rho a_{j_q} \\ &= -AR' \end{aligned}$$

Da $\Theta \neq \pm\infty$, muß l_{n_e} bzw. u_{n_e} einen endlichen Wert annehmen, nach Definition kann dies nur c_{n_e} sein. Daher gilt also $\Theta = \frac{c_{n_e} - g_{n_e}}{\Delta g_{n_e}}$ und die Updates für g und h entsprechen denen aus Algorithmus 1.1, für den die Beziehungen schon in Lemma 1.25 gezeigt wurden. Bleibt also noch die Erhaltung der dualen Zulässigkeit.

Für Basisvariablen gilt $g'_{B'} = (A^T A^{-T}_{\cdot B'} c_{B'})_{B'} = c_{B'}$. Für Nichtbasisvariablen ist Θ als Minimum (bzw. Maximum negativer Werte) gerade so gewählt, daß g'_{n_e} den Wert c_{n_e} annimmt und keine andere pricing-Variable ihre Schranken verletzt. Die Schranken für g_{j_q} werden in Schritt 2 neu gesetzt, so daß auch diese Variable zulässig bleibt. ■

Satz 1.49 *Terminiert Algorithmus 1.3 in Schritt 1, so ist die optimale Lösung von (1.9) gefunden. Terminiert er in Schritt 3, so ist (1.9) unzulässig.*

Beweis.

- Der Algorithmus terminiert in Schritt 1
  Nach (1.48) und wegen $L \leq x \leq U$ ist die Spaltenbasis B primal und dual zulässig.
- Der Algorithmus terminiert in Schritt 3
  Die Lösung bleibt für eine beliebig große (bzw. kleine) duale Schrittweite Θ zulässig, da keine der Variablen ihre Schranken verletzt. Desweiteren ist die Steigung der dualen Zielfunktion negativ, wie in Lemma 3.6 gezeigt wird. Der duale Zielfunktionswert kann beliebig klein werden. Wegen der schwachen Dualität kann keine primal zulässige Lösung existieren.

■

1.7 Die Zeilendarstellung der Basis

In Definition 1.16 wurde eine Spaltenbasis definiert um eine Ecke des Polyeders festzulegen, der die zulässige Menge des linearen Programms bildet. Anschaulich ist dies aber eigentlich nicht, auch wenn diese Vorgehensweise in der Standardliteratur üblich ist. Betrachtet man ein Polyeder wie in Abbildung 1.5, so würde man eine Ecke eher durch die Hyperebenen definieren, die sich in diesem Punkt schneiden, als durch auf null gesetzte Variablen. Dies ist der Ansatz der Zeilenbasis.

Wir betrachten in diesem Abschnitt das lineare Programm mit Nebenbedingungsmatrix $D \in \mathbb{R}^{m \times n}$, $m \geq n$

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Dx & \geq \quad d \end{array} \tag{1.11}$$

Definition 1.50 *Ein geordnetes Paar $Z = (B, N)$ von Mengen $B, N \subseteq \{1, \ldots, m\}$ heißt Zeilenbasis eines gegebenen LPs , wenn folgendes gilt:*

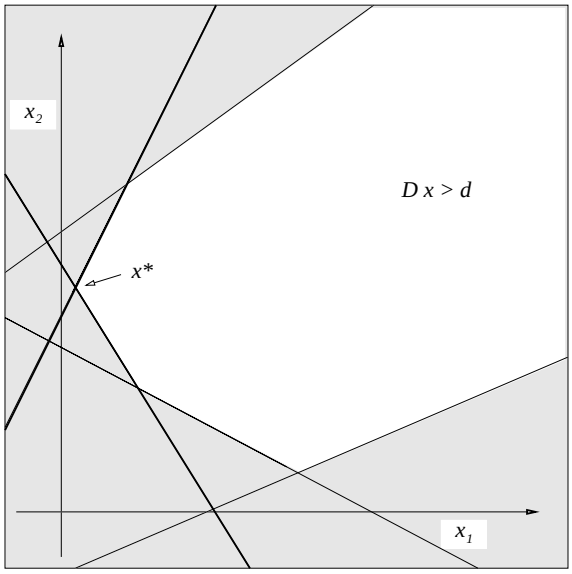


Abbildung 1.5: Ecke eines Polytops

1. $B \cup N = \{1, \ldots, m\}$
2. $B \cap N = \emptyset$
3. $|B| = n$
4. $D_{B.}$ *ist regulär*

Definition 1.51 *Eine Ungleichung $D_{i.}x \geq d_i$ heißt Basisungleichung einer Zeilenbasis Z, wenn $i \in B$. Sie heißt Nichtbasisungleichung, wenn $i \in N$. Als Basis bezeichnet man je nach Zusammenhang die Indexmenge B oder die Menge der Basisungleichungen $\{D_{i.}x \geq d_i : i \in B\}$.*
Die Matrix $D_{B.}$ wird Basismatrix genannt, den Vektor $x = D_{B.}^{-1}d$ nennt man Basislösungsvektor. Gilt $Dx \geq d$, so heißt x zulässiger Basislösungsvektor und Z (primal) zulässige Zeilenbasis. Gilt $c^T x \geq c^T y \ \forall \ y \in P$, so heißt x optimaler Basislösungsvektor und Z (primal) optimale Zeilenbasis.
$y = D_{B.}^{-T}c$ nennt man den dualen Vektor, $s = Dx$ den Vektor der Schlupfvariablen.

Analog zum Vorgehen in den ersten Abschnitten dieses Kapitels kann man die Optimalitätsbedingung für eine Basislösung als Folgerung aus dem Darstellungssatz nachweisen: $y \geq 0$. Zu einer Zeilenbasis läßt sich ein Algorithmus aufstellen, der denen für die Spaltenbasis sehr ähnlich ist. Um dies noch deutlicher zu machen, initialisiert man folgendermaßen:

0. INIT

$$\begin{array}{lllll} A & = & D^T & & \\ f & = & y & = & A_{.B}^{-1}\, c \\ h & = & x & = & A_{.B}^{-T}\, d_B \\ g & = & s & = & A^T h \end{array}$$

Mit diesen Bezeichnungen läßt sich das Programm (1.11) mit Algorithmus 1.2 lösen (bis auf einen kleinen Unterschied bei der Bestimmung von Θ).

Auch für die Zeilenbasis läßt sich ein dualer Algorithmus finden, der auf einer optimalen Basis — also $y \geq 0$ — arbeitet. Mit der obigen Initialisierung verwendet man dazu Algorithmus 1.1.
Der Grund für diese mathematische Äquivalenz liegt natürlich in der Dualität begründet, (1.11) ist das duale Programm zu (1.2). Eine Zeilenbasis ist für einige Programme günstiger, siehe hierzu den nächsten Abschnitt.

Um einen Algorithmus für Bereichsungleichungen formulieren zu können, formt man das LP (1.9) in ein Zeilen-LP um:

$$\begin{array}{lccc} \max & c^T x & & \\ \text{s.t.} & l & \leq A^T x & \leq u \end{array} \tag{1.12}$$

Hierbei wurden $l = \begin{pmatrix} l' \\ l'' \end{pmatrix}, u = \begin{pmatrix} u' \\ u'' \end{pmatrix}$ und $A^T = \begin{pmatrix} I \\ A' \end{pmatrix}$ gesetzt. Eine Zeilenbasis wird analog zu 1.43 definiert.

Definition 1.52 *Ein geordnetes Paar $Z = (B, N)$ von Mengen $B = B_f \uplus B_x \uplus B_l \uplus B_u$, $N = N_f \uplus N_x \uplus N_l \uplus N_u \uplus N_n$ heißt Zeilenbasis von LP 1.12, wenn folgendes gilt:*

1. $B \cup N = \{1, \ldots, n+m\}$
2. $B_f, B_l, B_u, B_x, N_f, N_l, N_u, N_n$ *und* N_x *sind paarweise disjunkt*
3. $|B| = n$
4. $A^T_{B\cdot}$ *ist regulär*
5. $$\begin{array}{lcccl} i \in B_l \cup N_u & \Rightarrow & -\infty < l_i & \neq & u_i = \infty \\ i \in B_u \cup N_l & \Rightarrow & -\infty = l_i & \neq & u_i < \infty \\ i \in N_b & \Rightarrow & -\infty < l_i & \neq & u_i < \infty \\ i \in B_f \cup N_n & \Rightarrow & -\infty = l_i & \textit{und} & u_i = \infty \\ i \in B_x \cup N_f & \Rightarrow & -\infty < l_i & = & u_i < \infty \end{array}$$

Eine Bereichsungleichung $l_i \leq A^T_{i\cdot} x \leq u_i$ gibt zwei parallele Seitenflächen des Polyeders an. Von ihnen kann aber nur eine zur Bestimmung der aktuellen Ecke herangezogen werden und auch nur eine verletzt sein. Daher führt man wieder eine Aufteilung der Basisungleichungen in verschiedene Mengen ein. Hierbei bedeutet nun

- $i \in B_l$: Es gilt $A^T_{i\cdot} x = l_i$.
- $i \in B_u$: Es gilt $A^T_{i\cdot} x = u_i$.
- $i \in B_x$: Es gilt $A^T_{i\cdot} x = l_i = u_i$.
- $i \in B_f$: Es gilt $A^T_{i\cdot} x = 0$ (künstlich gesetzt, Ungleichungen immer erfüllt).

Die Unterteilung der Nichtbasismenge erfolgt nur aus algorithmischen Gründen, eigentlich würde eine Menge N ausreichen. Durch sie kann aber besser bestimmt werden, in welche Basisungleichungenmenge eine Variable bei Eintritt in die Basis gehört. Vergleiche hierzu den Abschnitt 2.3.3 über die in SoPlex implementierte Basis.

Um Zulässigkeit und Optimalität überprüfen zu können, sind zwei weitere Definitionen notwendig.
Wie in den Abschnitten über Dualität und nichtlineare Optimierung gezeigt wurde, bedingen "$\leq$-Ungleichungen" negative und "$\geq$-Ungleichungen" positive duale Variablen, die hier dem Zulässigkeitsvektor entsprechen. Bei Gleichungen spielt das Vorzeichen keine Rolle. Bei einer echten Bereichsungleichung ist immer eine der beiden dualen Variablen null, daher wird nur ein Wert verwaltet und aus dem Vorzeichen auf den Bezug geschlossen.

Definition 1.53 *Definiere die Schranken des Zulässigkeitsvektors wie folgt,* $i \in \{1, \ldots, n+m\}$*:*

$$L_i := \begin{cases} 0 & \text{für } i \in B_f \cup N_n \\ -\infty & \text{für } i \in B_l \cup N_u \\ 0 & \text{für } i \in B_u \cup N_l \\ -\infty & \text{für } i \in B_x \cup N_f \cup N_b \end{cases} \qquad U_i := \begin{cases} 0 & \text{für } i \in B_f \cup N_n \\ 0 & \text{für } i \in B_l \cup N_u \\ \infty & \text{für } i \in B_u \cup N_l \\ \infty & \text{für } i \in B_x \cup N_f \cup N_b \end{cases}$$

Um den Basislösungsvektor bestimmen zu können, muss die Inverse der Basismatrix mit der aktuellen rechten Seite multipliziert werden. Diese wird mit dem Vektor r bezeichnet.

Definition 1.54 *Definiere den Vektor* $r \in \mathbb{R}^n$ *durch*

$$r_i := \begin{cases} 0 & \text{für } j_i \in B_f \\ l_{j_i} & \text{für } j_i \in B_l \cup B_x \\ u_{j_i} & \text{für } j_i \in B_u \end{cases}$$

1.8 Der Algorithmus mit allgemeiner Basis

Im Hinblick auf die Effizienz von Implementierungen des Simplex-Algorithmus sind zwei wichtige Unterscheidungen zu treffen. Die erste ist die zwischen einfügendem und entfernendem Algorithmus, womit gemeint ist, ob im pricing eine Variable bestimmt wird, die in die Basis wechselt oder aber eine, die von B nach N geht. Die zweite Unterscheidung betrifft die Dimension der Basis, da diese entscheidend ist für die Laufzeit des Algorithmus.

	Spaltenbasis	Zeilenbasis
primal	einfügend	entfernend
dual	entfernend	einfügend

Tabelle 1.2: Zusammenhang zwischen Basisdarstellung und Algorithmustyp

Was den ersten Punkt betrifft, besteht offensichtlich eine Äquivalenz zwischen dem primalen Algorithmus mit Spaltenbasis und dem dualen bei Benutzung einer Zeilenbasis, siehe hierzu Tabelle 1.2.

Der einfügende Algorithmus arbeitet auf einer primal zulässigen, der entfernende dagegen auf einer primal optimalen Basislösung. Die jeweiligen Vor- und Nachteile liegen bei der Generierung neuer Startbasen für modifizierte LPs und wurden schon in Abschnitt 1.3 angesprochen.
Was die Dimension betrifft, tritt der Vorteil einer Zeilenbasis bei den sogenannten Bereichsungleichungen zutage. Das duale Programm des LPs (1.9) hat folgende Gestalt:

$$\begin{array}{lrcl} \min & {u'}^T w_1 - {l'}^T v_1 + {u''}^T w_2 - {l''}^T v_2 & & \\ \text{s.t.} & w_1 - v_1 + {A'}^T w_2 - {A'}^T v_2 & = & c \\ & v_1, v_2, w_1, w_2 & \geq & 0 \end{array}$$

Die Dimension der Nebenbedingungsmatrix hat sich verdoppelt. In der Zeilenbasisdarstellung hat die Basismatrix dagegen nur die Dimension n.

Es bleibt also festzuhalten, daß es vier verschiedene Arten von Simplex-Algorithmen gibt, von denen sich je zwei nach Tabelle 1.2 entsprechen, die aber durchaus eine andere Dimension der Basismatrix verursachen können.
Um beispielsweise pricing oder ratio test Klassen für zwei Algorithmen gleichzeitig nutzen zu können, kann man eine allgemeine Basis definieren, so daß ein Algorithmus nach entsprechender Initialisierung der Basis unabhängig davon ist, ob eine Zeilen- oder Spaltenbasis zugrunde liegt. Dies hat den großen Vorteil, daß man bei der Formulierung neuer Klassen basisunabhängig schreiben kann und der Anwender trotzdem den Vorteil der Dimensionswahl behält.

Für eine formale Definition und die zugehörigen Beweise sei auf [Wun96] verwiesen. Für das Verständnis dieser Arbeit reicht die Feststellung aus, daß f den Zulässigkeitsvektor und g, h den pricing- bzw. copricing-Vektor bezeichnen und Matrix, Zielfunktionsvektor und rechte Seite entsprechend initialisiert sein müssen. Sei dazu eine Zeilen- oder Spaltenbasis von (1.9) gegeben. Die Bezeichnungen werden dann nach Tabelle 1.3 definiert.
Die allgemeine Basis wird mit einer Zeilen- oder Spaltenbasis identifiziert. Dies ist wohldefiniert, da die Basismatrix einer Spaltenbasis S mit der Basis B genau dann regulär ist, wenn die Basismatrix einer Zeilenbasis mit Basis $J \setminus B$ in Zeilendarstellung regulär ist, wie man durch Permutation der Basismatrizen und Anwendung des Determinantenkriteriums zeigt.

In dieser Arbeit wird die allgemeine Basis nur bedingt genutzt, da die Modifikationen am ratio test eine bestimmte Struktur voraussetzen und nur für jeweils eine der Basisdarstellungen gültig sind, wie im Kapitel drei erläutert wird. Daher ist die Einführung der allgemeinen Basis nicht unbedingt sinnvoll für die vorgenommenen Änderungen. Sehr wohl wird allerdings die Schreibweise beibehalten, da das Programm SoPlex die auftauchenden Variablen auf diese Art repräsentiert und die Gemeinsamkeiten der Algorithmen sehr deutlich hervortreten. Hier soll nur einer der beiden Algorithmen für eine allgemeine Basis angegeben werden um das Konzept deutlicher zu machen.

Algorithmus 1.4 (Entfernender Algorithmus)
Sei B eine Basis von (1.9) und gelte $l_N \leq g_N \leq u_N$

	Spaltenbasis	Dimension	**Zeilenbasis**	Dimension
A	$\begin{pmatrix} A' & -I \end{pmatrix}$	$m \times (n+m)$	$\begin{pmatrix} I & A'^T \end{pmatrix}$	$n \times (n+m)$
b	0	m	c'	n
c	$\begin{pmatrix} c' \\ 0 \end{pmatrix}$	$n+m$	c'	n
L, U	$\begin{pmatrix} l' \\ l'' \end{pmatrix}, \begin{pmatrix} u' \\ u'' \end{pmatrix}$	$n+m$	Nach Definition 1.53	$n+m$
l, u	Nach Definition 1.46	$n+m$	$\begin{pmatrix} l' \\ l'' \end{pmatrix}, \begin{pmatrix} u' \\ u'' \end{pmatrix}$	$n+m$
R	Nach Definition 1.44	$n+m$	0	$n+m$
r	c_B	m	Nach Definition 1.54	n
f	$x_B = A_{.B}^{-1}(b - AR)$	m	$y_B = A_{.B}^{-1}(b - AR)$	n
h	$y = A_{.B}^{-T} r$	m	$x = A_{.B}^{-T} r$	n
g	$A^T h$	$n+m$	$A^T h$	$n+m$

Tabelle 1.3: Initialisierung der allgemeinen Basis

0. INIT

Initialisiere alle Größen gemäß Tabelle 1.3

1. PRICING

Ist $L_B \leq f \leq U_B$, so ist x optimal.

Sonst wähle $j_q \in B$ mit
- Fall a) $f_q > U_{j_q}$
- Fall b) $f_q < L_{j_q}$

2.

Setze

$$\begin{aligned} \Delta h &= A_{.B}^{-T} \vec{e_q} \\ \Delta g &= A^T \Delta h \end{aligned}$$

Falls Spaltenbasis setze

Fall a) $\Theta_0 = -\infty$, $\rho = U_{j_q}$, $l_{j_q} = -\infty$

Fall b) $\Theta_0 = +\infty$, $\rho = L_{j_q}$, $u_{j_q} = \infty$

Falls Zeilenbasis setze

Fall a) $\Theta_0 = +\infty$, $\rho = U_{j_q}$

Fall b) $\Theta_0 = -\infty$, $\rho = L_{j_q}$

3. RATIO TEST

Falls $\Theta_0 > 0$ setze

$$\begin{aligned}\Theta_+ &= \frac{u_{n_{e_+}} - g_{n_{e_+}}}{\Delta g_{n_{e_+}}} \text{ mit } n_{e_+} \in \arg\min\{\tfrac{u_i - g_i}{\Delta g_i} : \Delta g_i > 0\}\\ \Theta_- &= \frac{l_{n_{e_-}} - g_{n_{e_-}}}{\Delta g_{n_{e_-}}} \text{ mit } n_{e_-} \in \arg\min\{\tfrac{l_i - g_i}{\Delta g_i} : \Delta g_i < 0\}\\ \Theta &= \min\{\Theta_+, \Theta_-, \Theta_0\}\end{aligned}$$

Falls $\Theta_0 < 0$ setze

$$\begin{aligned}\Theta_+ &= \frac{l_{n_{e_+}} - g_{n_{e_+}}}{\Delta g_{n_{e_+}}} \text{ mit } n_{e_+} \in \arg\max\{\tfrac{l_i - g_i}{\Delta g_i} : \Delta g_i > 0\}\\ \Theta_- &= \frac{u_{n_{e_-}} - g_{n_{e_-}}}{\Delta g_{n_{e_-}}} \text{ mit } n_{e_-} \in \arg\max\{\tfrac{u_i - g_i}{\Delta g_i} : \Delta g_i < 0\}\\ \Theta &= \max\{\Theta_+, \Theta_-, \Theta_0\}\end{aligned}$$

Ist $\Theta = \pm\infty$, so ist das LP unzulässig oder unbeschränkt.
Sonst setze $n_e = n_{e_+} \Longleftrightarrow \Theta = \Theta_+$ und $n_e = n_{e_-} \Longleftrightarrow \Theta = \Theta_-$

4.

Falls $n_e \neq j_q$ setze

$$\Delta f = A_{\cdot B}^{-1} a_{n_e}$$

5. UPDATE

$$\begin{aligned}h &= h + \Theta\Delta h\\ g &= g + \Theta\Delta g\end{aligned}$$

Falls $n_e = j_q$ setze

$$\begin{aligned}B_l &= B_l \setminus \{j_q\} \text{ und } B_u = B_u \cup \{j_q\} \quad \text{falls } j_q \in B_l\\ B_l &= B_l \cup \{j_q\} \text{ und } B_u = B_u \setminus \{j_q\} \quad \text{falls } j_q \in B_u\end{aligned}$$

Falls $n_e \in N$ setze

$$\begin{aligned}\Phi &= \frac{f_q - \rho}{\Delta f_q}\\ f &= f - \Phi\Delta f + (R_{n_e} + \Phi - \rho)\,\vec{e_q}\\ B &= B \setminus \{j_q\} \cup \{n_e\}\\ N &= N \setminus \{n_e\} \cup \{j_q\}\end{aligned}$$

(wobei die Einordnung der Indizes gemäß Definition erfolgt)
Aktualisiere die Vektoren L, U, l, u, R, r.

6. Gehe zu Schritt **1.**

Kapitel 2

Das Softwarepaket SoPlex

Der praktische Teil dieser Arbeit basiert auf dem Programmpaket SoPlex, erstellt von Roland Wunderling im Rahmen seiner Promotion am Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB). SoPlex steht für **S***equential* **o***bject-oriented sim***Plex**. Es wurde in C++ geschrieben und folgt den Ansätzen der objektorientierten Programmierung (OOP). Es ist unter den Lizenzbedingungen des ZIB unter der im Literaturverzeichnis angegebenen URL als source code erhältlich [Sop96].

Der objektorientierte Ansatz hat den großen Vorteil, wie im ersten Abschnitt über die Grundzüge der OOP ausgeführt wird, daß man sehr einfach über klar definierte Schnittstellen eigene Implementierungen bestimmter Teilbereiche in das gesamte Programmpaket einbauen und den Rest des state-of-the-art Lösers ohne weiteres übernehmen kann.
Ferner bekommt man hervorragende Vergleichsmöglichkeiten mit anderen Klassen, da die Bedingungen identisch sind. Schreibt man dagegen einen eigenen solver um bestimmte Modifizierungen zu testen, so wird der generelle Unterschied zwischen den verglichenen solvern den Blick auf mögliche Verbesserungen erschweren.

Nach der allgemeinen Darstellung der Eigenschaften und Vorteile der OOP wird die Klassenstruktur von SoPlex als Beispiel einer objektorientierten Implementierung eines Lösers für lineare Programme dargestellt.

Die für diese Arbeit besonders interessanten Klassen werden dann im letzten Abschnitt genauer beleuchtet.

2.1 Objektorientierte Programmierung und C++

Die Geschichte der Programmierung beginnt nach vorherrschender Meinung der Experten mit ersten theoretischen Ideen von Babbage (1792-1871) und der ersten niedergeschriebenen Programmiersprache der Welt von Konrad Zuse (1942-1945/46), welche er den Plankalkül [Zus45] nannte. Seit der Entwicklung des ersten Computers wurden geschätzte eintausend Programmiersprachen entwickelt und es existieren viele Untersuchungen über mögliche Unterscheidungen und Typisierungen, siehe beispielsweise [Zus99]. An dieser Stelle soll nur ein kurzer Überblick gegeben werden um das objektorientierte Programmierparadigma abgrenzen zu können.

Die ersten Programmiersprachen waren sogenannte *strukturierte* oder *imperative Programmiersprachen*. Ein Programm besteht aus einer Reihe von elementaren Operationen auf elementaren Daten, die durch bestimmte Konstruktionen wie Schleifen, Verzweigungen oder Sprungbefehle einen Programmablauf ergeben, der sehr gut durch die sogenannten *Flussdiagramme* dargestellt werden kann.
Diese Programmiertechnik erwies sich in der Praxis bei steigenden Ansprüchen an die Leistungsfähigkeit und Qualität der Software als zunehmend ungeeignet für größere Projekte. Um dies besser fassen zu können, müssen wir Qualitätsfaktoren betrachten. Die wichtigsten externen Qualitätsfaktoren sind nach [Mey88]:

- *Korrektheit* ist die Fähigkeit von Softwareprodukten ihre Aufgaben genauso zu erfüllen, wie es Anforderungen und Spezifikation vorgeben.
- *Robustheit* ist die Fähigkeit von Softwaresystemen auch unter untypischen Bedingungen zu funktionieren.
- *Erweiterbarkeit* ist die Leichtigkeit mit der Softwareprodukte an Änderungen der Spezifikation angepaßt werden können.
- *Wiederverwendbarkeit* ist die Fähigkeit von Softwareprodukten als Teil oder als Ganzes für eine neue Anwendung wiederverwendet zu werden.
- *Kompatibilität* ist die Leichtigkeit mit der Softwareprodukte mit anderen kombiniert werden können.

Ein Ansatz um diesen Ansprüchen gerecht zu werden, ist die *Modularisierung*. Eine präzise Definition des Wortes Modul existiert in diesem Zusammenhang nicht, sie hängt zu sehr von den Eigenheiten der jeweiligen Sprache ab. Oft wird unter Modularisierung jedoch die Zerlegung von großen Programmen in kleinere Einheiten verstanden, die Module genannt werden. Seit ca. 1970 wird dieses Konzept bei Programmiersprachen berücksichtigt.
Die Vorteile der Modularisierung liegen auf der Hand. Kleinere Module sind leichter auf Fehler zu prüfen, durch die klare Vorgabe der Schnittstellen auch in einem anderen Zusammenhang einsetzbar und besser zu erweitern als ein einziges großes Programm.
Sowohl die objektorientierte als auch die funktionale (prozedurale) Programmierung entsprechen diesem Konzept.
Der Unterschied zwischen diesen beiden Programmiertechniken ist ein prinzipieller in der Herangehensweise an die Aufgabe. Ganz allgemein kann man bei der Betrachtung eines Computerprogramms zwischen Daten und Algorithmen unterscheiden. Die Daten umfassen jede Art von Variablen, die im Laufe des Programms auftauchen. Die Algorithmen dagegen operieren genau auf diesen Daten und verändern bzw. nutzen sie in einer nicht näher bestimmten Art und Weise.
Die *prozedurale Programmierung* (C, FORTRAN, Pascal,...) stellt die Prozeduren in den Vordergrund und betrachtet die Daten als in- bzw. output dieser Prozeduren. Der Ablauf des Programmes wird durch gezielten Aufruf von Funktionen geregelt.
Die *objektorientierte Programmierung* (OOP) betrachtet primär Daten und sieht Algorithmen, die diese Daten nutzen, als zugehörige Methoden an. Daten und Methoden werden hier zu Objekten zusammengefaßt, wobei diese Methoden sich untereinander aufrufen und dadurch den Programmablauf bestimmen.

Eine Definition des objektorientierten Entwurfs ist nach [Mey88]

Objektorientierter Entwurf ist die Konstruktion von Softwaresystemen als strukturierte Sammlung von Implementierungen abstrakter Datentypen.

Datenstrukturen sind auch in Sprachen wie PL/1 oder sogar Visual Basic (TYPE) schon bekannt. Mit diesen Datenstrukturen konnte beispielsweise die Programmierung eines stacks mit den Operationen push und pull realisiert werden. Hierbei handelt es sich aber um eine spezielle Implementierung eines stacks, nicht eine abstrakte Beschreibung von Datenstrukturen und deren Operationen wie bei den abstrakten Datentypen. Die Theorie der abstrakten Datentypen beschreibt eine Klasse von Datenstrukturen nicht durch die Implementierung, sondern durch eine externe Sicht.

Die OOP ist eine noch relativ junge Programmiertechnik, auch wenn es inzwischen eine Vielzahl objektorientierter Programmiersprachen gibt wie beispielsweise Smalltalk, CLOS, Cecil, Objective Caml, Haskell, Obliq und natürlich die beiden am häufigsten benutzten, Java und C++. Während das Konzept schon ein wenig älter ist und 1973 mit Smalltalk von Goldberg und anderen die erste rein objektorientierte Programmiersprache entstand, existiert die Programmiersprache C++ seit ungefähr 1983, als sie bei AT&T unter Leitung von Bjarne Stroustrup entwickelt wurde. Seitdem gab es immer wieder neue releases, seit 1989 existiert ein ANSI-Kommitee zur Standardisierung der Sprache C++.

Der Vorteil des objektorientierten Ansatzes liegt in einer besseren Strukturierung des Programmes. Er entspricht eher dem top-bottom-Prinzip als der prozedurale, da als allererstes die grobe Aufteilung in die verschiedenen Klassen zu erfolgen hat und erst auf unterer Ebene die jeweilige Implementierung der Methoden vorgenommen wird. Die OOP erlaubt es dem Programmierer im Vergleich zur prozeduralen Programmierung daher auch größere Projekte noch übersichtlich zu halten und unter den Gesichtspunkten Korrektheit, Robustheit, Erweiterbarkeit, Wiederverwendbarkeit und Kompabilität zu optimieren.

Um diesen Zielen näher zu kommen, gibt es einige fundamentale Mechanismen, die den gängigen objektorientierten Sprachen gemein sind. Zu ihnen zählen die Zusammenfassung in Objekte, Kapselung, Vererbung und Polymorphismen.

- **Objekte und Klassen**

  Ein Objekt ist eine gekapselte Datenstruktur, die sowohl Daten als auch Methoden enthält, die auf diesen Daten operieren. Eine Klasse ist der Datentyp eines Objektes, also die Beschreibung von Typ und Zusammenhang der enthaltenen Daten und Methoden.
  Ein Beispiel hierfür ist eine Klasse Vektor, von der man mehrere Objekte besitzt, die die Werte und Updaterichtungen der Vektoren f, g und h sowie Methoden um diese Werte zu modifizieren, enthalten. Ein anderes Beispiel wäre eine Klasse Basis, die die aktuelle Basis des linearen Programmes wiedergibt und von der man nur ein einziges Objekt benötigt und vernünftigerweise anlegt.

- **Kapselung**

  Eine wichtige Neuerung gegenüber den auch in prozeduralen Sprachen üblichen Datenstrukturen sind hierbei die Zugriffsrechte. Die Daten des Objektes werden nur von

den eigenen Methoden verändert und nicht mehr von beliebigen, schlecht wartbaren Funktionen an anderer Stelle im Programm. Methoden kommunizieren untereinander über so genannte Botschaften und sorgen so für klare und saubere Schnittstellen.

Die einzelnen Klassen sind nicht nur robuster, sondern, wenn man die Schnittstellen berücksichtigt, auch leicht austauschbar und durch andere Implementierungen zu ersetzen.

- **Vererbung**

  Klassen können aus bereits bestehenden Klassen abgeleitet werden. Das bedeutet, sie übernehmen als Grundgerüst die komplette Datenstruktur und fügen zu dieser weitere Struktur hinzu um eine zusätzliche Spezialisierung vorzunehmen. Da auch bereits abgeleitete Klassen als neue Basisklasse dienen können, sind ganze Vererbungsketten möglich. In C++ sind auch mehrere Basisklassen erlaubt, man spricht hier von *multipler Vererbung*.
  Die Vererbung hat im Wesentlichen drei Vorteile. Zum einen können erprobte und bewährte Strukturen übernommen werden, ohne neu implementiert werden zu müssen oder die Basisklasse zu verändern. Zum zweiten sind viele Strukturen in der realen Welt ebenfalls hierarchisch organisiert und können so sehr gut abgebildet werden. Zum dritten können abgeleitete Klassen Basisklassen ersetzen, so kann man Polymorphismus nutzen.

  Bleibt man beim obigen Beispiel einer Vektor-Klasse, so könnte man sich eine Einheitsvektor-Klasse als Spezialisierung vorstellen, die als zusätzliche Methoden schnellere Implementierungen bezüglich Initialisierung und Berechnung bietet.

- **Polymorphismen**

  Als Polymorphismus wird die Eigenschaft der oben angesprochenen Botschaften bezeichnet, in verschiedenen Zusammenhängen anders zu wirken. Dies hat konkret mehrere Bedeutungen.

  Besteht die Möglichkeit, beliebige Objekte als Parameter zu übergeben, so spricht man von *dynamic typing* im Gegensatz zum *static typing*, bei dem der Typ eines übergebenen Datentyps schon zur Übersetzzeit festgelegt sein muß. Static typing ist der Standard bei den gängigen imperativen Programmiersprachen wie C, FORTRAN oder Pascal.

  Methoden können den gleichen Namen haben und es wird erst anhand der übergebenen Parameter zur Laufzeit entschieden, welche Methode ausgeführt wird. Dies ist in compilierten Programmiersprachen nicht üblich, hier legt der Compiler im allgemeinen schon fest, mit welcher Speicheradresse ein Funktionsaufruf verknüpft ist. Diese Art der Zuordnung von Methoden zu Aufrufen erst zur Laufzeit wird als *late binding* bezeichnet.

  Ein weiterer Polymorphismus ist die Möglichkeit, Operatoren zu überladen um das Programm lesbarer zu machen. Auch Funktionsnamen können natürlich überladen

werden.

In C++ gibt es dynamic typing und late binding nur in Spezialfällen:

- Im Zusammenhang einer Vererbungshierarchie. Eine abgeleitete Klasse kann als Basisklasse übergeben werden.
- Erzwungenes late binding durch den Zusatz *virtual* im Falle gleicher Methodennamen abgeleiteter Klassen.
- Parametrisierte Klassen, also templates.

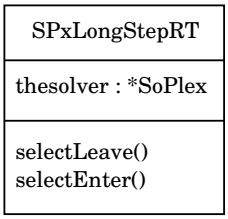
Von daher ist C++ im strengen Sinne keine reine objektorientierte Sprache. Die bereitgestellten Approximationen genügen aber in den meisten Zusammenhängen den Ansprüchen an Objektorientierung. Der Vorteil liegt in einer schnelleren Abarbeitung, ein Methodenaufruf ist nach [Wun96] vom Aufwand her dem Aufruf einer C-Funktion vergleichbar. C++ eignet sich daher für die effiziente Implementierung mathematischer Algorithmen.

2.2 Unified Modeling Language

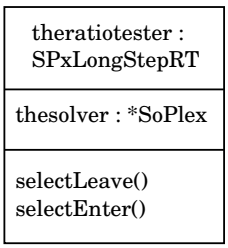
Die Unified Modeling Language (UML) ist eine graphische Sprache zur Visualisierung, Spezifikation, Konstruktion und Dokumentation von Softwaresystemen und nimmt seit einigen Jahren einen enormen Stellenwert im Softwareengineering ein. In dieser Arbeit sollen die wichtigsten Konstrukte genutzt werden um die Struktur von SoPlex visualisieren zu können. Hier eine kurze Nennung der dafür notwendigen Konventionen, für eine ausführliche Darstellung siehe beispielsweise [BRJ99].

- **Darstellung von Klassen und Objekten**

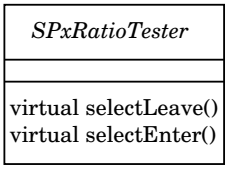
  Klassen werden als Rechtecke mit getrennten Bereichen für Klassenname, Attribute und Methoden dargestellt, wobei die letzten beiden Bereiche je nach Zusammenhang auch weggelassen werden können.



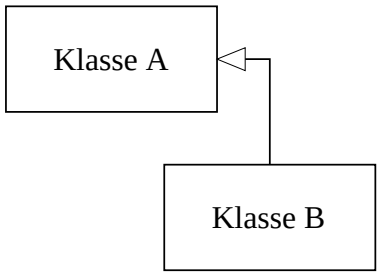
  Objekte, also instanziierte Klassen, sehen ähnlich aus, enthalten aber einen Doppelpunkt und den Typ nachgestellt.



Abstrakte Klassen sind Klassen, die nicht direkt instanziiert werden können, sondern noch einmal abgeleitet werden müssen. In C++ sind das alle Klassen, die mindestens eine virtuelle Funktion beinhalten. Solche Klassen werden durch einen kursiv geschriebenen Klassennamen gekennzeichnet.

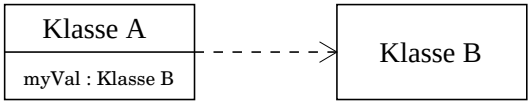


- **Generalisierung**



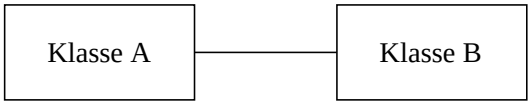
Der Vorgang *Klasse B wird von Klasse A abgeleitet, erbt also dessen Attribute und Operationen* wird durch einen Pfeil von der abgeleiteten Klasse zur Basisklasse dargestellt.

- **Abhängigkeit**



Eine Klasse ist abhängig von einer anderen, wenn diese von ihr benutzt wird. Dies wird durch eine gestrichelte Linie mit Pfeilspitze von der abhängigen zur unabhängigen Klasse visualisiert.

- **Assoziation**



Zwei Klassen werden assoziiert genannt, wenn sie Methoden der anderen aufrufen können. Die Darstellung erfolgt über eine einfache Verbindungslinie.

2.3 Beschreibung von SoPlex

In diesem Abschnitt soll der Aufbau des Softwarepaketes SoPlex beschrieben werden. SoPlex besteht aus mehr als 50000 Zeilen Quellcode und ist, wie oben erwähnt, unter bestimmten Lizenzbedingungen frei erhältlich. Es wurde unter starker Ausnutzung der Vorteile der objektorientierten Programmierung geschrieben. Daher eignet sich die Darstellung durch eine graphische Sprache zur Beschreibung objektorientierter Softwaresysteme wie die *Unified Modeling Language* besonders.
Grundlage dieser Arbeit ist die Version vom 23. Februar 1999.

2.3.1 Die Klassenstruktur von SoPlex

Insbesondere die Vererbung von Basisklassen auf abgeleitete Klassen nimmt einen wichtigen Platz im logischen Aufbau des Programmes ein. Um diesen verstehen zu können, betrachten wir das leicht vereinfachte statische Klassendiagramm 2.1.

Im Zentrum der Abbildung steht die Klasse `SoPlex`. Sie erbt gleich dreifach: von `SPxLP`, `cacheLPSolver` und `SPxBasis`. `SPxLP` enthält die kompletten Informationen über das lineare Programm. Die rechte Seite, die Schranken, der Zielfunktionsvektor und die Optimierungsrichtung (Minimieren oder Maximieren) sind als Komponenten enthalten. Die Nebenbedingungsmatrix A wird gleich doppelt gespeichert, einmal zeilen- (`LPRowSet`) und einmal spaltenweise (`LPColSet`). Dies geschieht aus Gründen der Effizienz — Berechnungen von Matrix-Vektorprodukten Ax bzw. $A^T y$ sind beispielsweise in je einer der beiden Darstellungen deutlich schneller als in der anderen. Vektoren und Matrizen werden entweder dense oder sparse abgespeichert, Grundlage sind die Klassen `DVector` und `SVector`.
`cacheLPSolver` ist von `LPSolver` abgeleitet und bietet einige Verbesserungen beim Zugriff auf häufig benötigte Speicherbereiche. `LPSolver` stellt die Schnittstelle nach oben dar. Wird SoPlex, beispielsweise in Branch- and Cut-Algorithmen, als Löser eingesetzt, so ist es natürlich auch nur eine mögliche Implementierung eines LP-Solvers — nach der objektorientierten Philosophie sollte also die Möglichkeit bestehen, einen anderen Solver zu schreiben, der benutzt werden kann, ohne das darüberliegende Programm zu verändern.
`SPxBasis` schließlich beinhaltet die aktuelle Basis. Um auf die SoPlex-interne Numerierung der Variablen zugreifen zu können, ist `SPxBasis` von `SPxLP` abhängig, auch wenn auf den Pfeil aus Gründen der Übersichtlichkeit verzichtet wurde. Für jede Variable wird mittels `Desc` ein Status mitgeführt, der im wesentlichen angibt, ob die Variable in der aktuellen Basis ist oder nicht (siehe 2.3.3). Die Operationen zum Lösen der anfallenden linearen Gleichungssysteme $Ax = b$ und $A^T y = c$ werden von den Methoden `solveLeft()` und `solveRight()` einer Implementierung von `SLinSolver` vorgenommen.

Als Komponenten besitzt die Klasse `SoPlex` unter anderem vier Zeiger auf algorithmische Klassen, die allesamt widerum Zeiger auf `SoPlex` besitzen, so daß es sich um gleichberechtigte Assoziationen handelt. Die Klassen `SPxSimplifier`, `SPxStarter`, `SPxPricer` und `SPxRatioTester` sind abstrakte Klassen. Sie definieren einige Methoden als *virtual*, daher

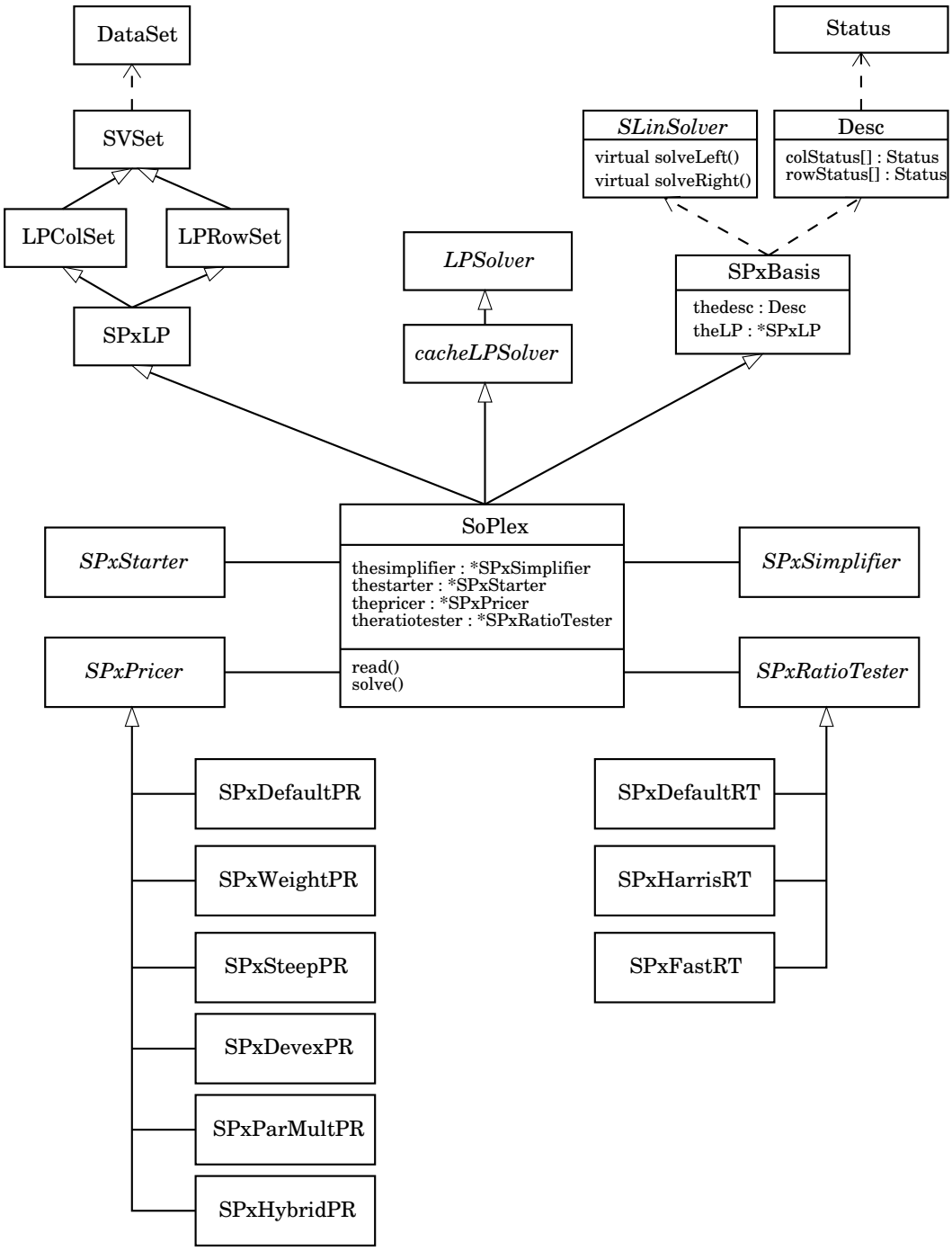


Abbildung 2.1: Statisches Klassendiagramm von SoPlex

können sie nicht selber instanziiert werden, sondern müssen abgeleitet werden von Klassen, die diese Funktionen dann implementieren.
Die gewünschten Implementierungen werden von der `solve()` Methode von `SoPlex` aufgerufen. Sie greifen auf die Daten von `SoPlex` zu und liefern ein Rechenergebnis zurück: ein vereinfachtes lineares Programm, eine Startbasis respektive einzufügender bzw. zu entfernender Index.

2.3.2 Elementare Klassen

SoPlex enthält effiziente Implementierungen für die meisten anfallenden Aufgaben. Dazu gehören

- `DataArray<T>`
  Template-Klasse zur effizienten Allokierung und Freigabe von Speicher für arrays beliebiger Datentypen.
- `IsList`
  Einfach verkettete Liste.
- `IdList`
  Doppelt verkettete Liste.
- `IdRing`
  Zu einem Ring geschlossene, doppelt verkettete Liste.
- `DataHashTable`
  Hash-Tabelle.
- `Sorter`
  Sortierklasse, implementiert Divide-and-Conquer Sortieralgorithmus.
- `CmdLine`
  Parser für die Argumentliste von Programmen.
- `Random`
  Zufallszahlengenerator.
- `Timer`
  Stoppuhr.
- `SVector, DVector, UpdateVector, UnitVector, SSVector, DSVector`
  Vektorklassen mit verschiedenen Schwerpunkten, optimiert hinsichtlich Speicherbedarf, Zugriffsmöglichkeit und/oder Funktionalität.

2.3.3 Die Basis

Die Klasse `SPxBasis` implementiert eine allgemeine Basis, wie sie in Abschnitt 1.8 vorgestellt wurde. Diese Basis besteht aus Mengen von insgesamt $n + m$ Indizes. In SoPlex werden die Mengen der allgemeinen Basis aber nicht gespeichert, sondern jeder einzelnen Variable wird ein Status zugeordnet. Allein die Basisvariablen müssen in einer Menge mitgeführt werden, da ihre Reihenfolge für die Darstellung der Basismatrix entscheidend ist.

Status der i-ten Variablen	Spaltenbasis	Zeilenbasis
P_ON_UPPER	$i \in N_u$	$i \in B_u$
P_ON_LOWER	$i \in N_l$	$i \in B_l$
P_FIXED	$i \in N_x$	$i \in B_x$
P_FREE	$i \in N_f$	$i \in B_f$
D_ON_UPPER	$i \in B_u$	$i \in N_u$
D_ON_LOWER	$i \in B_l$	$i \in N_l$
D_ON_BOTH	$i \in B_b$	$i \in N_b$
D_FREE	$i \in B_f$	$i \in N_f$
D_UNDEFINED	$i \in B_n$	$i \in N_n$

Tabelle 2.1: Zusammenhang zwischen Variablenstatus und Basen

Die Bezeichnungen entsprechen dann laut Tabelle 2.1 denen aus den Definitionen 1.43 und 1.52.
Jeder Zeile und Spalte des linearen Programmes ist eine Variable bzw. Covariable zugeordnet, die einen eindeutigen primalen **oder** dualen Status hat und speichert, ob sie eine Zeilen- oder Spaltenvariable ist. Dies ist wichtig, da Zeilenvariablen bei Benutzung einer Spaltenbasis Schlupfvariablen darstellen und andersherum.
In den Bezeichnungen gibt der erste Buchstabe an, ob es sich um einen primalen oder dualen Status handelt. Bei einem primalen Status wird die primale Variable auf eine Schranke festgelegt, bei einem dualen Status wird die duale Variable, die zu der entsprechenden Schranke gehört, auf null gesetzt, wie es die *complementary slackness* vorsieht:

Status	Spaltenvariable $l_i^{'} \leq x_i \leq u_i^{'}$	Zeilenvariable $l_i^{''} \leq A_i.x \leq u_i^{''}$
P_ON_UPPER	$x_i = u_i^{'} < \infty$	$A_i.x = u_i^{''} < \infty$
P_ON_LOWER	$x_i = l_i^{'} > -\infty$	$A_i.x = l_i^{''} > -\infty$
P_FIXED	$-\infty < l_i^{'} = x_i = u_i^{'} < \infty$	$-\infty < l_i^{''} = A_i.x = u_i^{''} < \infty$
P_FREE	$-\infty = l_i^{'} < x_i = 0 < u_i^{'} = \infty$	$-\infty = l_i^{''} < A_i.x = 0 < u_i^{''} = \infty$
D_ON_UPPER	$-\infty < l_i^{'} \neq u_i^{'} = \infty$	$-\infty < l_i^{''} \neq u_i^{''} = \infty$
D_ON_LOWER	$-\infty = l_i^{'} \neq u_i^{'} < \infty$	$-\infty = l_i^{''} \neq u_i^{''} < \infty$
D_ON_BOTH	$-\infty < l_i^{'} \neq u_i^{'} < \infty$	$-\infty < l_i^{''} \neq u_i^{''} < \infty$
D_FREE	$-\infty < l_i^{'} = u_i^{'} < \infty$	$-\infty < l_i^{''} = u_i^{''} < \infty$
D_UNDEFINED	$-\infty = l_i^{'} \neq u_i^{'} = \infty$	$-\infty = l_i^{''} \neq u_i^{''} = \infty$

Tabelle 2.2: Variablenstatus in SoPlex und Variablengrenzen

Für den Fall der Spaltenbasis liest sich Tabelle 2.2 wie folgt: ein primaler Status bezeichnet eine Nichtbasisvariable. P_ON_UPPER gibt eine Variable an, die an der oberen Schranke $u_i^{'}$ bzw. $u_i^{''}$ (Schlupfvariable) festgehalten wird, P_ON_LOWER eine an der unteren, P_FIXED eine Nichtbasisvariable mit $l_i^{'} = u_i^{'}$ bzw. $l_i^{''} = u_i^{''}$ und P_FREE eine Nichtbasisvariable ohne Schranken, die auf null gesetzt wird, wie man es aus dem Simplex ohne Schranken kennt.
Ein dualer Status entspricht einer Basisvariablen. Eigentlich würde hier ein einziger Status (B) ausreichen, doch kann man das zusätzliche Wissen über die Struktur der Schranken für eine schnellere Implementierung nutzen, außerdem ist der Aufbau der Variablen für Zeilen-

und Spaltenbasis symmetrischer. Bezeichnet v die duale Variable einer zu einer unteren Schranke gehörenden Ungleichung und w die der oberen Schranke (vergleiche Abschnitt 3.1), so bedeutet `D_ON_UPPER` $w = 0$, `D_ON_LOWER` $v = 0$, `D_ON_BOTH` $v = w = 0$, `D_FREE` $v = w$ beliebig und `D_UNDEFINED`, daß es keine zugehörige duale Variable gibt. Letzteres ist immer dann der Fall, wenn die primale Variable nicht beschränkt ist.

Für den Fall einer Zeilenbasis dreht sich die Betrachtungsweise gerade um, hier entspricht nun ein primaler Status einer Basisvariablen und ein dualer einer Nichtbasisvariablen. Für die Basisvariablen sind die zugehörigen Ungleichungen (genauer: eine der beiden Bereichsungleichungen) gerade mit Gleichheit erfüllt, wie es anschaulich bei der Herleitung gefordert wurde. Hier ist nun eine Spaltenvariable die Schlupfvariable, in der Basis ist ihr Wert gleich einer Schranke. Der duale Status dient erneut zum Festlegen der beiden dualen Variablen.

Um noch einmal zusammenzufassen: SoPlex verwaltet für ein Programm der Form (1.9) stets $n + m$ Variablen-Status. Bei einer Spaltenbasis entsprechen die ersten n den Spaltenvariablen, die letzten m den Schlupfvariablen s. Haben die Variablen einen primalen Status, so sind sie Nichtbasisvariablen und ihr Wert ergibt sich aus ihren Schranken. Haben sie einen dualen Status, so sind sie Basisvariablen und der Wert der zu den Ungleichungsbeschränkungen gehörenden dualen Variablen ist aus dem Status ableitbar.
Bei einer Zeilenbasis entsprechen m Variablen den Zeilen des LPs und n Variablen sind die "Schlupfungleichungen" $l_i' \leq \vec{e_i}^T x \leq u_i'$. Ein primaler Status bedeutet die Zugehörigkeit der Zeile zur Basis, ein dualer zur Menge der Nichtbasisvariablen.

Der duale Status wird insbesondere dazu benutzt, die Einteilung der Variablen nach Verlassen oder Eintritt in die Basis schnell und bequem vornehmen zu können. Abbildung 2.2 gibt die Zusammenhänge an.

Es gehören immer zwei Variablenstatus zusammen, bis auf die Ausnahme `D_ON_BOTH`. Hier ist in beiden Richtungen eine Fallabfrage vonnöten. Bei den beiden unten abgebildeten Statuspaaren ist ein Wechsel nur in eine Richtung möglich. Eine fixierte Variable kann beispielsweise bei Benutzung einer Spaltenbasis nicht in die Basis wechseln, sie kann aber durchaus zulässig in dieser enthalten sein und entfernt werden.

Die Klasse `SPxBasis` verwaltet aber nicht nur die Basis-Indexvektoren und die Basismatrix, sondern führt auch einfache mathematische Operationen wie Matrix-Vektor-Produkte auf der Basismatrix aus. Die etwas aufwendigeren, wie das Lösen von Gleichungssystemen, gibt die Klasse an eine Implementierung von `SLinSolver` weiter. Momentan gibt es nur eine Implementierung, die Klasse `SLUFactor`. Diese implementiert eine LU-Zerlegung der Basismatrix, die Speicherung von Update-Matrizen und eine dynamische Bestimmung des Zeitpunktes, an dem die Zerlegung neu berechnet wird, abhängig von der Anzahl der Nichtnulleinträge und Implementierungsdetails (Vektorrechner, pricing Strategie, ...).

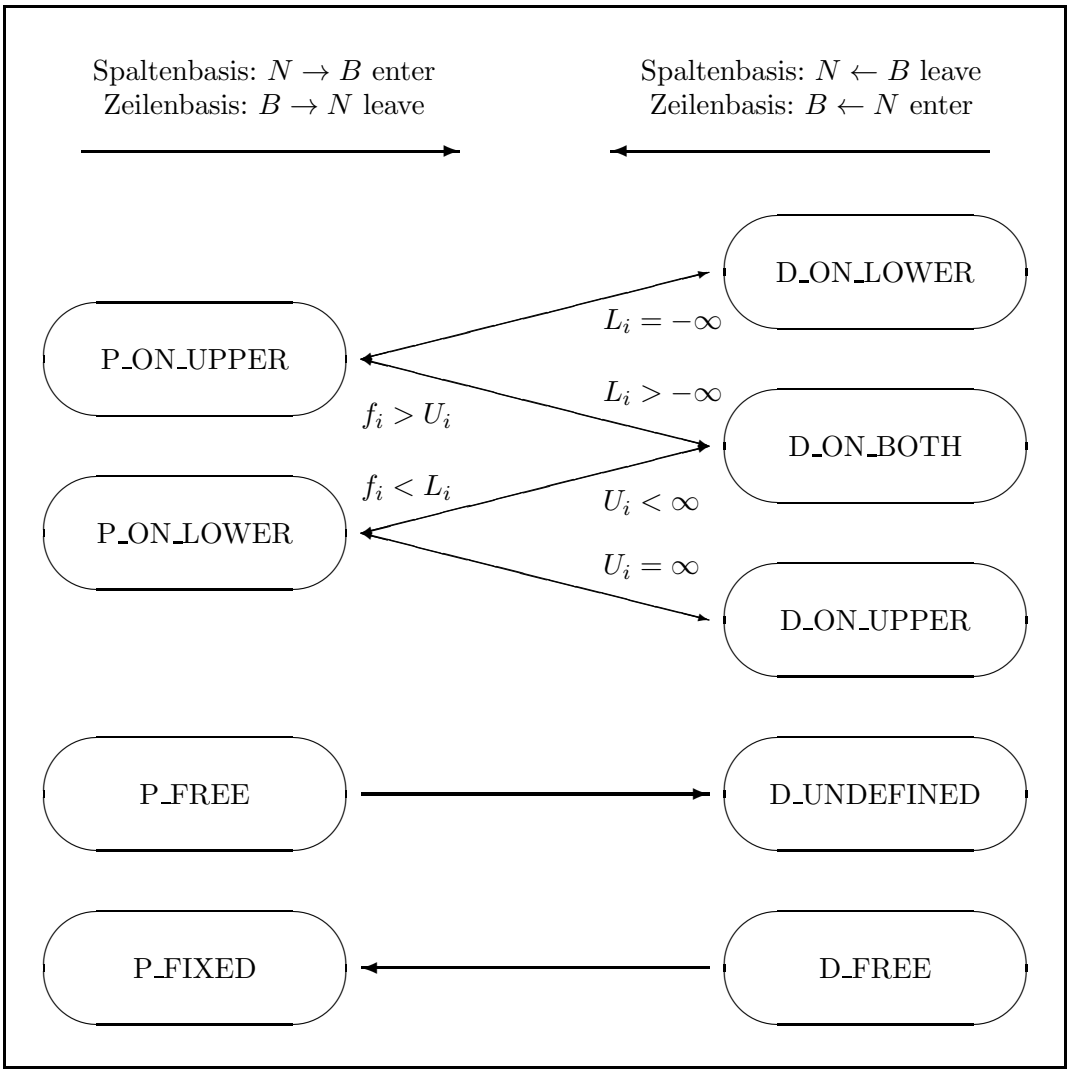


Abbildung 2.2: Transformation der Variablen bei Basisupdates

2.3.4 Die Vektoren f, g und h

Nach Tabelle 1.3 haben die Vektoren f, g und h eine jeweils andere Bedeutung und Dimension bei Spalten- und Zeilenbasis. Dies wird intern durch Methoden behandelt, die dem Anwender je nach Basisrepresentation andere Vektorobjekte zurückgeben. Diese Methoden sind

f	`SoPlex::fVec()`
g	`SoPlex::pVec()`
h	`SoPlex::coVec()`

Tabelle 2.3: Variablen in SoPlex

Die Dimension der Vektoren wird durch die Methoden `dim()` und `coDim()` bestimmt.

rep()	`ROW`	`COLUMN`
dim()	n	m
coDim()	m	n

Tabelle 2.4: Variablendimensionen in SoPlex

Der Vektor f hat demnach die Dimension `dim()`, der Vektor h dagegen `coDim()`. Der Vektor g sollte nach Tabelle 1.3 $n+m$ — also `dim() + coDim()` — Einträge besitzen. Es werden jedoch nur `coDim()` abgespeichert, da die anderen `dim()` Einträge für die Schlupfvariablen gerade dem Wert h entsprechen. Für die Spaltenbasis sieht man das so:

$$\begin{aligned} g_{\{n+1,\ldots,n+m\}} &= (A^T h)_{\{n+1,\ldots,n+m\}} \\ &= ((A^{'}\ I)^T h)_{\{n+1,\ldots,n+m\}} \\ &= h \end{aligned}$$

Für die Zeilenbasis gilt entsprechend

$$\begin{aligned} g_{\{1,\ldots,n\}} &= (A^T h)_{\{1,\ldots,n\}} \\ &= ((I\ A^{'T})^T h)_{\{1,\ldots,n\}} \\ &= h \end{aligned}$$

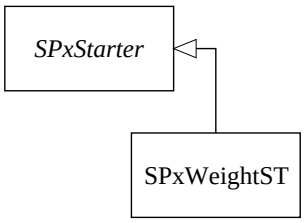
Bei Abfragen des pricing-Vektors wird also auch immer der copricing-Vektor berücksichtigt, man vergleiche beispielsweise den Quelltext des ratio tests im Anhang. Die Schranken liegen, da der Beitrag der Schlupfvariablen zur Zielfunktion null beträgt, zwischen 0 und $\pm\infty$.

2.3.5 Simplex Klassen

Nach Abbildung 2.1 gibt es fünf Klassen, in denen der Ablauf der im ersten Kapitel vorgestellten Algorithmen festgelegt ist. Zentral die Klasse `SoPlex`, von ihr aus werden dann die

Klassen `SPxStarter`, `SPxSimplifier`, `SPxPricer` und `SPxRatioTester` aufgerufen (genauer: die implementierten Klassen davon). Die in der aktuellen SoPlex -Version ausgelieferten Implementierungen sollen kurz vorgestellt werden.

- **Startbasis Klassen**
  Wie in Abschnitt 1.5.1 ausgeführt wurde, ist jede beliebige Basis als Startbasis geeignet, da Schranken geshiftet werden und dann ein Zwei-Phasen-Algorithmus ausgeführt wird. Allerdings ist es effizienter, mit einer günstig gelegenen Ecke zu beginnen. Eine solche wird mit Hilfe der Klasse `SPxStarter` bestimmt.
  Es gibt momentan nur eine Implementierung, die Klasse `SPxWeightST`. Sie verleiht den Indizes bestimmte Gewichte, die die Präferenz mit der diese in die Startbasis aufgenommen werden sollten, wiedergeben. Die Methoden dieser Klasse können wiederum überschrieben werden, so daß man die Gewichte nach anderen Kriterien verteilen kann.

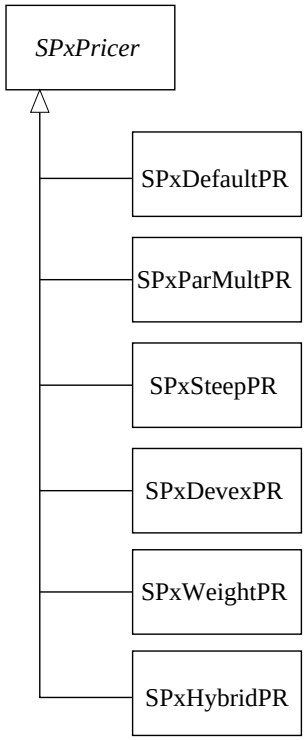


- **Preprocessing Klassen**
  Das preprocessing ist eine wirkungsvolle Methode um ein LP vor Beginn des Algorithmus so zu modifizieren, daß es effizienter gelöst werden kann. Obwohl die Schnittstelle `SPxSimplifier` vorhanden ist, gibt es bisher noch keine Implementierungsklasse.

- **Pricing Klassen**

  Das pricing wird in SoPlex von der Klasse `SPxPricer` übernommen. Es existieren die folgenden sechs Implementierungen, die den pricing Strategien aus Abschnitt 1.5.2 entsprechen:

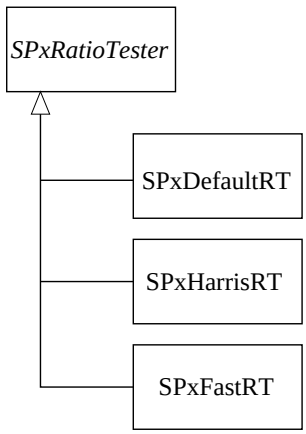
`SPxDefaultPR`	Most violation pricing
`SPxParMultPR`	Partial multiple pricing
`SPxSteepPR`	Steepest edge pricing
`SPxDevexPR`	Devex pricing
`SPxWeightPR`	Weighted pricing
`SPxHybridPR`	Hybrid pricing



- **Ratio test Klassen**

  Der ratio test wird in SoPlex von der Klasse `SPxRatioTest` übernommen. Es existieren die folgenden drei Implementierungen, die den ratio test Strategien aus Abschnitt 1.5.3 entsprechen:

`SPxDefaultRT`	Text book ratio test
`SPxHarrisRT`	Harris ratio test
`SPxFastRT`	SoPlex ratio test



  In dieser Arbeit wurde eine zusätzliche Klasse `SPxLongStepRT` hinzugefügt.

2.3.6 Parallelisierung

Die Klassenstruktur von SoPlex ist so angelegt, daß die Parallelisierung des Codes ohne großen Mehraufwand durchgeführt werden konnte. Alle algorithmischen Methoden und Datenstrukturen werden vererbt und nur die typischen Aufgaben aus dem Feld der Parallelisierung wie Synchronisation, Kommunikation und Verteilung der Arbeit auf die Prozessoren wurden neu programmiert. Während der sequentielle Quellcode mit Dokumentation ungefähr 52000 Zeilen umfaßt, liegt der Mehraufwand für den parallelen Code mit verteiltem Speicher bei 3900 Zeilen. Die Parallelisierung greift an verschiedenen Stellen:

- Maxima-Suche bei pricing und ratio test
- Matrix-Vektor Produkte und Vektoradditionen
- Block-Pivoting durch Umstrukturierung der Algorithmen um mehrere Austauschschritte auf einmal möglich zu machen
- Parallelisierung der LU-Zerlegung und der Lösung der Gleichungssysteme

Roland Wunderling kommt in seiner Arbeit [Wun96] zu dem Ergebnis, daß ein begrenzter speedup möglich ist, der allerdings sehr von der Struktur des jeweiligen Problems abhängt. Genauere Ergebnisse sind in seiner Arbeit zu finden.

2.4 Die Schnittstelle der ratio test Klassen

In diesem Abschnitt soll die virtuelle Klasse `SPxRatioTester` beschrieben werden, da eine neue ratio test Klasse von ihr abgeleitet werden muß.

```
class SPxRatioTester
{
public:
    virtual void load( SoPlex* lp ) = 0 ;
    virtual void clear( ) = 0 ;
    virtual SoPlex* solver() const = 0 ;
```

Mit der Methode `load()` wird dem Objekt ein Zeiger auf das Objekt der Klasse `SoPlex` übergeben, damit auf dessen Daten zugegriffen werden kann. Die Methode `clear()` löscht diesen Zeiger wieder, `solver()` liefert seinen Wert.

```
    virtual void setType( SoPlex::Type ) = 0 ;
    virtual  ~SPxRatioTester() {}
```

Mit `setType()` wird der ratio test Klasse mitgeteilt, daß ein Umschalten des Algorithmus stattfindet, vergleiche Abschnitt 1.5.1. `~SPxRatioTester()` ist der Destruktor.

```
    virtual int selectLeave(double& val) = 0 ;
    virtual SoPlex::Id selectEnter(double& val) = 0 ;
} ;
```

Die Methode `selectLeave()` wird vom geladenen `solver()` aufgerufen, wenn der einfügende Algorithmus ausgeführt wird. Ziel ist die Bestimmung des Index j_q, der die Basis verlassen soll. Beim Aufruf gilt

$$L - \delta \;\leq\; f \;\leq\; U + \delta$$

Der Parameter `val` enthält den Wert Φ_0 bzw. Θ_0 als maximale Schrittweite und wird mit der neu bestimmten, kleineren Schrittweite überschrieben. Dieser Wert muß das gleiche Vorzeichen wie das ursprüngliche `val` haben. Der Rückgabewert ist der Index der Variable, die die Basis verlassen soll und durch das Update eine ihrer Schranken erreicht.

Die Methode `selectEnter()` wird entsprechend aufgerufen, wenn der entfernende Algorithmus ausgeführt wird, das Ziel ist die Bestimmung eines Indizes, der in die Basis wechseln soll. Hier gilt bei Aufruf

$$l - \delta \;\leq\; g \;\leq\; u + \delta$$

Eine Implementierung dieser Klasse wird in Kapitel drei vorgestellt.

Kapitel 3

Lange Schritte im dualen Algorithmus

In diesem Kapitel soll ein neuer Ansatz motiviert werden, den ratio test beim Simplex Algorithmus durchzuführen. Dazu kehren wir zu der in der Standardliteratur üblichen Notation mit Spaltenbasis zurück und schauen uns formal die duale Zielfunktion und die Änderung ihres Wertes bei Wahl einer Schrittweite Θ des Updates der dualen Variablen λ etwas genauer an. Die Idee ist nämlich die folgende: die Updateweite Θ wird — sieht man mal von der Stabilisierung ab — gerade so gewählt, daß die duale Zulässigkeit gewahrt bleibt. Durch eine Änderung des Status der betroffenen primalen Variable ändern sich die Schranken der dualen Variable allerdings, so daß man durchaus noch dual zulässig weiter gehen kann, man macht einen sogenannten langen Schritt.
Die Untersuchung der dualen Zielfunktion dient nun der Entscheidung, wie weit dieser Schritt sein sollte. Wir werden dabei im ersten Teil sehen, daß die Ableitung der Zielfunktion für eine gegebene Richtung stückweise konstant ist und Änderungen an den Stellen Θ^i erfährt, an denen die Komponenten des Vektors $g(\Theta) - c$ ihr Vorzeichen wechseln.
Der Wert der Ableitung kann durch den Wert auf dem vorherigen Intervall und einen nichtnegativen Term beschrieben werden, so daß wir ein einfaches Kriterium für das Minimum entlang der Richtung Δh haben. Diese Idee geht zurück auf die Arbeiten von R. Gabasov et al., siehe [GKK79] und [Gab93]. Eine kurze Beschreibung der Methodik findet sich in [Tim97].
Im zweiten Teil geht es um die praktische Einbindung dieser theoretischen Überlegungen in den Simplex-Algorithmus, außerdem wird eine Übertragung des Ergebnisses auf die anderen Algorithmen untersucht. Die in das Programm eingebaute Klasse `SPxLongStepRT` wird mit dem Quelltext der wichtigsten Passagen vorgestellt. Zum Ende des Kapitels werden ein paar Bemerkungen über die Stabilisierung des ratio tests und das Vorgehen beim Update gemacht.

3.1 Theoretische Grundlagen

Betrachten wir das Spalten-LP (1.10)

$$\begin{array}{lccccc} \max & c^T x & & & & \\ \text{s.t.} & & & Ax & = & b \\ & L & \leq & x & \leq & U \end{array}$$

b ist in unserem Fall gerade 0 und wird nur der Übersichtlichkeit halber mitgeführt. Um das duale Programm leichter bestimmen zu können, schreiben wir das primale Programm als

$$\begin{array}{lrcl} \max & c^T x & & \\ \text{s.t.} & Ax & = & b \\ & -Ix & \leq & -L \\ & Ix & \leq & U \end{array}$$

Wie in Abschnitt 1.3 begründet, entsteht das duale Programm durch das Skalieren der einzelnen Zeilen, wobei Gleichungen zu freien und Ungleichungen zu vorzeichenbeschränkten Variablen führen. Der Optimierungssinn dreht sich um und Zielfunktionsvektor und rechte Seite wechseln gerade ihre Rollen. x ist formal frei, die dualen Nebenbedingungen sind also durch eine Gleichung beschränkt. Das duale Programm hat dann die Form

$$\begin{array}{lrcl} \min & b^T h + U^T w - L^T v & & \\ \text{s.t.} & A^T h + w - v & = & c \\ & v, w & \geq & 0 \end{array} \tag{3.1}$$

Wir betrachten nun die duale Variable $\lambda = (h, v, w)$. Hierbei sei h gegeben. Vektor g sei definiert durch $g := A^T h \ = v - w + c$.
Wir definieren nun v und w so, daß die duale Zulässigkeit gewährleistet ist und die Zielfunktion nicht durch eine Translation von v und w unnötig erhöht wird:

$$\begin{array}{lcl} g_j > c_j & \Longrightarrow & v_j := g_j - c_j,\ w_j := 0 \\ g_j < c_j & \Longrightarrow & v_j := 0,\ w_j := -g_j + c_j \\ g_j = c_j & \Longrightarrow & v_j := 0,\ w_j := 0 \end{array} \tag{3.2}$$

Offensichtlich ist nun $A^T h + w - v = c$. Desweiteren sei κ ein beliebiger Vektor aus dem $\mathbb{R}^n$, für den $A\kappa = b$ gilt.
Für $\Theta \geq 0$ und gegebenes Δh sei

$$\lambda(\Theta) := \lambda + \Theta\Delta\lambda = \begin{pmatrix} h(\Theta) \\ v(\Theta) \\ w(\Theta) \end{pmatrix} := \begin{pmatrix} h \\ v \\ w \end{pmatrix} + \Theta \begin{pmatrix} \Delta h \\ \Delta v \\ \Delta w \end{pmatrix}$$

ein weiterer dual zulässiger Punkt, für den Δv und Δw entsprechend (3.2) für

$$\begin{aligned} g(\Theta) := g + \Theta\Delta g &= A^T h(\Theta) \\ &\Longleftrightarrow \Delta g = A^T \Delta h \end{aligned}$$

gewählt werden, so daß $A^T h(\Theta) + w(\Theta) - v(\Theta) = c$ gilt und $v(\Theta), w(\Theta) \geq 0$.
Die Vektoren Δv und Δw sind nicht nur von Δg abhängig, sondern auch von den Vorzeichen der Komponenten von $g - c$ und $g(\Theta) - c$. Genauer:

$$
\begin{array}{lll}
g_j \le c_j \text{ und } g_j(\Theta) \le c_j \;\Rightarrow & \begin{array}{ll} w_j = c_j - g_j & w_j(\Theta) = c_j - g_j(\Theta) \\ v_j = 0 & v_j(\Theta) = 0 \end{array} & \begin{array}{ll} \Rightarrow & \Delta w_j = -\Delta g_j \\ \Rightarrow & \Delta v_j = 0 \end{array} \\
g_j \ge c_j \text{ und } g_j(\Theta) \ge c_j \;\Rightarrow & \begin{array}{ll} w_j = 0 & w_j(\Theta) = 0 \\ v_j = g_j - c_j & v_j(\Theta) = g_j(\Theta) - c_j \end{array} & \begin{array}{ll} \Rightarrow & \Delta w_j = 0 \\ \Rightarrow & \Delta v_j = \Delta g_j \end{array} \\
g_j \le c_j \text{ und } g_j(\Theta) \ge c_j \;\Rightarrow & \begin{array}{ll} w_j = c_j - g_j & w_j(\Theta) = 0 \\ v_j = 0 & v_j(\Theta) = g_j(\Theta) - c_j \end{array} & \begin{array}{ll} \Rightarrow & \Delta w_j = \frac{g_j}{\Theta} \\ \Rightarrow & \Delta v_j = \frac{g_j(\Theta)}{\Theta} \end{array} \\
g_j \ge c_j \text{ und } g_j(\Theta) \le c_j \;\Rightarrow & \begin{array}{ll} w_j = 0 & w_j(\Theta) = c_j - g_j(\Theta) \\ v_j = g_j - c_j & v_j(\Theta) = 0 \end{array} & \begin{array}{ll} \Rightarrow & \Delta w_j = \frac{-g_j(\Theta)}{\Theta} \\ \Rightarrow & \Delta v_j = \frac{-g_j}{\Theta} \end{array}
\end{array} \tag{3.3}
$$

Offensichtlich spielen die Stellen Θ eine besondere Rolle, an denen zumindest eine Komponente von $g(\Theta) - c$ das Vorzeichen wechselt. Sei im folgenden O.B.d.A. $c = 0$. Wenn von einem Vorzeichenwechsel gesprochen wird, ist damit also ein Überschreiten des Wertes c_j nach unten oder oben gemeint. Um unserem Ziel näher zu kommen, die Ableitung der Zielfunktion nach Θ bestimmen zu können, definieren wir

Definition 3.1 *Θ^i sei der Wert Θ, an dem zum i-ten Mal mindestens eine Komponente $g_j(\Theta)$ das Vorzeichen wechselt:* $\Theta^i = \max\{\Theta > \Theta^{i-1} : g_j(\Theta) g_j(\Theta^{i-1}) \ge 0 \; \forall\, j \in J\}$.
q sei der größte auftretende Index i: $\Theta^q := \max\{\Theta^i\}$, $\Theta^0 := 0$, $\Theta^{q+1} := \infty$.

Definition 3.2 *$J_i^{<,=}$ sei die Menge der Indizes j, deren zugehörige Komponenten von g diesen Vorzeichenwechsel an der Stelle Θ^i von minus nach plus erfahren:*
$J_i^{<,=} := J^{<,=}(\Theta^i) := \{j \in J : g_j < 0, g_j(\Theta^i) = 0\}$.
Analog dazu seien $J_i^{\le,=}, J_i^{\ge,\ge}, \ldots$ definiert: der erste Parameter (upper script) gibt die Beziehungsoperation zwischen g_j und 0 an, der zweite die zwischen $g_j(\Theta^i)$ und 0.
Fehlt der Index i, so gilt die zweite Beziehungsoperation zwischen $g_j(\Theta)$ und 0:
$J^{\ge,<} := J^{\ge,<}(\Theta) := \{j \in J : g_j \ge 0, g_j(\Theta) < 0\}$

Die Mengen $J^{\cdot,\cdot}$ beinhalten also Indizes, deren zugehörige Komponenten des Vektors g die Schranke c überschreiten oder nicht und entsprechen damit in gewisser Weise den Mengen $N_u, N_l, B, \ldots$. Wir formulieren ein Lemma um mit dem Wechsel der Indizes bei zunehmendem Θ besser umgehen zu können.

Lemma 3.3 *Für $0 \le i \le q-1$ und $0 \le \Theta^i \le \Theta \le \Theta^{i+1}$ gilt*

1. $J^{\ge,<} = J_i^{\ge,<} \bigcup J_i^{\ge,=}$
2. $J^{\le,>} = J_i^{\le,>} \bigcup J_i^{\le,=}$
3. $J_i^{\ge,\ge} = J^{\ge,\ge} \bigcup J_i^{\ge,=}$
4. $J_i^{\le,\le} = J^{\le,\le} \bigcup J_i^{\le,=}$

bis auf Indizes j, deren zugehörige Komponente $g_j(\Theta)$ konstant null ist.

Beweis. Folgt direkt aus der Definition der Mengen und $g_j(\Theta) = g + \Theta \Delta g$. ■

Betrachten wir nun die duale Zielfunktion an der Stelle $\lambda(\Theta)$:

$$
\begin{aligned}
\phi(\lambda(\Theta)) &= b^T(h + \Theta\Delta h) + U^T(w + \Theta\Delta w) - L^T(v + \Theta\Delta v) \\
&= \phi(\lambda) + \Theta(b^T\Delta h + U^T\Delta w - L^T\Delta v) \\
&= \phi(\lambda) + \Theta(\Delta h^T A\kappa + U^T\Delta w - L^T\Delta v)
\end{aligned}
$$

$$= \phi(\lambda) + \Theta(\Delta g^T \kappa + U^T \Delta w - L^T \Delta v)$$

Wir zerlegen diesen Ausdruck in Terme gemäß (3.3) und Definition 3.2.

$$\begin{aligned}\phi(\lambda(\Theta)) &= \phi(\lambda) + \Theta\left(\sum_{j\in J^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j\in J^{\geq,\geq}} \Delta g_j(\kappa_j - L_j)\right) \\ &\quad + \sum_{j\in J^{\geq,<}} (g_j(\Theta)(\kappa_j - U_j) - g_j(\kappa_j - L_j)) \\ &\quad + \sum_{j\in J^{\leq,>}} (g_j(\Theta)(\kappa_j - L_j) - g_j(\kappa_j - U_j))\end{aligned} \tag{3.4}$$

Uns interessiert nun, welche Änderungen die Funktion an den Stellen Θ^i erfährt. Nach Definition der Indexmengen vereinfacht sich (3.4) für $0 \leq \Theta \leq \Theta^1$ zu:

$$\phi(\lambda(\Theta)) = \phi(\lambda) + \Theta\left(\sum_{j\in J^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j\in J^{\geq,\geq}} \Delta g_j(\kappa_j - L_j)\right) \tag{3.5}$$

Im weiteren sei $1 \leq i \leq q$. Für $\Theta^i \leq \Theta \leq \Theta^{i+1}$ gilt:

$$\begin{aligned}\phi(\lambda(\Theta)) &= \phi(\lambda(\Theta^i)) + \phi(\lambda(\Theta)) - \phi(\lambda(\Theta^i)) \\ &\overset{(3.4)}{=} \phi(\lambda(\Theta^i)) + \Theta\left(\sum_{j\in J^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j\in J^{\geq,\geq}} \Delta g_j(\kappa_j - L_j)\right) \\ &\quad + \sum_{j\in J^{\geq,<}} (g_j(\Theta)(\kappa_j - U_j) - g_j(\kappa_j - L_j)) \\ &\quad + \sum_{j\in J^{\leq,>}} (g_j(\Theta)(\kappa_j - L_j) - g_j(\kappa_j - U_j)) \\ &\quad - \Theta^i\left(\sum_{j\in J_i^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j\in J_i^{\geq,\geq}} \Delta g_j(\kappa_j - L_j)\right) \\ &\quad - \sum_{j\in J_i^{\geq,<}} (g_j(\Theta^i)(\kappa_j - U_j) - g_j(\kappa_j - L_j)) \\ &\quad - \sum_{j\in J_i^{\leq,>}} (g_j(\Theta^i)(\kappa_j - L_j) - g_j(\kappa_j - U_j)) \\ &\overset{Lemma 3.3}{=} \phi(\lambda(\Theta^i)) + (\Theta - \Theta^i)\left(\sum_{j\in J^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j\in J^{\geq,\geq}} \Delta g_j(\kappa_j - L_j)\right) \\ &\quad - \Theta^i\left(\sum_{j\in J_i^{\leq,=}} \Delta g_j(\kappa_j - U_j) + \sum_{j\in J_i^{\geq,=}} \Delta g_j(\kappa_j - L_j)\right)\end{aligned}$$

$$+ \sum_{j \in J^{\geq,<}} (g_j(\Theta) - g_j(\Theta^i))(\kappa_j - U_j) - \sum_{j \in J_i^{\geq,=}} g_j(\kappa_j - L_j)$$
$$+ \sum_{j \in J^{\leq,>}} (g_j(\Theta) - g_j(\Theta^i))(\kappa_j - L_j) - \sum_{j \in J_i^{\leq,=}} g_j(\kappa_j - U_j) = \dots$$

Mit $g_j = -\Theta^i \Delta g_j$ für Indizes j mit $g_j(\Theta^i) = 0$ ergibt sich

$$\begin{aligned}
\dots &= \phi(\lambda(\Theta^i)) + (\Theta - \Theta^i) \left(\sum_{j \in J^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j \in J^{\geq,\geq}} \Delta g_j(\kappa_j - L_j) \right) \\
&\quad + \sum_{j \in J^{\geq,<}} (g_j(\Theta) - g_j(\Theta^i))(\kappa_j - U_j) \\
&\quad + \sum_{j \in J^{\leq,>}} (g_j(\Theta) - g_j(\Theta^i))(\kappa_j - L_j) \\
&= \phi(\lambda(\Theta^i)) + (\Theta - \Theta^i) \left(\sum_{j \in J^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j \in J^{\geq,\geq}} \Delta g_j(\kappa_j - L_j) \right. \\
&\quad \left. + \sum_{j \in J^{\geq,<}} \Delta g_j(\kappa_j - U_j) + \sum_{j \in J^{\leq,>}} \Delta g_j(\kappa_j - L_j) \right) \qquad (3.6)
\end{aligned}$$

Die Zielfunktion ist an den Stellen Θ^i, und damit für alle Θ, stetig, wie man diesem Ausdruck entnehmen kann. Leitet man (3.6) nach Θ ab, so erhält man als rechtsseitige Ableitung an der Stelle Θ^i für $\Theta^i \leq \Theta \leq \Theta^{i+1}$

$$\begin{aligned}
\left. \frac{d\,\phi(\lambda(\Theta))}{d\,\Theta} \right|_{\Theta = \Theta^i +} &= \sum_{j \in J^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j \in J^{\geq,\geq}} \Delta g_j(\kappa_j - L_j) \\
&\quad + \sum_{j \in J^{\geq,<}} \Delta g_j(\kappa_j - U_j) + \sum_{j \in J^{\leq,>}} \Delta g_j(\kappa_j - L_j)
\end{aligned}$$

Die linksseitige Ableitung ist gleich der rechtsseitigen an der Stelle Θ^{i-1}. Die duale Zielfunktion erfährt damit an den Stellen Θ^i eine Änderung der Steigung von

$$\begin{aligned}
&\left. \frac{d\,\phi(\lambda(\Theta))}{d\,\Theta} \right|_{\Theta = \Theta^i +} - \left. \frac{d\,\phi(\lambda(\Theta))}{d\,\Theta} \right|_{\Theta = \Theta^{i-1} +} = \\
&\qquad \sum_{j \in J_i^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) + \sum_{j \in J_i^{\geq,\geq}} \Delta g_j(\kappa_j - L_j) + \sum_{j \in J_i^{\geq,<}} \Delta g_j(\kappa_j - U_j) \\
&\qquad + \sum_{j \in J_i^{\leq,>}} \Delta g_j(\kappa_j - L_j) - \sum_{j \in J_{i-1}^{\leq,\leq}} \Delta g_j(\kappa_j - U_j) - \sum_{j \in J_{i-1}^{\geq,\geq}} \Delta g_j(\kappa_j - L_j) \\
&\qquad - \sum_{j \in J_{i-1}^{\geq,<}} \Delta g_j(\kappa_j - U_j) - \sum_{j \in J_{i-1}^{\leq,>}} \Delta g_j(\kappa_j - L_j) \\
&\overset{Lemma 3.3}{=} \sum_{j \in J_i^{\leq,=}} \Delta g_j(\kappa_j - L_j) - \Delta g_j(\kappa_j - U_j) + \sum_{j \in J_i^{\geq,=}} \Delta g_j(\kappa_j - U_j) - \Delta g_j(\kappa_j - L_j)
\end{aligned}$$

$$= \sum_{j \in J_i^{\leq,=}} \Delta g_j (U_j - L_j) + \sum_{j \in J_i^{\geq,=}} \Delta g_j (L_j - U_j)$$

Die erste Summe enthält nur nicht-negative, die zweite nur nicht-positive Faktoren, die gesamte Änderung ist also nicht-negativ:

$$\left.\frac{d\,\phi(\lambda(\Theta))}{d\,\Theta}\right|_{\Theta=\Theta^i+} - \left.\frac{d\,\phi(\lambda(\Theta))}{d\,\Theta}\right|_{\Theta=\Theta^{i-1}+} = \sum_{j \in J: g_j(\Theta^i)=0} |\Delta g_j|(U_j - L_j) \qquad (3.7)$$

Mit der

Definition 3.4 *Der Wert der Ableitung der Zielfunktion $\phi(\lambda(\Theta))$ auf dem Intervall $\left[\Theta^i, \Theta^{i+1}\right]$ wird mit α_i bezeichnet und beträgt für $1 \leq i \leq q$ nach (3.7):*

$$\alpha_i \;=\; \alpha_{i-1} + \sum_{j \in J: g_j(\Theta^i)=0} |\Delta g_j|(U_j - L_j)$$

wobei α_0 die anfängliche Steigung nach (3.5) bezeichnet:

$$\alpha_0 = \sum_{j \in J} \beta_j \qquad \text{mit } \beta_j := \begin{cases} \Delta g_j(\kappa_j - U_j) & \text{für } g_j < 0 \text{ oder } g_j = 0 \text{ und } \Delta g_j < 0 \\ \Delta g_j(\kappa_j - L_j) & \text{für } g_j > 0 \text{ oder } g_j = 0 \text{ und } \Delta g_j > 0 \\ 0 & \text{für } \Delta g_j = 0 \end{cases}$$

können wir den Abschnitt mit folgendem Satz beschließen.

Satz 3.5 *Gegeben seien reelles $\Theta \geq 0$, zwei Vektoren $y, \Delta y \in \mathbb{R}^m$ und Vektoren $v, w, \Delta v, \Delta w$ aus dem $\mathbb{R}^n$, die nach (3.2) und (3.3) bestimmt seien, so daß $\lambda = (y, v, w)$ und $\lambda(\Theta) = (y(\Theta), v(\Theta), w(\Theta))$ zulässige duale Lösungen für (3.1) sind.*
Dann wird die Zielfunktion $\phi(\lambda(\Theta))$ genau dann an der Stelle Θ^{i^*}, $1 \leq i^* \leq q$, minimiert, wenn gilt

$$\alpha_{i^*-1} \leq 0 \text{ und } \alpha_{i^*} \geq 0$$

Ist dagegen $\alpha_q < 0$, so ist die Zielfunktion unbeschränkt.

Beweis.

1. Die Zielfunktion ist stückweise linear und stetig (siehe (3.6)).
2. Die α_i's wachsen monoton, da $|\Delta g_j|(U_j - L_j) \geq 0 \;\forall\, j \in J$.
   α_i ist die Steigung auf dem Intervall $\left[\Theta^i, \Theta^{i+1}\right]$.
3. Eine stetige Funktion mit monoton wachsender Steigung nimmt ihr Minimum an der Stelle an, an der die Steigung vom Negativen ins Positive wechselt.

4. Gilt dagegen $\alpha_q < 0$, so ist die Steigung der Funktion für alle $\Theta \geq 0$ negativ, es kann daher kein Minimum geben.

■

3.2 Implementierung im Simplex

Das gefundene Ergebnis wollen wir nun auf den Algorithmus 1.3 anwenden. Nehmen wir dazu an, wir wären im Schritt 3 des Algorithmus, also beim ratio test. Nach Lemma 1.48 ist h eine dual zulässige Lösung. Der Vektor $\kappa := (f, 0) + R$ erfüllt $A\kappa = b$ und v, w sind implizit bestimmt durch den Status der zugehörigen Variablen x_i.
Mit dem folgenden Lemma wollen wir festhalten, daß die duale Zielfunktion für $\Theta \neq 0$ fällt.

Lemma 3.6 *Gegeben seien die Vektoren $f, g, h, \Delta g$ und Δh sowie der Index der die Basis verlassenden Variablen j_q und ein $\Theta_0 = \pm\infty$. Im Schritt 3 von Algorithmus 1.3 werde ein $\Theta \neq 0$ bestimmt. Dann gilt $\phi(\lambda(\Theta)) < \phi(\lambda)$.*

Beweis. Das im ratio test bestimmte Θ hat das gleiche Vorzeichen wie Θ_0. Die Steigung der dualen Zielfunktion beträgt anfangs

$$\alpha_0 \quad = \quad \sum_{j \in J} \beta_j = \sum_{j \in J} \begin{cases} \Delta g_j(\kappa_j - U_j) & \text{für } g_j < c_j \text{ oder } g_j = c_j \text{ und } \Delta g_j < 0 \\ \Delta g_j(\kappa_j - L_j) & \text{für } g_j > c_j \text{ oder } g_j = c_j \text{ und } \Delta g_j > 0 \\ 0 & \text{für } \Delta g_j = 0 \end{cases}$$

Aus der dualen Zulässigkeit folgt für $g_j < c_j$ gerade $\kappa_j \in N_u$ und damit $\kappa_j = U_j$, entsprechend $g_j > c_j \Rightarrow \kappa_j = L_j$. Für $i \in N_x \cup N_f$ ist der Ausdruck ebenfalls null. Wegen $\Delta g_B = \vec{e_q}$ bleibt nur noch ein Summand β_{j_q} übrig. Für $\Theta_0 > 0$ ergibt sich

$$\boxed{\alpha_0 = \kappa_{j_q} - L_{j_q} < 0}$$

da $\quad \Theta_0 > 0 \quad \Longleftrightarrow \quad \kappa_{j_q} < L_{j_q} \quad \Longleftrightarrow \quad g_{j_q} = c_{j_q}$ und $\Delta g_{j_q} > 0$.

Ist Θ dagegen negativ, so nimmt man die Herleitung mit umgekehrtem Vorzeichen beim Update vor und erhält

$$\boxed{\alpha_0 = U_{j_q} - \kappa_{j_q} < 0}$$

da $\quad \Theta_0 < 0 \quad \Longleftrightarrow \quad U_{j_q} < \kappa_{j_q} \quad \Longleftrightarrow \quad g_{j_q} = c_{j_q}$ und $\Delta g_{j_q} < 0$. ■

Die Einbauidee liegt nun auf der Hand. Man ändert solange an jeder Stelle Θ^i den Wert der Steigung α durch ein positives Update, wie diese Steigung negativ ist. Wird die Steigung erstmals positiv, haben wir das Minimum gefunden. Um die duale Zulässigkeit, die für die partielle Korrektheit des Algorithmus Voraussetzung ist, zu behalten, müssen einige Werte geändert werden.
Zum einen muß der Status der Nichtbasisvariable geändert werden — war sie an der oberen Schranke, kommt sie auf die untere und andersherum. Damit geht nach Definition der Schranken l und u von g ein Wechsel dieser Werte einher. Und schließlich ändert sich mit

dem Wert von x_N auch R und damit die rechte Seite $b - AR$, auch f bzw. Δf muß also verändert werden.

Anschaulich bedeutet dies, daß keine Nachbarecke mehr ausgewählt wird, wie dies beim klassischen Simplex-Verfahren der Fall ist. Die neue Basislösung liegt auf einer anderen Seite des Polyeders. Die "übersprungenen" Nichtbasisvariablen springen von einer Schranke zur anderen und der Zulässigkeitsvektor ändert sich gleich um mehrere Kanten, vergleiche Abschnitt 3.4.

3.2.1 Der LongStep Algorithmus

Algorithmus 3.1 (Entfernender Algorithmus mit LongStep ratio test)
Sei S eine Spaltenbasis von (1.9) und gelte $l \leq g \leq u$

0. INIT

Initialisiere alle Größen gemäß Tabelle 1.3

1. PRICING

Ist $L_B \leq f \leq U_B$, so ist $x = \begin{pmatrix} f \\ 0 \end{pmatrix} + R$ optimal.

Sonst wähle $j_q \in B$ mit
- Fall a) $f_q > U_{j_q}$
- Fall b) $f_q < L_{j_q}$

2.

Setze

$$\begin{aligned} \Delta h &= A_{.B}^{-T} \vec{e_q} \\ \Delta g &= A^T \Delta h \\ \text{Fall a)} \quad & \Theta_0 = -\infty,\ l_{j_q} = -\infty \\ \text{Fall b)} \quad & \Theta_0 = +\infty,\ u_{j_q} = \infty \end{aligned}$$

3. RATIO TEST

Falls $\Theta_0 > 0$ (also $f_q < L_{j_q}$) setze

$$\begin{aligned} \Theta_j &= \begin{cases} \frac{u_{n_j} - g_{n_j}}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{l_{n_j} - g_{n_j}}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases} \\ \alpha_0 &= f_q - L_{j_q} \end{aligned}$$

Falls $\Theta_0 < 0$ (also $f_q > U_{j_q}$) setze

$$\begin{aligned} \Theta_j &= \begin{cases} \frac{l_{n_j} - g_{n_j}}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{u_{n_j} - g_{n_j}}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases} \\ \alpha_0 &= U_{j_q} - f_q \end{aligned}$$

Sortiere die Θ_j in aufsteigender Reihenfolge: $|\Theta_{s_1}| \leq |\Theta_{s_2}| \leq \ldots \leq |\Theta_{s_q}|$
Berechne $\alpha_k = \alpha_{k-1} + |\Delta g_{s_k}|(U_{s_k} - L_{s_k})\ \forall\, 1 \leq k \leq q$
Ist $\alpha_q \leq 0$, so ist das LP unzulässig.

Sonst bestimme den Index s_ν mit $\alpha_{\nu-1} \leq 0$ und $\alpha_\nu \geq 0$
Setze $n_e = s_\nu, \Theta = \Theta_{n_e}$ und $\forall\, 1 \leq i < \nu$
Falls $s_i \in N_l$:
$N_l = N_l \backslash \{s_i\}$ und $N_u = N_u \cup \{s_i\}$
$l_{s_i} = -\infty,\ u_{s_i} = c_{s_i}$
Falls $s_i \in N_u$:
$N_u = N_u \backslash \{s_i\}$ und $N_l = N_l \cup \{s_i\}$
$l_{s_i} = c_{s_i},\ u_{s_i} = \infty$

4.
Entfällt.

5. UPDATE

$$\begin{aligned} h &= h + \Theta \Delta h \\ g &= g + \Theta \Delta g \end{aligned}$$

Falls $n_e = j_q$ setze

$$\begin{aligned} B_l &= B_l \setminus \{j_q\} \text{ und } B_u = B_u \cup \{j_q\} \quad \text{falls } j_q \in B_l \\ B_l &= B_l \cup \{j_q\} \text{ und } B_u = B_u \setminus \{j_q\} \quad \text{falls } j_q \in B_u \end{aligned}$$

Falls $n_e \in N$ setze

$$\begin{aligned} B &= B \setminus \{j_q\} \cup \{n_e\} \\ N &= N \setminus \{n_e\} \cup \{j_q\} \end{aligned}$$

(wobei die Einordnung der Indizes gemäß Definition erfolgt)
Aktualisiere die Vektoren l, u, R und berechne

$$f = A_{\cdot B}^{-1} (b - AR)$$

6. Gehe zu Schritt **1.**

Es soll nun die partielle Korrektheit von Algorithmus 3.1 bewiesen werden. Das leistet der folgende Satz:

Satz 3.7 *Algorithmus 3.1 arbeitet partiell korrekt: Terminiert er in Schritt 1, so ist x die optimale Lösung von LP (1.9). Terminiert er in Schritt 3, so ist das lineare Programm unzulässig. Terminiert er nicht, so werden degenerierte Schritte mit $\Theta = 0$ ausgeführt.*

Beweis. Übersprungene Indizes können keine unendlichen primalen Schranken haben, sonst wäre die zugehörige Steigung nicht mehr negativ, Indizes aus N_x haben keine dualen Schranken und werden gar nicht ausgewählt (siehe auch Abbildung 2.2). Die duale Zulässigkeit bleibt durch die Anpassung der Schranken l und u von g erhalten. Da gleichzeitig der Status der Nichtbasisvariablen angepaßt wird, sind diese mit der Definition 1.46 konsistent. Bei Terminierung in Schritt 1 gilt daher primale und duale Zulässigkeit, die optimale Lösung ist gefunden. Bei Terminierung in Schritt 3 ist eine beliebig lange Schrittweite Θ dual zulässig wählbar. Damit wird die duale Zielfunktion nach Satz 3.5 beliebig klein, wegen der schwachen Dualität ist das Programm (1.9) unzulässig.
Der ratio test wählt keine Schrittweiten aus, die zu einer Erhöhung des Zielfunktionswertes führen. Aus der endlichen Anzahl der Basislösungen folgt daher bei Nichtterminierung das Kreiseln des Algorithmus. ■

3.3 Implementierung bei einer Zeilenbasis

Der Abschnitt 3.2 wirft natürlich die Frage auf, ob solche langen Schritte auch in den anderen drei Algorithmen möglich sind. Dazu betrachten wir das Zulässigkeitsverhalten, wenn Variablen ihren Status ändern. Zur Wiederholung seien hier nochmal die vier verschiedenen Algorithmen mit der Zulässigkeit, auf der sie arbeiten und der, die sie zu erreichen suchen um Optimalität zu gewährleisten, dargestellt. S bezeichnet dabei eine Spaltenbasis, Z eine Zeilenbasis. L, U, l, u, f und g seien nach Tabelle 1.3 (Seite 50) definiert. Wichtig ist hier insbesondere, daß L, U für eine Spaltenbasis fest sind, während sie für eine Zeilenbasis vom Status der Variablen abhängen. Für l, u gilt gerade die Umkehrung.

Zulässigkeit	Primal, S	Dual, S	Primal, Z	Dual, Z
Es gilt	$L_B \leq f \leq U_B$	$l \leq g \leq u$	$l \leq g \leq u$	$L_B \leq f \leq U_B$
Optimal, wenn	$l \leq g \leq u$	$L_B \leq f \leq U_B$	$L_B \leq f \leq U_B$	$l \leq g \leq u$

Tabelle 3.1: Übersicht Zulässigkeiten der Algorithmen

Die Frage, die sich also stellt, ist die, wie man die gegebene Zulässigkeit durch eine konsistente Änderung der Schranken auch für einen längeren Schritt bewahren kann. Für den primalen Algorithmus bei einer Spaltenbasis ist dies nicht möglich, weil eine Änderung des Wertes von x_N auch eine Änderung von $f = x_B$ verursacht, wodurch die notwendige primale Zulässigkeit verloren gehen kann — die Schranken L, U sind schließlich fest.
Das gleiche Argument gilt beim primalen Zeilenalgorithmus — die Schranken l, u von g sind fest gegeben, hier ist beim ratio test kein zulässiger längerer Schritt möglich.

Anders jedoch beim dualen Zeilenalgorithmus. Stellen wir uns vor, für eine Bereichsungleichung gelte $L_{j_i} = f_i - \Phi \Delta f_i$. Nach Definition von L kann der Wert nur 0 sein. Wechselt die Variable i nun vom Status B_u zum Status B_l oder andersherum, so ändert sich gerade das erlaubte Vorzeichen von f_i und der Updatewert Φ darf zulässig einen betragsmäßig größeren Wert annehmen. Die Ursache ist ganz einfach, daß statt einer "$\leq$-Ungleichung" eine "$\geq$-Ungleichung" im aktuellen dualen Programm verwendet wird und sich damit das erlaubte Vorzeichen der dualen Variable f_i ändert. Auch hier stellt sich die Frage, wie weit ein solches Update dann sein sollte. Dazu betrachten wir das Zeilen-LP (1.12) mit der Bezeichnung $A = (I \; A^{'T})$, vergleiche Tabelle 1.3:

$$\begin{array}{lccccc} \max & c^T x & & & & \\ \text{s.t.} & l & \leq & A^T x & \leq & u \end{array}$$

und bilden zu diesem formal das duale Programm:

$$\begin{array}{lccc} \min & u^T w - l^T v & & \\ \text{s.t.} & Aw - Av & = & c \\ & v, w & \geq & 0 \end{array} \tag{3.8}$$

Gegeben sei nun ein Vektor $y \in I\!R^{n+m}$ mit $Ay = c$. Wir leiten eine zulässige Lösung v, w ab, indem wir

$$\begin{aligned} v_j = 0,\, w_j = y_j \quad &\text{wenn} \quad y_j > 0 \\ w_j = 0,\, v_j = -y_j \quad &\text{wenn} \quad y_j < 0 \\ v_j = w_j = 0 \quad &\text{wenn} \quad y_j = 0 \end{aligned}$$

setzen. Nach einem zulässigen Update $\Phi\Delta y$ mit $\Phi \geq 0$ passen wir auch Δv und Δw an und erhalten wie bei 3.3 die eindeutigen Werte

$$\begin{array}{lcll} y_j \leq 0 \text{ und } y_j(\Phi) \leq 0 & \Rightarrow & w_j = 0 \quad w_j(\Phi) = 0 & \Rightarrow \quad \Delta w_j = 0 \\ & & v_j = -y_j \quad v_j(\Phi) = -y_j(\Phi) & \Rightarrow \quad \Delta v_j = -\Delta y_j \\ y_j \geq 0 \text{ und } y_j(\Phi) \geq 0 & \Rightarrow & v_j = 0 \quad v_j(\Phi) = 0 & \Rightarrow \quad \Delta v_j = 0 \\ & & w_j = y_j \quad w_j(\Phi) = y_j(\Phi) & \Rightarrow \quad \Delta w_j = \Delta y_j \\ y_j \leq 0 \text{ und } y_j(\Phi) \geq 0 & \Rightarrow & w_j = 0 \quad w_j(\Phi) = y_j(\Phi) & \Rightarrow \quad \Delta w_j = \frac{y_j(\Phi)}{\Phi} \\ & & v_j = -y_j \quad v_j(\Phi) = 0 & \Rightarrow \quad \Delta v_j = \frac{y_j}{\Phi} \\ y_j \geq 0 \text{ und } y_j(\Phi) \leq 0 & \Rightarrow & w_j = y_j \quad w_j(\Phi) = 0 & \Rightarrow \quad \Delta w_j = -\frac{y_j}{\Phi} \\ & & v_j = 0 \quad v_j(\Phi) = -y_j(\Phi) & \Rightarrow \quad \Delta v_j = -\frac{y_j(\Phi)}{\Phi} \end{array} \tag{3.9}$$

Diese werden in die duale Zielfunktion ϕ an einer Stelle $\lambda(\Phi) = (v(\Phi), w(\Phi))$

$$\begin{aligned} \phi(\lambda(\Phi)) &= u^T(w + \Phi\Delta w) - l^T(v + \Phi\Delta v) \\ &= \phi(\lambda) + \Phi(u^T\Delta w - l^T\Delta v) \end{aligned}$$

eingesetzt. Mit Mengen $J^{\cdot,\cdot}$ nach Definition 3.2 entspricht das weitere Vorgehen dem aus Abschnitt 3.1. Die Steigung der dualen Zielfunktion läßt sich für eine Zeilenbasis und $1 \leq i \leq q$ wie folgt darstellen

$$\begin{aligned} \alpha_0 := \left.\frac{d\,\phi(\lambda(\Phi))}{d\,\Phi}\right|_{\Phi=\Phi^0+} &= \sum_{j \in J^{\leq,\leq}} \Delta y_j l_j + \sum_{j \in J^{\geq,\geq}} \Delta y_j u_j \\ \alpha_i := \left.\frac{d\,\phi(\lambda(\Phi))}{d\,\Phi}\right|_{\Phi=\Phi^i+} &= \left.\frac{d\,\phi(\lambda(\Phi))}{d\,\Phi}\right|_{\Phi=\Phi^{i-1}+} + \sum_{j \in J: y_j(\Phi^i)=0} |\Delta y_j|(u_j - l_j) \end{aligned}$$

α_i bezeichnet die Steigung auf dem Intervall $[\Phi^i, \Phi^{i+1}]$. Dieses theoretische Ergebnis soll nun auf den einfügenden Algorithmus bei Verwendung einer Zeilenbasis angewandt werden. Der Zulässigkeitsvektor f wird nach Tabelle 1.3 berechnet als

$$f = \begin{pmatrix} I \\ A' \end{pmatrix}_{.B}^{-T} c'$$

und erfüllt mit den Bezeichnungen $A = (I\ A'^T), c = c', y_B = f, y_N = 0$ das Gleichungssystem

$$Ay = c$$

Diese duale Variable kann bei geeigneter Anpassung der Schranken also dazu benutzt werden, einen längeren zulässigen Schritt auszuführen. Auch hier wollen wir wieder den kompletten Algorithmus mit modifiziertem ratio test und der Schreibweise der allgemeinen Basis angeben.

Algorithmus 3.2 (Einfügender Algorithmus mit LongStep ratio test)
Sei Z eine Zeilenbasis von (1.9) und gelte $L \leq f \leq U$

0. INIT

Initialisiere alle Größen gemäß Tabelle 1.3

1. PRICING

Ist $l \leq g \leq u$, so ist x optimal.

Sonst wähle $n_e \in N$ mit
- Fall a) $g_{n_e} > u_{n_e}$
- Fall b) $g_{n_e} < l_{n_e}$

2.

Setze

$$\begin{aligned} \Delta f &= A_{\cdot B}^{-1} a_{n_e} \\ \text{Fall a)} \quad & \Phi_0 = -\infty,\ \rho = U_{j_q} \\ \text{Fall b)} \quad & \Phi_0 = +\infty,\ \rho = L_{j_q} \end{aligned}$$

3. RATIO TEST

Falls $\Phi_0 > 0$ (also $g_{n_e} < l_{n_e}$) setze

$$\begin{aligned} \Phi_j &= \begin{cases} \frac{U_{j_i} - f_i}{\Delta f_i} & \text{falls } \Delta f_i > 0 \\ \frac{L_{j_i} - f_i}{\Delta f_i} & \text{falls } \Delta f_i < 0 \end{cases} \\ \alpha_0 &= g_{n_e} - l_{n_e} \end{aligned}$$

Falls $\Phi_0 < 0$ (also $g_{n_e} > u_{n_e}$) setze

$$\begin{aligned} \Phi_j &= \begin{cases} \frac{L_{j_i} - f_i}{\Delta f_i} & \text{falls } \Delta f_i > 0 \\ \frac{U_{j_i} - f_i}{\Delta f_i} & \text{falls } \Delta f_i < 0 \end{cases} \\ \alpha_0 &= u_{n_e} - g_{n_e} \end{aligned}$$

Sortiere die Φ_j in aufsteigender Reihenfolge: $|\Phi_{s_1}| \leq |\Phi_{s_2}| \leq \ldots \leq |\Phi_{s_q}|$
Berechne $\alpha_k = \alpha_{k-1} + |\Delta f_{s_k}|(u_{s_k} - l_{s_k})\ \forall\, 1 \leq k \leq q$
Ist $\alpha_q \leq 0$, so ist das LP unbeschränkt.

Sonst bestimme den Index s_ν mit $\alpha_{\nu-1} \leq 0$ und $\alpha_\nu \geq 0$
Setze $j_q = s_\nu, \Phi = \Phi_{j_q}$ und $\forall\, 1 \leq i < \nu$
Falls $s_i \in B_l$:
$B_l = B_l \backslash \{s_i\}$ und $B_u = B_u \cup \{s_i\}$
$U_{s_i} = L_{s_i},\ L_{s_i} = -\infty$
Falls $s_i \in B_u$:
$B_u = B_u \backslash \{s_i\}$ und $B_l = B_l \cup \{s_i\}$
$L_{s_i} = U_{s_i},\ U_{s_i} = \infty$

4.

Entfällt.

5. UPDATE

$$f = f - \Phi\Delta f + (R_{n_e} + \Phi - \rho)\, \vec{e_q}$$

Falls $j_q = n_e$ setze

$$B_l = B_l \setminus \{j_q\} \text{ und } B_u = B_u \cup \{j_q\} \quad \text{falls } j_q \in B_l$$
$$B_l = B_l \cup \{j_q\} \text{ und } B_u = B_u \setminus \{j_q\} \quad \text{falls } j_q \in B_u$$

Falls $j_q \in B$ setze

$$B = B \setminus \{j_q\} \cup \{n_e\}$$
$$N = N \setminus \{n_e\} \cup \{j_q\}$$

(wobei die Einordnung der Indizes gemäß Definition erfolgt)
Aktualisiere die Vektoren L, U, r und berechne

$$h = A_{.B}^{-T} r$$
$$g = A^T h$$

6. Gehe zu Schritt **1.**

Mit dem folgenden Lemma wollen wir die Wahl von α_0 als Startsteigung begründen und zeigen, daß durch den ratio test tatsächlich eine Abstiegsrichtung bestimmt wird.

Lemma 3.8 *Gegeben seien die Vektoren f, g, h und Δf sowie der Index der in die Basis eintretenden Variablen n_e und ein $\Phi_0 = \pm\infty$. Im Schritt 3 von Algorithmus 3.2 werde ein $\Phi = \Phi_{s_\nu} \neq 0$ bestimmt. Dann gilt $\phi(\lambda(\Phi)) < \phi(\lambda)$.*

Beweis. Das im ratio test bestimmte Φ hat das gleiche Vorzeichen wie Φ_0. Die Steigung der dualen Zielfunktion beträgt anfangs

$$\alpha_0 = \sum_{j \in J} \beta_j = \sum_{j \in J} \begin{cases} \Delta y_j l_j & \text{für } y_j < 0 \text{ oder } y_j = 0 \text{ und } \Delta y_j < 0 \\ \Delta y_j u_j & \text{für } y_j > 0 \text{ oder } y_j = 0 \text{ und } \Delta y_j > 0 \\ 0 & \text{für } \Delta y_j = 0 \end{cases}$$

Für $j \in N \setminus \{n_e\}$ ist nach Definition $y = \Delta y = 0$, die Indizes spielen daher keine Rolle. Aus der dualen Zulässigkeit folgt für Basisindizes aus $y_j < 0$ notwendig $j \in B_l \cup B_x$ und aus $y_j > 0$ $j \in B_u \cup B_x$. Für den in die Basis wechselnden Index n_e gilt $\Delta y_{n_e} = \pm 1$. Damit läßt sich die anfängliche Steigung schreiben als

$$\alpha_0 = \sum_{j \in J^{\leq,\leq}} \Delta y_j l_j + \sum_{j \in J^{\geq,\geq}} \Delta y_j u_j$$

$$= \sum_{j \in B_l} \Delta y_j l_j + \sum_{j \in B_u} \Delta y_j u_j + \sum_{j \in B_x} \Delta y_j l_j + \begin{cases} u_{n_e} & \text{wenn } \Delta y_{n_e} = 1 \\ -l_{n_e} & \text{wenn } \Delta y_{n_e} = -1 \end{cases}$$

Nach Definition von r und mit $\Phi_0 > 0 \iff g_{n_e} < l_{n_e} \iff n_e \in N_u \iff \Delta y_{n_e} < 0$ folgt

$$\alpha_0 = \Delta y^T r - l_{n_e} = \Delta y^T A_{\cdot B}^{-T} h - l_{n_e} = a_{n_e}^T h - l_{n_e} = g_{n_e} - l_{n_e}$$

$$\boxed{\alpha_0 = g_{n_e} - l_{n_e} < 0}$$

Ist Φ dagegen negativ, so nimmt man die Herleitung mit umgekehrtem Vorzeichen beim Update vor und erhält

$$\boxed{\alpha_0 = u_{n_e} - g_{n_e} < 0}$$

■

Damit ist das Konzept der langen Schritte auch für den einfügenden Algorithmus anwendbar, wenn man auf einer Zeilenbasis arbeitet. In Abschnitt 1.3.3 wurden die Vorteile des dualen Algorithmus angesprochen. Eine interessante Frage ist nunmehr, ob der einfügende Algorithmus bei einer Zeilenbasis eventuell ein anderes Verhalten beim Hinzufügen von Spalten bzw. Zeilen zeigt als der entfernende Algorithmus bei einer Spaltenbasis, so daß man das Verfahren sowohl beim branch-and-cut als auch beim branch-and-price günstig einsetzen kann, wenn man die Basisdarstellung anpaßt. In der Tat arbeiten die Algorithmen auf einer anderen Zulässigkeit, man vergleiche hierzu Tabelle 3.1. Beim Hinzufügen einer Zeile zu A bleibt eine zulässige Lösung g zulässig, beim Hinzufügen einer Spalte gilt dies für den Vektor f. Allerdings ist A für eine Zeilenbasis so definiert, daß die Transponierte der ursprünglichen Matrix A' benutzt wird, damit wird aus dem Hinzufügen einer Spalte eine zusätzliche Zeile und umgekehrt. Das Konzept der langen Schritte ist also nur beim Hinzufügen von zusätzlichen Ungleichungen nutzbar, will man eine zulässige Basislösung als neue Startbasis eines modifizierten LPs benutzen.
Aus dem gleichen Grund ist es auch nicht möglich, es für beide Algorithmen zu nutzen, wenn man das 2-Phasen-LP löst oder shifting nutzt — etwa indem man nicht nur den Algorithmustyp, sondern auch die Art der verwendeten Basisdarstellung umstellt um den anderen dualen Algorithmus auszuführen. Die gefundene Zulässigkeit wird nur beim primalen Algorithmus als Startzulässigkeit gefordert.

Die beiden Algorithmen funktionieren besser, wenn viele Variablen auf beiden Seiten beschränkt sind. Da die Ungleichungen bei einer Spaltenbasis als Schlupfvariablen und die normalen Variablen bei einer Zeilenbasis als Schlupfungleichungen verwaltet werden, nutzen beide Basisdarstellungen die Schranken von Variablen **und** Ungleichungen.

Der Vorteil, den Algorithmus 3.2 gegenüber Algorithmus 3.1 bietet, liegt darin, daß man auf einer Basis der Größe n arbeiten kann. Bei linearen Programmen mit mehr Ungleichungen als Variablen ist dies im allgemeinen wesentlich effizienter.

3.4 Updates

Betrachten wir wieder den Fall der Spaltenbasis, die Ausführungen können direkt auf eine Zeilenbasis übertragen werden. Nach Lemma 1.48 gilt nach dem Update in Algorithmus

1.3

$$f^{'} \quad = \quad A^{-1}_{.B'} \, (b - AR^{'})$$

Führt man einen langen Schritt über Indizes $k = s_i,\ 1 \leq i < \nu$ aus, so springen die Nichtbasisvariablen x_k von einer Schranke zur anderen, der Vektor R^{LS} beträgt dann nach einem solchen Schritt

$$R^{LS} \quad = \quad R^{'} + \sum_{k=s_i} \beta_k \vec{e_k} \qquad \text{mit } \beta_k = \begin{cases} (U_k - L_k) & \text{für } k \in N_l \\ (L_k - U_k) & \text{für } k \in N_u \end{cases}$$

Damit ist die in Algorithmus 1.48 verwendete Updateformel nicht mehr gültig. Wir suchen einen zusätzlichen Updatevektor Δf^{LS}, so daß $f^{LS} := f^{'} + \Delta f^{LS}$ eine gültige Lösung ist, also $f^{LS} = A^{-1}_{.B'} \, (b - AR^{LS})$ gilt. Ein gültiges Update hat wegen

$$A_{.B'}(f^{LS} - f^{'}) \quad = \quad b - AR^{LS} - b + AR^{'} = A(R^{'} - R^{LS}) = -A \sum_{k=s_i} \beta_k \vec{e_k}$$

die Form

$$\boxed{\Delta f^{LS} = -A^{-1}_{.B'} \textstyle\sum_{k=s_i} \beta_k a_k}$$

Der Zulässigkeitsvektor wechselt damit nicht mehr unbedingt zu einer Nachbarecke, sondern führt gleich über mehrere Kanten, vergleiche Lemma 1.26. Die Schrittweiten β_k sind natürlich gerade so bestimmt, daß wieder eine Ecke des Polyeders erreicht wird. Der Schritt führt über mehrere Kanten und damit in bestimmten Fällen durch das Innere des Polyeders hindurch zu einer anderen Seite.

Für ein solches Update sind eine Lösung des Gleichungssystems und mehrere Additionen und Multiplikationen notwendig, daher greift man besser auf eine komplette Neuberechnung des Vektors $f^{LS} = A^{-1}_{.B'}(b - AR^{LS})$ zurück. Der Aufwand des Updates bei Algorithmus 1.3 besteht, da der Vektor Δf auf jeden Fall für den ratio test berechnet werden muß, lediglich aus der Multplikation und Addition eines Vektors, wobei dieser in der Regel auch noch sparse ist. Aus diesem Grund ist eine Iteration bei Verwendung des LongStep ratio tests teurer als eine normale Iteration.

In der Klasse `SPxLongStepRT` wurde daher ein Mischkonstrukt gewählt — nur wenn wirklich ein langer Schritt ausgeführt wird, löst man das Gleichungssystem neu, sonst greift man auf die Update-Formel zurück.

Die Einführung dieser Unterscheidung machte einige Änderungen in den Klassen `SoPlex` und `SPxRatioTester` erforderlich. Da das Verfahren nicht ganz mit der Definition von Simplex-Algorithmen konform ist, konnte die strenge Kapselung nicht ganz beibehalten werden. In der Klasse `Soplex` wird vor dem Update nun eine Abfrage an das ratio test Objekt ausgeführt, welches Verfahren zur Berechnung von f verwendet werden soll. Die Auswirkungen auf die Rechenzeit werden in Kapitel vier dargestellt.

3.5 Stabilität

Wie die theoretische Untersuchung in Abschnitt 1.5.3 vermuten ließ und die praktischen Tests mit einer einfachen Umsetzung des Algorithmus 3.1 bestätigten (vergleiche dazu Kapitel 4), ist eine Stabilisierung notwendig um auch numerisch problematischere Programme lösen zu können.

Hier sollen drei Ansätze vorgestellt werden. Die Grundidee ist dabei allen gemein: es soll ein betragsmäßig großes $|\Delta f_q|$ bzw. $|\Delta g_{n_e}|$ ausgewählt werden um die Kondition der neuen Basismatrix klein zu halten. Für den Rest des Abschnitts betrachten wir wieder eine Spaltenbasis. Algorithmus 3.3 gibt zur Wiederholung den ratio test an, den wir aus Algorithmus 3.1 kennen:

Algorithmus 3.3 (LongStep ratio test ohne Stabilisierung)

Bestimmung des eintretenden Index

- Falls $\Theta_0 > 0$ (also $f_q < L_{j_q}$) setze
$$\Theta_j = \begin{cases} \frac{u_{n_j} - g_{n_j}}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{l_{n_j} - g_{n_j}}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases}$$
$$\alpha_0 = f_q - L_{j_q}$$
- Falls $\Theta_0 < 0$ (also $f_q > U_{j_q}$) setze
$$\Theta_j = \begin{cases} \frac{l_{n_j} - g_{n_j}}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{u_{n_j} - g_{n_j}}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases}$$
$$\alpha_0 = U_{j_q} - f_q$$
- Sortiere die Θ_j in aufsteigender Reihenfolge: $|\Theta_{s_1}| \leq |\Theta_{s_2}| \leq \ldots \leq |\Theta_{s_q}|$
- Berechne $\alpha_k = \alpha_{k-1} + |\Delta g_{s_k}|(U_{s_k} - L_{s_k}) \ \forall\, 1 \leq k \leq q$
- Ist $\alpha_q \leq 0$, so ist das LP unzulässig.
  Sonst bestimme den Index s_ν mit $\alpha_{\nu-1} \leq 0$ und $\alpha_\nu \geq 0$
- Setze $n_e = s_\nu$ und $\Theta = \Theta_{n_e}$

Anpassen der übersprungenen Indizes

- Setze $\forall\, 1 \leq i < \nu$
  Falls $s_i \in N_l$:
  $N_l = N_l \backslash \{s_i\}$ und $N_u = N_u \cup \{s_i\}$
  $l_{s_i} = -\infty,\ u_{s_i} = c_{s_i}$
  Falls $s_i \in N_u$:
  $N_u = N_u \backslash \{s_i\}$ und $N_l = N_l \cup \{s_i\}$
  $l_{s_i} = c_{s_i},\ u_{s_i} = \infty$

Der zweite Teil des Algorithmus, die Anpassung der übersprungenen Indizes, ist für die weiteren vorgestellten Varianten identisch, er wird daher nicht mehr mit angegeben.

Um Indizes mit betragsmäßig größerem $|\Delta g_j|$ bevorzugt auszuwählen, wird in Algorithmus 3.4 ein zusätzlicher Wert $\delta/\Delta g_j$ addiert bzw. subtrahiert. Der Wert Θ wird allerdings exakt bestimmt, um wieder eine Ecke des Polyeders zu erhalten. Achtung: Bei der Addition von δ muß unbedingt auf die Reihenfolge geachtet werden, um Auslöschungseffekte zu vermeiden!

Algorithmus 3.4 (LongStep ratio test mit δ)

Bestimmung des eintretenden Index

- Falls $\Theta_0 > 0$ (also $f_q < L_{j_q}$) setze
$$\Theta_j = \begin{cases} \frac{u_{n_j} - g_{n_j} + \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{l_{n_j} - g_{n_j} - \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases}$$
$$\alpha_0 = f_q - L_{j_q}$$
- Falls $\Theta_0 < 0$ (also $f_q > U_{j_q}$) setze
$$\Theta_j = \begin{cases} \frac{l_{n_j} - g_{n_j} - \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{u_{n_j} - g_{n_j} + \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases}$$
$$\alpha_0 = U_{j_q} - f_q$$
- Sortiere die Θ_j in aufsteigender Reihenfolge: $|\Theta_{s_1}| \leq |\Theta_{s_2}| \leq \ldots \leq |\Theta_{s_q}|$
- Berechne $\alpha_k = \alpha_{k-1} + |\Delta g_{s_k}|(U_{s_k} - L_{s_k}) \ \forall\, 1 \leq k \leq q$
- Ist $\alpha_q \leq 0$, so ist das LP unzulässig.
Sonst bestimme den Index s_ν mit $\alpha_{\nu-1} \leq 0$ und $\alpha_\nu \geq 0$
- Setze $n_e = s_\nu$
- Falls $\Theta_0 > 0$ setze
$$\Theta = \begin{cases} \frac{u_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} > 0 \\ \frac{l_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} < 0 \end{cases}$$
- Falls $\Theta_0 < 0$ setze
$$\Theta = \begin{cases} \frac{l_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} > 0 \\ \frac{u_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} < 0 \end{cases}$$

Diese Stabilisierungsmaßnahme ist für Probleme der Praxis nicht ausreichend, man vergleiche hierzu Kapitel 4. Algorithmus 3.5 gibt eine Variante des Harris-Test für den LongStep Algorithmus an, die sich an dem in SoPlex implementierten Verfahren orientiert.

Algorithmus 3.5 (Stabilisierter LongStep ratio test)

Bestimmung des eintretenden Index

- Falls $\Theta_0 > 0$ (also $f_q < L_{j_q}$) setze
$$\Theta_j = \begin{cases} \frac{u_{n_j} - g_{n_j} + \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{l_{n_j} - g_{n_j} - \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases}$$
$$\alpha_0 = f_q - L_{j_q}$$
- Falls $\Theta_0 < 0$ (also $f_q > U_{j_q}$) setze
$$\Theta_j = \begin{cases} \frac{l_{n_j} - g_{n_j} - \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{u_{n_j} - g_{n_j} + \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases}$$
$$\alpha_0 = U_{j_q} - f_q$$
- Setze $\texttt{maxabs} = \max\{|\Delta g_{n_j}|\}$, $\texttt{max} = \max\{\Theta_j\}$ für $\Theta_0 > 0$,
und $\texttt{max} = -\min\{\Theta_j\}$ für $\Theta_0 < 0$.

- Sortiere die Θ_j in aufsteigender Reihenfolge: $|\Theta_{s_1}| \leq |\Theta_{s_2}| \leq \ldots \leq |\Theta_{s_q}|$
- Berechne $\alpha_k = \alpha_{k-1} + |\Delta g_{s_k}|(U_{s_k} - L_{s_k}) \ \forall\, 1 \leq k \leq q$
- Ist $\alpha_q \leq 0$, so ist das LP unzulässig.
  Sonst bestimme den Index s_ν mit $\alpha_{\nu-1} \leq 0$ und $\alpha_\nu \geq 0$
- Falls $|\Delta g_{s_\nu}| \geq 10^{-5}$ `maxabs` setze $n_e = s_\nu$
  Sonst
  Setze $\nu = 0$
  Berechne n_e als den Index, für den $|\Delta g_{n_e}| \geq$ `minstab` und der Betrag der exakten Schrittweite maximal, aber kleiner gleich `max` ist. Falls keiner gefunden werden kann, Neuberechnung mit modifiziertem δ und `minstab`.
- Falls $\Theta_0 > 0$ setze
$$\Theta = \begin{cases} \frac{u_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} > 0 \\ \frac{l_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} < 0 \end{cases}$$
- Falls $\Theta_0 < 0$ setze
$$\Theta = \begin{cases} \frac{l_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} > 0 \\ \frac{u_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} < 0 \end{cases}$$

Die Parameter δ und `minstab` liegen in der Größenordnung 10^{-6} bzw. 10^{-5}, werden aber verändert, so daß der Algorithmus auf degenerierte Ecken (mehrere Schritte mit kleinem $|\Delta g_{n_e}|$ hintereinander) reagieren kann.

Wie in Abschnitt 1.5.3 beschrieben wurde, bedeutet die Relaxierung um δ eine mögliche Verletzung der Schranken um einen Betrag δ: nunmehr gilt $l_i - \delta \leq g_i \leq u_i + \delta$ anstelle von $l_i \leq g_i \leq u_i$. Führt man dies für die langen Schritte aus, werden eventuell gleich mehrere Schranken verletzt. Das bestimmte Θ kann nun durchaus ein anderes Vorzeichen besitzen als Θ_0 und damit zu einer Verschlechterung (im Optimierungssinn) des Zielfunktionswertes führen. Tritt dieser Fall auf, war auch das Ändern der Variablenstatus nicht korrekt und muß später eventuell wieder rückgängig gemacht werden, was zu einer Erhöhung der Iterationszahl führt.
Um diesen Effekt abzumildern, formulieren wir mit Algorithmus 3.6 eine weitere Heuristik, die sich nur marginal von Algorithmus 3.5 unterscheidet.

Algorithmus 3.6 (Modifizierter stabilisierter LongStep ratio test)

Bestimmung des eintretenden Index

- Falls $\Theta_0 > 0$ (also $f_q < L_{j_q}$) setze
$$\Theta_j = \begin{cases} \frac{u_{n_j} - g_{n_j} + \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{l_{n_j} - g_{n_j} - \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases}$$
$$\alpha_0 = f_q - L_{j_q}$$
- Falls $\Theta_0 < 0$ (also $f_q > U_{j_q}$) setze
$$\Theta_j = \begin{cases} \frac{l_{n_j} - g_{n_j} - \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} > 0 \\ \frac{u_{n_j} - g_{n_j} + \delta}{\Delta g_{n_j}} & \text{falls } \Delta g_{n_j} < 0 \end{cases}$$
$$\alpha_0 = U_{j_q} - f_q$$

- Setze `maxabs` $= \max\{|\Delta g_{n_j}|\}$, `max` $= \max\{\Theta_j\}$ für $\Theta_0 > 0$,
  und `max` $= -\min\{\Theta_j\}$ für $\Theta_0 < 0$.
- Sortiere die Θ_j in aufsteigender Reihenfolge: $|\Theta_{s_1}| \leq |\Theta_{s_2}| \leq \ldots \leq |\Theta_{s_q}|$
- Berechne $\alpha_k = \alpha_{k-1} + |\Delta g_{s_k}|(U_{s_k} - L_{s_k})\ \forall\, 1 \leq k \leq q$
- Ist $\alpha_q \leq 0$, so ist das LP unzulässig.
  Sonst bestimme den Index s_ν mit $\alpha_{\nu-1} \leq 0$ und $\alpha_\nu \geq 0$
- Falls $\Theta_{s_\nu} < \xi$ setze $\nu = 1$
- Falls $|\Delta g_{s_\nu}| \geq 10^{-5}$ `maxabs` setze $n_e = s_\nu$
  Sonst
  Setze $\nu = 0$
  Berechne n_e als den Index, für den $|\Delta g_{n_e}| \geq$ `minstab` und der Betrag der exakten Schrittweite maximal, aber kleiner gleich `max` ist. Falls keiner gefunden werden kann, Neuberechnung mit modifiziertem δ und `minstab`.
- Falls $\Theta_0 > 0$ setze
$$\Theta = \begin{cases} \frac{u_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} > 0 \\ \frac{l_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} < 0 \end{cases}$$
- Falls $\Theta_0 < 0$ setze
$$\Theta = \begin{cases} \frac{l_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} > 0 \\ \frac{u_{n_e} - g_{n_e}}{\Delta g_{n_e}} & \text{falls } \Delta g_{n_e} < 0 \end{cases}$$

Der Unterschied liegt in der Abfrage "Falls $\Theta_{s_\nu} < \xi$". Sie soll verhindern, daß lange Schritte ausgeführt werden, die keinen entscheidenden Fortschritt in der Zielfunktion bewirken und Variablenstatus unnötig versetzen. Denkbar ist auch eine Einbeziehung der Steigung α. Die Bestimmung des Parameters ξ muß heuristisch erfolgen, wie dies bei δ, `minstab` oder `maxcycle` auch schon der Fall ist.

3.6 Die Klasse `SPxLongStepRT`

In diesem Abschnitt soll die konkrete Implementierung des Algorithmus 3.6 in das Softwarepaket SoPlex dargestellt werden. Dazu betrachten wir die zugehörige header-Datei `spxlongsteprt.hh` und beschreiben kurz die enthaltenen Methoden:

```
#ifndef DEFspxlongrt // prevent multiple includes
#define DEFspxlongrt

#include "spxratiotester.hh"

class SPxLongStepRT : public SPxRatioTester
{
protected:
    double minStab ;
    double epsilon ;
    double delta ;
    double delta0 ;
    SoPlex* thesolver ;
```

Interne Variablen der Klasse, deren Bedeutung aus dem Namen ersichtlich sein sollte.

```
SoPlex* solver() const { return thesolver ; }
void load( SoPlex* solver ) ;
void clear() ;
int selectLeave(double& val, double enterTest = 0) ;
SoPlex::Id selectEnter(double& val, int leaveIdx=0) ;
void setType( SoPlex::Type ) ;
```

Dies sind die Implementierungen der in 2.4 beschriebenen Methoden. Die Argumentlisten der Methoden `selectLeave` und `selectEnter` mußten leicht modifiziert werden, damit den Methoden der Index q bzw. die anfängliche Steigung α_0 bekannt ist, diese sind ansonsten nur als lokale Variablen der aufrufenden Methode gespeichert.

```
void resetTols() ;
void relax() ;
void tighten() ;
```

Diese Methoden verändern die Parameter δ, `minStab` und `epsilon`.

```
int maxDelta( double& val, double& abs, UpdateVector& upd, Vector& low,
      Vector& up, int start, int incr ) ;
virtual int maxDelta( double& val, double& abs ) ;
virtual SoPlex::Id maxDelta( int& nr, double& val, double& abs ) ;
int minDelta( double& val, double& abs, UpdateVector& upd, Vector& low,
      Vector& up, int start, int incr ) ;
virtual int minDelta( double& val, double& abs, UpdateVector& upd, Vector& low,
      Vector& up ) ;
virtual int minDelta( double& val, double& abs ) ;
virtual SoPlex::Id minDelta( int& nr, double& val, double& abs ) ;
```

Diese Methoden bestimmen $\max / \min\{\Theta_j\}$.

```
int maxSelect( double& val, double& stab, double& best,
      double& bestDelta, double max, const UpdateVector& upd,
      const Vector& low, const Vector& up, int start = 0,
      int incr  = 1 ) ;
virtual int maxSelect( double& val, double& stab, double& bestDelta,
      double max) ;
virtual int maxSelect( int nr, double& val, double& stab, double& bestDelta, double max) ;
int minSelect( double& val, double& stab, double& best,
      double& bestDelta, double max, const UpdateVector& upd,
      const Vector& low, const Vector& up, int start = 0,
      int incr  = 1 ) ;
virtual int minSelect( double& val, double& stab,
      double& bestDelta, double max) ;
virtual int minSelect( int nr,double& val,double& stab,
      double& bestDelta, double max ) ;
```

Hier wird der Index, für den die Schrittweite maximal (minimal), aber kleiner gleich `max` und der Betrag des Updates größer als `stab` ist, bestimmt.

```
int minReleave( double& sel, int leave, double maxabs ) ;
int maxReleave( double& sel, int leave, double maxabs ) ;
int minShortLeave( double& sel, int leave, double max, double abs ) ;
int maxShortLeave( double& sel, int leave, double max, double abs ) ;
virtual int minReenter( double& sel, double max, double maxabs,
      SoPlex::Id id, int nr ) ;
virtual int maxReenter( double& sel, double max, double maxabs,
      SoPlex::Id id, int nr ) ;
virtual int shortEnter( SoPlex::Id& enterId, int nr, double max, double maxabs ) ;
```

Diese Methoden überprüfen die getroffene Auswahl auf numerische Stabilität und veranlassen gegebenenfalls, daß ein neuer Index ausgewählt wird.

Die oben angegebenen Attribute und Methoden entsprechen denen in `SPxFastRT`. Die neu hinzugefügten folgen.

```
int  theupdate ;
int  thecounter1, thecounter2 ;

double *thetap, *thetac ;
double alpha ;
int numbiegen ;
int * umbiegen ;
```

In `theupdate` wird festgehalten, ob ein Update des Zulässigkeitsvektors ausgeführt werden kann. `thecounter1` ist ein interner Zähler für die Anzahl der langen Schritte. `thetap` und `thetac` sind Felder, in denen die Werte Θ bzw. Φ abgespeichert werden. Es sind zwei Felder notwendig, da ein Teil der Variablen g im Vektor h gespeichert wird, vergleiche Abschnitt 2.3.4. `alpha` gibt den Wert der aktuellen Steigung an, im Feld `umbiegen` werden die Indizes mitgeführt, deren Status umgesetzt werden muß.

```
int maxDeltaLS( double& val, double& abs, UpdateVector& upd, Vector& low,
      Vector& up, int start, int incr, double *th ) ;
int minDeltaLS( double& val, double& abs, UpdateVector& upd, Vector& low,
      Vector& up, int start, int incr, double *th ) ;
virtual SoPlex::Id longStep( int& nr, double& val, double& abs, int sense ) ;
void umbasteln() ;
SoPlex::Id selectEnterCol(double& val, int leaveIdx) ;
SoPlex::Id selectEnterRow(double& val) ;

#endif // #DEFspxlongrt#
```

Die Unterscheidung der Klassen `selectEnterCol` und `selectEnterRow` wurde vorgenommen, da die langen Schritte nur bei Benutzung einer Spaltenbasis im entfernenden Modus funktionieren. Für eine Zeilenbasis wird daher die gleiche Methode aufgerufen wie in `SPxFastRT`. Die Methode `maxDeltaLS` entspricht im wesentlichen `maxDelta`, allerdings werden die ermittelten Werte hier im array `theta` gespeichert.
Die Methode `umbasteln()` führt die Änderung der Variablenstatus und der zugehörigen Schranken durch. Die Implementierung der Methoden wird im Anhang A wiedergegeben.

Kapitel 4

Tests und Ergebnisse

In diesem Kapitel sollen die erstellten ratio test Klassen an linearen Programmen getestet werden. Gute Kollektionen von Beispielen bieten die Sammlungen `netlib` und `miplib`, die unter [NET01] bzw. [MIP01] frei erhältlich sind und als Standard-LPs für das Testen neuer Algorithmen angesehen werden dürfen.

Die Tests wurden auf einem Intel Pentium III Prozessor mit 666 MHz unter Linux mit den folgenden Grundeinstellungen ausgeführt:

- Spaltenbasis (da für fast alle Probleme $n > m$ gilt)
- Steepest Edge Pricing, vergleiche Abschnitt 1.5.2
- Skaliertes LP
- Crashbasis, vergleiche Abschnitt 1.5.1
- `maxcycle`= 150, vergleiche Abschnitt 1.5.4
- $\delta = 10^{-6}$, vergleiche Abschnitt 1.5.3

In den folgenden Tabellen wurden einige Kürzel verwendet, die hier kurz erläutert werden sollen.

- **E** ist die Anzahl der Iterationen im entering Modus. Nach Tabelle 1.2 ist dies für eine Spaltenbasis der primale Algorithmus, für den keine langen Schritte ausgeführt werden können.
- **L** ist die Anzahl der Iterationen im leaving Modus. Nach Tabelle 1.2 ist dies für eine Spaltenbasis der duale Algorithmus, für den lange Schritte ausgeführt werden können. Es sind immer beide Algorithmen notwendig um das LP in zwei Phasen zu lösen, die Gesamtzahl der Iterationen ergibt sich aus der Summe **E** + **L**.
- **t** bezeichnet die benötigte Zeit in Sekunden um das Problem zu lösen
- **LS** (Lange Schritte) steht für die Anzahl der Indizes s_i, für die der Status geändert wurde
- **It** gibt die Anzahl der Iterationen an, in denen mindestens ein Variablenstatus geändert wurde

- **ENTER** als Angabe unter der Tabelle bedeutet, daß der primale Algorithmus für Phase I und der duale für Phase II benutzt wurde.
- **LEAVE** als Angabe unter der Tabelle bedeutet, daß der duale Algorithmus für Phase I und der primale für Phase II benutzt wurde.

Ein Eintrag "-" bedeutet, daß entweder keine Lösung gefunden werden konnte, weil die Matrix singulär wurde oder eine falsche Lösung ermittelt wurde.

4.1 `netlib` Tests

Um einen ersten Eindruck von der Leistungsfähigkeit des LongStep-Algorithmus zu bekommen, wurden Tests mit einer einfachen Umsetzung von Algorithmus 3.1 und den variablenbeschränkten `netlib`-Problemen durchgeführt. Ausnahme ist das Programm FORPLAN, das nicht korrekt von SoPlex eingelesen werden konnte. Die Tabelle 4.1 gibt die Kennzahlen dieser Probleme wieder, wie sie in der `README` Datei der `netlib`-Bibliothek ausgewiesen sind (SoPlex weist bei einigen Programmen geringe Unterschiede in der Anzahl der Zeilen bzw. Spalten auf).

Die Werte in den Tabellen 4.2 und 4.3 geben einen ersten Eindruck über Zuverlässigkeit und Schnelligkeit der vorgestellten ratio tests. Die drei Verfahren `SPxDefaultRT`, `SPxHarrisRT` und `SPxFastRT` sind im Programmpaket SoPlex enthalten und bauen auf den Ausführungen in Abschnitt 1.5.3 auf. Mit `SPxLongStepRT` ist hier das Verfahren 3.3 gemeint, also die nicht-stabilisierte Umsetzung des ratio test aus Algorithmus 3.1. Zu beachten ist dabei, daß für die dualen Schritte keine Updates ausgeführt werden, sondern Neuberechnungen des Lösungsvektors f.

Es sind schon einige wichtige Ergebnisse abzulesen:

- Werden keine **L**-Schritte ausgeführt, entspricht `SPxLongStepRT` dem textbook ratio test `SPxDefaultRT`. Dies stimmt mit den Ergebnissen überein.
- Es kann eine große Rolle spielen, ob man das Phase I - Problem mit dem primalen oder dem dualen Algorithmus löst (betrachte beispielsweise Problem Nr. 13).
- Einige Probleme scheinen numerisch problematisch zu sein, da sie nur von `SPxFastRT` gelöst werden konnten.
- Auch wenn `LongStepRT` weniger **L**-Iterationen benötigt, ist die Rechenzeit oft höher.

Nr	Name	Zeilen	Spalten	NNE	Ungleichungen			
1	80BAU3B	2263	9799	29063	UP	LO	FX	
2	BOEING1	351	384	3865	UP	LO		
3	BOEING2	167	143	1339	UP	LO		
4	BORE3D	234	315	1525	UP	LO	FX	
5	CAPRI	272	353	1786	UP		FX	FR
6	CYCLE	1904	2857	21322	UP			FR
7	CZPROB	6072	12230	41873	UP			
8	D6CUBE	416	6184	43888		LO		
9	ETAMACRO	401	688	2489	UP	LO	FX	
10	FINNIS	498	614	2714	UP	LO	FX	
11	FIT1D	25	1026	14430	UP			
12	FIT1P	628	1677	10894	UP			
13	FIT2D	26	10500	138018	UP			
14	FIT2P	3001	13525	60784	UP			
15	GANGES	1310	1681	7021	UP	LO		
16	GFRD-PNC	617	1092	3467	UP	LO		
17	GREENBEA	2393	5405	31499	UP	LO	FX	
18	GREENBEB	2393	5405	31499	UP	LO	FX	FR
19	GROW15	301	645	5665	UP			
20	GROW22	441	946	8318	UP			
21	GROW7	141	301	2633	UP			
22	KB2	44	41	291	UP			
23	MODSZK1	688	1620	4158				FR
24	NESM	663	2923	13988	UP	LO	FX	
25	PEROLD	626	1376	6026	UP	LO	FX	FR
26	PILOT	1442	3652	43220	UP	LO	FX	
27	PILOT.JA	941	1988	14706	UP	LO	FX	FR
28	PILOT.WE	723	2789	9218	UP	LO	FX	FR
29	PILOT4	411	1000	5145	UP		FX	FR
30	PILOTNOV	976	2172	13129	UP		FX	
31	RECIPE	92	180	752	UP	LO	FX	
32	SEBA	516	1028	4874	UP	LO		
33	SHELL	537	1775	4900	UP	LO	FX	
34	SIERRA	1228	2036	9252	UP			
35	STAIR	357	467	3857	UP		FX	FR
36	STANDATA	360	1075	3038	UP		FX	
37	STANDGUB	362	1184	3147	UP		FX	
38	STANDMPS	468	1075	3686	UP		FX	
39	TUFF	334	587	4523	UP	LO	FX	FR
40	VTP.BASE	199	203	914	UP	LO	FX	FR

Tabelle 4.1: Beschreibung der netlib Programme mit beschränkten Variablen

	DefaultRT			**HarrisRT**			**FastRT**			**LongStepRT**				
Nr	L	E	t	L	E	t	L	E	t	L	E	t	LS	It
1	3739	1478	17.84	3771	1476	23.24	3705	1551	18.71	-	-	-	-	-
2	-	-	-	297	217	0.42	300	204	0.41	-	-	-	0	0
3	51	73	0.04	52	73	0.05	46	73	0.04	53	73	0.07	6	6
4	160	0	0.04	160	0	0.04	161	0	0.05	178	0	0.07	7	5
5	286	39	0.09	289	39	0.09	280	43	0.09	206	39	0.1	141	57
6	0	992	1.98	-	-	-	0	882	1.57	0	992	1.95	0	0
7	1089	3	1.44	1046	3	1.81	1085	2	1.51	1453	3	14.52	0	0
8	683	82	5.61	720	82	7.81	601	85	5.24	734	82	16.03	0	0
9	26	491	0.3	30	460	0.29	30	514	0.33	24	491	0.29	10	4
10	281	203	0.21	276	203	0.22	282	207	0.22	294	203	0.33	11	11
11	430	0	0.27	430	0	0.42	430	0	0.35	59	0	0.16	865	51
12	612	0	1.02	617	0	0.96	606	0	1.01	721	0	1.92	437	286
13	5063	0	41.07	5073	0	61.94	4843	0	44.83	141	0	7.18	11137	136
14	7974	0	92.33	9815	0	114.92	8665	0	101.42	8336	0	137.18	21797	6701
15	1421	0	0.69	1400	0	0.64	1381	0	0.62	1239	0	1.81	323	252
16	427	19	0.27	427	19	0.31	437	13	0.29	364	19	0.37	168	25
17	-	-	-	-	-	-	2917	4206	43.82	-	-	-	0	0
18	7233	259	57.39	7183	259	65.48	7734	711	65.58	-	-	-	25	22
19	-	-	-	1965	0	3.84	1567	0	2.56	2221	0	4.4	6630	1289
20	-	-	-	3230	0	9.54	2680	0	6.31	3601	0	11.51	13931	2289
21	278	0	0.17	371	0	0.3	451	0	0.3	504	0	0.38	1034	286
22	40	0	0.01	40	0	0	40	0	0	51	0	0.01	12	11
23	659	0	0.43	654	0	0.53	664	0	0.47	650	0	1.04	0	0
24	-	-	-	3449	41	11.1	2943	9	6.25	f	f	f	2656	432
25	-	-	-	-	-	-	1459	2690	14.87	-	-	-	0	0
26	-	-	-	-	-	-	1013	6572	150.81	-	-	-	0	0
27	-	-	-	-	-	-	1575	2585	25.39	-	-	-	0	0
28	-	-	-	-	-	-	1060	2211	16.52	-	-	-	0	0
29	-	-	-	-	-	-	330	1145	4.67	-	-	-	0	0
30	-	-	-	-	-	-	1279	2581	20.21	-	-	-	0	0
31	44	3	0	44	3	0.01	47	1	0.01	42	3	0.01	11	11
32	167	1	0.08	167	1	0.07	167	1	0.07	167	4	0.12	5	3
33	523	1	0.28	523	1	0.32	532	1	0.29	514	1	0.5	24	18
34	591	0	0.23	594	0	0.25	623	0	0.24	607	0	0.51	142	67
35	249	188	0.77	248	188	0.84	248	180	0.85	254	188	0.91	10	8
36	77	4	0.02	84	4	0.02	73	4	0.03	88	4	0.04	4	4
37	78	4	0.04	84	4	0.03	72	4	0.03	79	4	0.03	2	2
38	270	4	0.08	267	4	0.08	271	4	0.07	259	4	0.16	0	0
39	247	0	0.13	233	0	0.14	217	4	0.12	-	-	-	20	12
40	140	53	0.06	164	57	0.07	145	61	0.06	106	53	0.07	31	11

Tabelle 4.2: Iterationen und Rechenzeit der ratio test Methoden, Phase I: ENTER. Long-Step Algorithmus 3.3 mit Neulösung des Gleichungssystems in jeder Iteration

	DefaultRT			**HarrisRT**			**FastRT**			**LongStepRT**				
Nr	L	E	t	L	E	t	L	E	t	L	E	t	LS	It
1	4632	295	16.42	4733	271	19.75	4504	181	15.49	-	-	-	1426	584
2	383	141	0.4	382	143	0.43	384	152	0.42	437	138	0.51	304	100
3	109	16	0.04	123	17	0.04	120	16	0.04	-	-	-	16	11
4	91	11	0.06	88	11	0.06	88	11	0.06	85	12	0.08	0	0
5	378	4	0.16	379	4	0.16	364	2	0.14	309	4	0.23	310	142
6	-	-	-	-	-	-	0	937	2.76	-	-	-	0	0
7	1092	157	4.4	1099	159	5.35	1029	172	4.11	1301	170	13.68	0	0
8	-	-	-	-	-	-	1828	7758	101.08	-	-	-	0	0
9	282	243	0.34	276	215	0.35	275	220	0.33	287	255	0.43	19	14
10	442	76	0.25	444	87	0.3	448	82	0.27	420	69	0.43	13	13
11	0	572	1.76	0	562	1.69	0	572	1.77	0	572	1.74	0	0
12	509	0	1.13	494	0	1.01	478	0	0.92	501	0	1.38	90	80
13	0	6279	193.68	0	6238	191.34	0	6428	197.2	0	6279	193.99	0	0
14	-	-	-	-	-	-	2619	2076	44.55	-	-	-	1370	1370
15	919	84	1.22	922	85	1.39	920	88	1.24	826	89	2.01	255	131
16	275	212	0.36	275	203	0.35	275	183	0.32	274	207	0.44	34	15
17	-	-	-	-	-	-	3655	6432	108.16	-	-	-	29	25
18	-	-	-	-	-	-	4010	2085	56.90	-	-	-	113	81
19	0	600	0.83	0	674	1.23	0	693	1.21	0	600	0.83	0	0
20	0	1013	3.31	-	-	-	0	1114	3.73	0	1013	3.29	0	0
21	0	287	0.19	0	285	0.28	0	264	0.17	0	287	0.19	0	0
22	0	40	0.01	0	40	0.01	0	42	0.01	0	40	0.01	0	0
23	109	0	0.15	109	0	0.18	109	0	0.16	109	0	0.24	0	0
24	1695	867	5.7	1749	879	6.43	1611	847	5.12	-	-	-	1067	202
25	-	-	-	-	-	-	882	652	6.08	-	-	-	2216	678
26	-	-	-	-	-	-	2437	1742	83.76	-	-	-	72	16
27	-	-	-	-	-	-	1094	1249	16.86	-	-	-	254	81
28	668	1546	12.09	617	1640	12.58	561	1546	11.95	-	-	-	397	178
29	321	642	2.85	345	588	2.86	282	712	3.45	-	-	-	23	14
30	-	-	-	-	-	-	1545	338	9.65	-	-	-	23	11
31	13	16	0.01	13	16	0	13	16	0.01	13	16	0.01	0	0
32	375	151	0.21	375	151	0.21	375	151	0.21	-	-	-	2	1
33	415	66	0.22	416	66	0.25	415	63	0.22	417	66	0.35	7	6
34	317	65	0.21	327	65	0.23	357	42	0.22	332	58	0.36	98	49
35	202	146	0.85	216	159	0.99	217	159	0.96	-	-	-	34	31
36	140	0	0.08	146	0	0.13	145	0	0.09	222	0	0.47	222	80
37	43	110	0.07	44	109	0.07	44	109	0.06	41	112	0.07	0	0
38	323	0	0.3	340	0	0.39	330	0	0.29	146	0	0.7	285	97
39	171	0	0.12	267	9	0.26	168	0	0.13	383	10	0.68	10	10
40	160	0	0.06	161	1	0.07	161	1	0.06	151	1	0.06	77	37

Tabelle 4.3: Iterationen und Rechenzeit der ratio test Methoden, Phase I: LEAVE. Long-Step Algorithmus 3.3 mit Neulösung des Gleichungssystems in jeder Iteration

Um die Punkte Stabilität und Schnelligkeit zu untersuchen, wurde `SPxLongStepRT` an zwei Punkten verändert. Zum einen wurde ein stabilisierendes δ (vergleiche Abschnitt 1.5.3) eingeführt. Das Verfahren entspricht damit Algorithmus 3.4. Zum zweiten wird in jeder Iteration entschieden, ob das Gleichungssystem $Af = -AR$ neu gelöst werden muß — dies ist der Fall, wenn es wirklich zu langen Schritten kam — oder ob man auf die Updateformel zurückgreifen kann. Anhand der Werte der Tabellen 4.2 und 4.3 wurde für jedes Problem ein Algorithmus für die Phase I bevorzugt, damit auch wirklich duale Iterationen ausgeführt werden. Das Ergebnis ist in Tabelle 4.4 wiedergegeben.

Die Stabilisierung hatte offenbar noch keinen Erfolg, ein Großteil der Testprobleme wird weiterhin falsch oder gar nicht gelöst. Der Geschwindigkeitsgewinn durch Updates, wenn keine langen Schritte ausgeführt werden, macht sich bemerkbar, wenn sowohl die Anzahl der Zeilen als auch die Anzahl der dualen Iterationen (LEAVE) ohne langen Schritt groß wird wie bei den Programmen 15 und 34.

Um alle Programme aus der `netlib` Bibliothek lösen zu können, mußte der `SPxLongStepRT` noch einmal umgeschrieben werden. Die Version 3.5 (vergleiche Seite 84) erreichte dieses Ergebnis, benötigte für einige Programme aber deutlich mehr Iterationen als `SPxFastRT`, das ab nun als einzige Referenz dienen soll, da `SPxDefaultRT` und `SPxHarrisRT` nicht stabil genug arbeiten. Tabelle 4.5 gibt die Werte von Algorithmus 3.5 für primalen und dualen Startalgorithmus an. Diese sollten am besten mit den folgenden Werten in den Tabellen 4.6 und 4.7 verglichen werden.

Da ein Grund für die zusätzlichen Iterationen sein könnte, daß es zu einer Art von Kreiseln mit sehr kleinen Schrittweiten Θ kommt, bei denen immer die gleichen Variablenstatus umgesetzt werden, wurde eine weitere Modifikation vorgenommen. Für kleine Schrittweiten Θ wird demnach kein langer Schritt mehr ausgeführt, sondern nach einem modifizierten Harris-Verfahren ein stabiler Index ausgewählt. Das Verfahren ist in Algorithmus 3.6 angegeben. Der Parameter ξ wurde nach einigen Testläufen gleich $500\,\delta$ gewählt.
Das Ergebnis ist für primale und duale Startbasis in den Tabellen 4.6 und 4.7 wiedergegeben.

In der Summe sind sich die beiden Algorithmen recht ähnlich mit leichten Vorteilen für `SPxFastRT`. Für bestimmte Programme kommt es aber zu starken Leistungsunterschieden, betrachte beispielsweise die Probleme 13 und 25 in Tabelle 4.6. Die Unterschiede in der Iterationszahl und damit auch in der Laufzeit sind deutlich größer, wenn der duale Algorithmus in Phase II und nicht in Phase I benutzt wird (siehe Tabelle 4.7).

Nr	Phase I	L	E	LS	IT	t_1	t_2	Speedup
1	E	1987	1478	1718	481	14.78	16.84	113 %
2	L	439	155	288	90	0.47	0.51	108 %
3	E	55	73	3	3	0.04	0.04	100 %
4	L	88	11	0	0	0.07	0.07	100 %
5	L	260	3	238	101	0.13	0.15	115 %
6	E	0	992	0	0	1.95	1.92	98 %
7	L	1055	149	0	0	6.2	6.79	109 %
8	E	796	82	0	0	10.96	11.28	102 %
9	L	265	230	14	11	0.35	0.39	111 %
10	L	418	75	13	13	0.29	0.36	124 %
11	E	63	0	860	53	0.16	0.15	93 %
12	L	469	0	50	48	0.87	0.96	110 %
13	E	154	0	11150	142	7.39	7.4	100 %
14	E	5665	0	17754	4819	92.6	95.68	103 %
15	E	1157	0	259	226	0.74	1.37	185 %
16	E	364	19	168	25	0.31	0.36	116 %
17	E	-	-	-	-	-	-	- %
18	E	-	-	-	-	-	-	- %
19	E	1151	0	2766	679	2.19	2.35	107 %
20	E	1786	0	4085	1024	4.13	4.47	108 %
21	E	503	0	966	276	0.48	0.5	104 %
22	E	33	0	8	8	0	0	100 %
23	E	664	0	0	0	0.6	0.79	131 %
24	E	-	-	-	-	-	-	- %
25	E	-	-	-	-	-	-	- %
26	E	-	-	-	-	-	-	- %
27	E	-	-	-	-	-	-	- %
28	E	-	-	-	-	-	-	- %
29	E	-	-	-	-	-	-	- %
30	E	-	-	-	-	-	-	- %
31	E	42	3	11	11	0.01	0.01	100 %
32	E	167	4	5	3	0.08	0.11	137 %
33	E	514	1	24	18	0.41	0.51	124 %
34	E	607	0	142	67	0.28	0.49	174 %
35	E	257	188	10	8	0.85	0.92	108 %
36	E	78	4	2	2	0.03	0.04	133 %
37	E	76	4	2	2	0.03	0.04	133 %
38	E	276	4	0	0	0.09	0.11	122 %
39	L	232	0	8	8	0.24	0.26	108 %
40	L	-	-	-	-	-	-	- %

Tabelle 4.4: Iterationen und Rechenzeit von Algorithmus 3.4. t_2 gibt die Zeit für Neulösung in jedem dualen Schritt an, t_1 die für Neulösung des Gleichungssystems nur bei langen Schritten.

	ENTER					LEAVE				
Nr	L	E	LS	It	t	L	E	LS	It	t
1	2234	1551	1551	362	13.48	4154	270	736	306	14.54
2	333	204	162	67	0.44	347	143	78	31	0.32
3	49	73	1	1	0.05	111	18	9	7	0.04
4	161	0	0	0	0.05	88	11	0	0	0.06
5	259	43	25	11	0.08	318	3	85	40	0.12
6	0	882	0	0	1.55	0	937	0	0	2.6
7	1085	2	0	0	1.62	1029	172	0	0	4.11
8	601	85	0	0	5.34	1828	7758	0	0	96.75
9	32	514	4	3	0.31	280	215	5	5	0.32
10	284	207	7	7	0.23	428	80	6	6	0.26
11	89	0	810	49	0.14	0	572	0	0	1.68
12	667	0	176	138	1.38	474	0	61	59	0.84
13	287	0	9656	118	7.06	0	6428	0	0	186.66
14	11659	0	11843	4470	170.55	1734	2187	1370	1370	45.6
15	1344	0	67	40	0.64	868	88	94	32	1.08
16	378	13	168	26	0.25	274	189	6	6	0.33
17	3775	5543	287	234	64.98	4352	7346	386	314	120.24
18	7345	832	226	203	59.6	9996	2381	972	794	136.33
19	711	0	518	121	0.82	0	693	0	0	1.13
20	2180	0	3331	596	4.69	0	1114	0	0	3.04
21	296	0	158	46	0.18	0	264	0	0	0.16
22	39	0	1	1	0.01	0	42	0	0	0
23	664	0	0	0	0.49	109	0	0	0	0.16
24	2040	16	4346	754	4.85	926	856	1044	306	4.3
25	14457	3652	5204	2491	66.34	931	659	199	135	5.98
26	888	6601	406	212	147.7	2224	2200	1380	665	96.92
27	22560	3758	7303	3691	149.11	999	1325	244	147	16.05
28	8704	3843	1570	998	54.52	594	1645	60	27	11.9
29	193	1151	115	46	4.24	283	664	9	7	2.83
30	1126	2581	420	238	19.02	17388	694	3426	2034	95.03
31	45	1	11	11	0.01	13	16	0	0	0
32	167	1	0	0	0.08	376	151	2	1	0.22
33	516	1	16	12	0.29	417	63	5	4	0.21
34	626	0	102	39	0.24	347	46	50	19	0.21
35	251	180	4	4	0.82	202	164	9	8	0.93
36	73	4	1	1	0.02	146	0	4	4	0.1
37	72	4	1	1	0.03	44	109	0	0	0.07
38	271	4	0	0	0.07	375	0	32	15	0.3
39	216	4	1	1	0.12	167	0	2	2	0.11
40	126	61	15	9	0.07	164	1	30	21	0.06
$\sum$	86803	31811	48506	15001	781.47	51986	39504	10304	6365	851.59

Tabelle 4.5: Iterationen und Rechenzeit von Algorithmus 3.5. Die linke Hälfte gibt die Ergebnisse für Phase I = ENTER an, die rechte die für Phase I = LEAVE. Vergleichswerte sind in der nächsten Tabelle zu finden.

	SPxFastRT			SPxLongStepRT					
Nr	L	E	t	L	E	LS	IT	t	Speedup
1	3705	1551	18.77	2264	1551	1482	370	13.71	136 %
2	300	204	0.39	304	204	76	33	0.4	97 %
3	46	73	0.04	49	73	4	3	0.04	100 %
4	161	0	0.05	161	0	0	0	0.04	125 %
5	280	43	0.09	264	43	23	10	0.09	100 %
6	0	882	1.54	0	882	0	0	1.55	99 %
7	1085	2	1.4	1085	2	0	0	1.61	86 %
8	601	85	4.8	601	85	0	0	5.41	88 %
9	30	514	0.32	30	514	0	0	0.32	100 %
10	282	207	0.21	286	207	7	7	0.21	100 %
11	430	0	0.31	90	0	808	47	0.14	221 %
12	606	0	0.99	663	0	113	103	1.2	82 %
13	4843	0	40.88	241	0	9468	79	6.48	630 %
14	8665	0	101.44	7064	0	10159	2971	103.39	98 %
15	1381	0	0.63	1362	0	56	30	0.67	94 %
16	437	13	0.27	378	13	168	26	0.24	112 %
17	2917	4206	43.34	2772	4291	216	178	46.57	93 %
18	7734	711	63.11	6792	711	134	117	53.97	116 %
19	1567	0	2.48	1108	0	687	54	1.69	146 %
20	2680	0	6.16	2621	0	2435	198	5.89	104 %
21	451	0	0.27	259	0	52	12	0.14	192 %
22	40	0	0.01	39	0	1	1	0	100 %
23	664	0	0.45	664	0	0	0	0.48	93 %
24	2943	9	5.95	3263	22	1114	189	9.09	65 %
25	1459	2690	14.48	11374	3606	1562	876	57.26	25 %
26	1013	6572	149.81	866	6598	175	91	147.17	101 %
27	1575	2585	25.11	4347	3279	256	141	48.65	51 %
28	1060	2211	16.25	4854	3641	263	135	42.46	38 %
29	330	1145	4.54	210	1151	80	28	4.28	106 %
30	1279	2581	19.94	1363	2581	31	24	20.73	96 %
31	47	1	0	45	1	11	11	0.01	100 %
32	167	1	0.07	167	1	0	0	0.06	116 %
33	532	1	0.27	516	1	16	12	0.3	90 %
34	623	0	0.24	626	0	102	39	0.24	100 %
35	248	180	0.76	251	180	4	4	0.77	98 %
36	73	4	0.02	73	4	1	1	0.03	66 %
37	72	4	0.03	72	4	1	1	0.03	100 %
38	271	4	0.08	271	4	0	0	0.07	114 %
39	217	4	0.11	217	4	0	0	0.12	91 %
40	145	61	0.07	126	61	15	9	0.06	116 %
$\sum$	50959	26544	525.68	57738	29714	29520	5800	575.57	91 %

Tabelle 4.6: Iterationen und Rechenzeit mit Verfahren 3.6, Phase I = ENTER.

	SPxFastRT			SPxLongStepRT					
Nr	L	E	t	L	E	LS	IT	t	Speedup
1	4504	181	14.43	4110	288	765	311	13.89	103 %
2	384	152	0.37	339	165	27	6	0.36	102 %
3	120	16	0.04	111	18	6	4	0.04	100 %
4	88	11	0.06	88	11	0	0	0.05	119 %
5	364	2	0.13	334	2	92	33	0.12	108 %
6	0	937	2.6	0	937	0	0	2.59	100 %
7	1029	172	3.79	1029	172	0	0	4.16	91 %
8	1828	7758	95.83	1828	7758	0	0	96.69	99 %
9	275	221	0.3	282	215	4	4	0.31	96 %
10	448	82	0.26	428	80	6	6	0.26	100 %
11	0	572	1.64	0	572	0	0	1.71	95 %
12	478	0	0.85	474	0	61	59	0.81	104 %
13	0	6428	185.07	0	6428	0	0	186.12	99 %
14	2619	2076	41.76	1734	2187	1370	1370	46.88	89 %
15	920	88	1.16	920	87	75	18	1.16	100 %
16	275	183	0.31	274	189	6	6	0.33	93 %
17	3655	6432	100.05	4001	6150	160	134	103.59	96 %
18	4010	2085	53.61	4317	1808	148	135	57.85	92 %
19	0	693	1.14	0	693	0	0	1.15	99 %
20	0	1114	3.08	0	1114	0	0	3.08	100 %
21	0	264	0.17	0	264	0	0	0.17	100 %
22	0	42	0.01	0	42	0	0	0.01	100 %
23	109	0	0.14	109	0	0	0	0.16	87 %
24	1611	847	4.86	1417	846	911	226	5.15	94 %
25	882	652	5.54	886	740	48	27	6.12	90 %
26	2437	1742	80.25	2392	1903	11	6	86.7	92 %
27	1094	1249	15.37	1180	1192	20	12	16.38	93 %
28	561	1546	11.34	558	1597	4	1	11.24	100 %
29	282	712	3.2	280	717	0	0	3.27	97 %
30	1545	338	9.17	1409	844	46	35	12.95	70 %
31	13	16	0.01	13	16	0	0	0.01	100 %
32	375	151	0.22	376	151	2	1	0.22	100 %
33	415	63	0.21	417	63	5	4	0.21	100 %
34	357	42	0.22	341	46	43	15	0.21	104 %
35	217	159	0.9	217	159	0	0	0.91	98 %
36	145	0	0.09	145	0	1	1	0.09	100 %
37	44	109	0.07	44	109	0	0	0.07	100 %
38	330	0	0.26	322	0	4	4	0.27	96 %
39	168	0	0.11	168	0	0	0	0.12	91 %
40	161	1	0.06	153	0	21	17	0.05	119 %
$\sum$	31743	37136	638.68	30696	37563	3836	2435	665.46	95 %

Tabelle 4.7: Iterationen und Rechenzeit mit Verfahren 3.6, Phase I = LEAVE.

4.2 miplib Tests

Der Algorithmus 3.6 soll in diesem Abschnitt anhand einer zweiten Menge von Testproblemen auf Stabilität und Effizienz getestet werden. In der miplib-Bibliothek sind gemischtganzzahlige Probleme enthalten, deren LP-Relaxationen hier betrachtet werden. Die Anzahlen der Zeilen und Spalten kann man Tabelle 4.8 entnehmen. Die Programme air05 und stein45 konnten nicht von SoPlex gelesen werden, daher wurden sie nicht untersucht.

Die Ergebnisse in den Tabellen 4.9 und 4.10 entsprechen größtenteils denen aus dem vorangegangenen Abschnitt. Das Programm 25 wurde in Tabelle 4.9 nicht bei der Summenbildung berücksichtigt, da es bei SPxLongStepRT zu einer falschen Lösung kam.

Nr	Name	Zeilen	Spalten
1	10teams	230	2025
2	air03	124	10757
3	air04	823	8904
4	arki001	1048	1388
5	bell3a	123	133
6	bell5	91	104
7	blend2	274	353
8	cap6000	2176	6000
9	dano3mip	3202	13873
10	danoint	664	521
11	dcmulti	290	548
12	dsbmip	1182	1886
13	egout	98	141
14	enigma	21	100
15	fast0507	507	63009
16	fiber	363	1298
17	fixnet6	478	878
18	flugpl	18	18
19	gen	780	870
20	gesa2	1392	1224
21	gesa2_o	1248	1224
22	gesa3	1368	1152
23	gesa3_o	1224	1152
24	gt2	29	188
25	harp2	112	2993
26	khb05250	101	1350
27	l152lav	97	1989
28	lseu	28	89
29	misc03	96	160

Nr	Name	Zeilen	Spalten
30	misc06	820	1808
31	misc07	212	260
32	mitre	2054	10724
33	mod008	6	319
34	mod010	146	2655
35	mod011	4480	10958
36	modglob	291	422
37	noswot	182	128
38	nw04	36	87482
39	p0033	16	33
40	p0201	133	201
41	p0282	241	282
42	p0548	176	548
43	p2756	755	2756
44	pk1	45	86
45	pp08a	136	240
46	pp08aCUTS	246	240
47	qiu	1192	840
48	qnet1	503	1541
49	qnet1_o	456	1541
50	rentacar	6803	9557
51	rgn	24	180
52	rout	291	556
53	set1ch	492	712
54	seymour	4944	1372
55	stein27	118	27
56	vpm1	234	378
57	vpm2	234	378

Tabelle 4.8: Beschreibung und Numerierung der miplib Programme

	SPxFastRT			SPxLongStepRT					
Nr	L	E	t	L	E	LS	IT	t	Speedup
1	816	0	1.67	1051	0	304	92	2.85	58 %
2	533	0	3.67	391	0	514	172	2.5	146 %
3	3759	0	42.61	4329	0	6916	2047	47.64	89 %
4	1026	13	1.68	1082	19	949	102	1.81	92 %
5	81	0	0.01	81	0	0	0	0.02	50 %
6	56	0	0.01	57	0	2	2	0.01	100 %
7	55	0	0.04	50	1	2	1	0.03	133 %
8	5813	0	25.04	296	0	5867	38	2.22	1127 %
9	27435	8278	921.25	29125	8806	43	2	1085.41	84 %
10	734	31	1.32	891	24	11	2	1.78	74 %
11	247	25	0.09	250	25	44	34	0.11	81 %
12	1693	683	4.48	1659	704	24	8	4.95	90 %
13	88	40	0.02	88	40	0	0	0.01	200 %
14	76	0	0.02	58	17	27	7	0.01	200 %
15	2367	785	160.67	4097	785	2792	791	298.26	53 %
16	181	0	0.07	192	0	47	22	0.08	87 %
17	188	119	0.13	188	119	0	0	0.13	100 %
18	3	7	0	3	7	0	0	0.01	100 %
19	298	0	0.1	313	0	20	13	0.12	83 %
20	537	152	0.42	534	152	30	26	0.45	93 %
21	517	144	0.35	542	144	48	42	0.39	89 %
22	605	126	0.44	611	126	31	26	0.47	93 %
23	477	295	0.47	459	295	45	41	0.46	102 %
24	42	0	0.01	37	0	27	4	0.01	100 %
25	3210	811	3.68	-	-	-	-	-	- %
26	87	0	0.03	88	0	5	5	0.03	100 %
27	450	0	0.68	486	0	804	250	0.61	111 %
28	34	0	0.01	33	0	16	6	0.01	100 %
29	34	0	0.01	41	0	11	9	0.01	100 %
30	464	522	0.86	463	522	3	3	0.88	97 %
31	66	0	0.05	100	0	20	12	0.07	71 %
32	7085	690	12.39	3764	690	4882	287	8.02	154 %
33	19	0	0.01	13	0	8	4	0	100 %
34	577	0	1.24	729	0	1067	300	1.18	105 %
35	2807	0	7.08	2754	0	104	96	6.89	102 %
36	277	10	0.08	277	10	0	0	0.09	88 %
37	96	0	0.02	96	0	0	0	0.03	66 %
38	140	0	17.18	156	0	124	53	19.8	86 %
39	30	0	0	19	0	14	4	0	100 %

Tabelle 4.9: Iterationen und Rechenzeit mit Verfahren 3.6, Phase I = ENTER.

	SPxFastRT			SPxLongStepRT					
Nr	L	E	t	L	E	LS	IT	t	Speedup
40	30	27	0.01	28	27	1	1	0.03	33 %
41	99	22	0.03	92	22	65	24	0.03	100 %
42	118	0	0.05	117	0	27	2	0.05	100 %
43	90	0	0.07	42	0	48	1	0.07	100 %
44	72	0	0.01	72	0	0	0	0.02	50 %
45	85	0	0.01	85	0	0	0	0.01	100 %
46	145	72	0.07	139	72	2	2	0.05	140 %
47	20	92	0.18	19	92	0	0	0.19	94 %
48	275	0	0.2	340	0	17	13	0.29	68 %
49	377	0	0.16	377	0	1	1	0.17	94 %
50	3485	765	26.34	4328	1343	133	102	47.53	55 %
51	90	0	0.01	77	0	17	5	0.01	100 %
52	0	260	0.11	0	260	0	0	0.13	84 %
53	313	0	0.09	313	0	0	0	0.08	112 %
54	4212	2966	118.17	4195	2966	69	58	115.95	101 %
55	24	15	0.02	36	15	0	0	0.02	100 %
56	186	0	0.04	188	0	0	0	0.05	79 %
57	225	0	0.06	228	0	8	2	0.05	119 %
$\sum$	69639	16139	1349.84	66082	17283	25248	4713	1652.12	81 %

Tabelle 4.9: Iterationen und Rechenzeit mit Verfahren 3.6, Phase I = ENTER, Fortsetzung.

	SPxFastRT			SPxLongStepRT					
Nr	L	E	t	L	E	LS	IT	t	Speedup
1	2410	130	6.56	2745	148	1076	176	8.9	73 %
2	293	490	10.45	249	535	348	105	10.85	96 %
3	4603	3125	123.83	5341	2813	10943	2376	117.66	105 %
4	1257	221	2.94	2503	225	2792	501	5.98	49 %
5	90	0	0.01	90	0	0	0	0.01	100 %
6	70	0	0.01	70	0	0	0	0.01	100 %
7	108	0	0.05	108	0	0	0	0.05	100 %
8	48	269	2.46	48	269	0	0	2.5	98 %
9	30321	8256	1060.27	30321	8256	0	0	1122.64	94 %
10	969	53	2.15	969	53	0	0	2.24	95 %
11	304	6	0.11	307	4	25	21	0.11	100 %
12	868	258	1.65	879	305	2	1	1.74	94 %
13	94	0	0	94	0	0	0	0.01	100 %
14	56	0	0.01	69	0	42	5	0.01	100 %
15	4377	0	207.95	6358	0	8720	1566	371.82	55 %
16	147	30	0.09	133	36	55	20	0.08	112 %
17	151	33	0.07	151	33	0	0	0.07	100 %
18	8	0	0	8	0	0	0	0	100 %
19	235	150	0.14	244	150	17	9	0.14	100 %
20	551	0	0.33	551	0	11	10	0.34	97 %
21	426	22	0.28	430	22	21	20	0.3	93 %
22	651	0	0.38	653	0	6	6	0.39	97 %
23	502	61	0.34	505	61	42	41	0.37	91 %
24	48	0	0.01	41	0	31	7	0	100 %
25	0	517	0.96	0	517	0	0	0.96	100 %
26	94	0	0.03	94	0	5	5	0.03	100 %
27	418	249	1.37	580	381	522	118	2.05	66 %
28	34	0	0	33	0	16	6	0.01	100 %
29	36	11	0.02	45	13	3	3	0.02	100 %
30	768	7	0.71	769	7	5	5	0.71	100 %
31	72	95	0.22	79	84	1	1	0.19	115 %
32	4038	607	19.04	2938	644	2505	244	20.93	90 %
33	20	0	0.01	14	0	8	4	0.01	100 %
34	576	226	2.48	643	185	864	254	2.07	119 %
35	0	1880	7.3	0	1880	0	0	7.31	99 %
36	245	15	0.1	240	15	23	19	0.11	90 %
37	3	26	0.01	3	26	0	0	0.01	100 %
38	9	176	39.26	9	176	0	0	38.99	100 %
39	30	0	0	19	0	14	4	0	100 %

Tabelle 4.10: Iterationen und Rechenzeit mit Verfahren 3.6, Phase I = LEAVE.

	SPxFastRT			SPxLongStepRT					
Nr	L	E	t	L	E	LS	IT	t	Speedup
40	80	0	0.02	70	0	65	24	0.01	200 %
41	122	0	0.03	79	0	110	40	0.01	299 %
42	118	0	0.05	117	0	27	2	0.05	100 %
43	90	0	0.06	42	0	48	1	0.05	119 %
44	71	0	0.01	71	0	0	0	0.02	50 %
45	77	0	0.01	77	0	0	0	0.01	100 %
46	145	0	0.05	147	0	1	1	0.04	125 %
47	711	282	2.48	711	282	0	0	2.56	96 %
48	511	0	0.64	495	0	18	14	0.66	96 %
49	495	0	0.36	495	0	0	0	0.4	89 %
50	1998	1465	37.97	2073	1255	42	38	34.01	111 %
51	39	17	0.01	33	18	6	3	0.01	100 %
52	0	403	0.21	0	403	0	0	0.21	100 %
53	294	0	0.07	294	0	0	0	0.07	100 %
54	2509	0	23.7	2895	0	180	96	29.68	79 %
55	33	0	0.01	33	0	0	0	0.01	100 %
56	155	0	0.04	162	0	20	4	0.04	100 %
57	190	0	0.06	191	0	1	1	0.07	85 %
$\sum$	62568	19080	1557.38	66318	18796	28615	5751	1787.53	87 %

Tabelle 4.10: Iterationen und Rechenzeit mit Verfahren 3.6, Phase I = LEAVE, Fortsetzung.

4.3 Zusammenfassung

Diese Arbeit hat gezeigt, daß es möglich ist, den Algorithmus zu stabilisieren und einen auf ihm basierenden zuverlässigen Löser zu erhalten. Für bestimmte Probleme mit vielen zu beiden Seiten beschränkten Variablen und Ungleichungen ist die Verringerung von Iterationen und Rechenzeit enorm.

Kommt es zu keinen langen Schritten — etwa weil das lineare Programm keine oder wenig beidseitig beschränkte Variablen enthält, so spielt der Mehraufwand für die Berechnung der Θ_j und von α für kleine bis mittlere Dimensionen keine große Rolle. Die Umsetzung der theoretischen Ideen ist damit sehr effizient gelungen und der zu erwartende Gewinn durch weitere Änderungen (denkbar wäre beispielsweise eine Vorabspeicherung der Werte $U_j - L_j$) dürfte gering ausfallen. Kennt man die ungünstige Struktur des zu behandelnden Problems, sollte man aber trotzdem besser auf Standardmethoden zurückgreifen. Denkbar ist hier eine automatische Einstellung je nach Anzahl der beschränkten Variablen und Ungleichungen.

Genauer untersucht werden muß allerdings, wann es zu einer Zunahme der Iterationszahl kommt und wie man dies gegebenenfalls verhindern kann. Die Auswirkungen der recht einfachen zusätzlichen Abfrage in Algorithmus 3.6 — man vergleiche die Anzahl der Iterationen des LongStep ratio tests in den Tabellen 4.5 und 4.6, 4.7 — begründen die Vermutung, daß hier wesentlich bessere Heuristiken gefunden werden können.

Ein anderer Ansatz, das Verfahren weiter zu verbessern, liegt in der Berücksichtigung der "übersprungenen" Indizes bei der Auswahl des stabilsten Indizes. Der lange Schritt muß ja nicht unbedingt so ausgeführt werden, daß die duale Zielfunktion lokal minimiert wird — kann man einen Index finden, der einen nicht viel schlechteren Gewinn in der Zielfunktion bringt und ein betragsmäßig deutlich größeres $|\Delta g_{n_e}|$ besitzt, so sollte dieser vorzuziehen sein.
Unter Umständen kann man dadurch für viele Schritte auf die Berechnung der durch δ modifizierten Θ_j verzichten und an ihrer Stelle die korrekten Schrittweiten betrachten — die Verletztung der Schranken kann schließlich zu einem Rückschritt in der Zielfunktion und damit gleich zu vielen "falsch" ausgeführten langen Schritten führen.

Die vorgestellten Ergebnisse sind insgesamt sehr ermutigend, das Verfahren der langen Schritte weiter zu untersuchen.

Anhang A

Quellcode

In diesem Anhang sollen die wichtigsten Methoden der Klasse `SPxLongStepRT` wiedergegeben werden.

```
void SPxLongStepRT::umbasteln() {

// Nichtbasisvariablengrenzen columns
    const double*  ub     = thesolver->upper();
    const double*  lb     = thesolver->lower() ;
// Nichtbasisvariablengrenzen rows
    const double*  rhs     = thesolver->rhs() ;
    const double*  lhs     = thesolver->lhs() ;

    int i, j ;
    SPxBasis::Desc& ds = thesolver->desc() ;
    SPxBasis::Desc::Status stat ;
    SoPlex::Id myId ;
    int basvar = 0 ;


// Setze Zustaende um
    for (j = 0; j < numbiegen; j++)
    {
        i = umbiegen[j] ;
        if (i < 0) {
           i = - i - 1;
           myId = thesolver->coId(i) ;
           stat = ds.rowStatus( i ) ;
           switch( stat ) {
              case SPxBasis::Desc::P_ON_UPPER :
                 ds.rowStatus(i) = SPxBasis::Desc::P_ON_LOWER ;
                 thesolver->theFrhs->multAdd( lhs[i] - rhs[i], thesolver->vector(myId) ) ;
                 (*(thesolver->theCoUbound))[i] = (*(thesolver->theCoLbound))[i] ;
                 (*(thesolver->theCoLbound))[i] = - SPxLP::infinity ;
                 break;
              case SPxBasis::Desc::P_ON_LOWER :
                 ds.rowStatus(i) = SPxBasis::Desc::P_ON_UPPER ;
                 thesolver->theFrhs->multAdd( rhs[i] - lhs[i], thesolver->vector(myId) ) ;
                 (*(thesolver->theCoLbound))[i] = (*(thesolver->theCoUbound))[i] ;
                 (*(thesolver->theCoUbound))[i] = SPxLP::infinity ;
                 break;
              default :
                 basvar++ ;
```

```
                cout << "Falscher Variablenstatus [" << stat << "].\n" ;
                break;
           }
        }
        else {
           myId = thesolver->id(i) ;
           stat = ds.colStatus( i ) ;
           switch( stat ) {
             case SPxBasis::Desc::P_ON_UPPER :
                ds.colStatus(i) = SPxBasis::Desc::P_ON_LOWER ;
                thesolver->theFrhs->multAdd( ub[i] - lb[i], thesolver->vector(myId) ) ;
                (*(thesolver->theLbound))[i] = (*(thesolver->theUbound))[i] ;
                (*(thesolver->theUbound))[i] = SPxLP::infinity ;
                break;
             case SPxBasis::Desc::P_ON_LOWER :
                ds.colStatus(i) = SPxBasis::Desc::P_ON_UPPER ;
                thesolver->theFrhs->multAdd( lb[i] - ub[i], thesolver->vector(myId) ) ;
                (*(thesolver->theUbound))[i] = (*(thesolver->theLbound))[i] ;
                (*(thesolver->theLbound))[i] = - SPxLP::infinity ;
                break;
             default :
                cout << "Falscher Variablenstatus [" << stat << "].\n" ;
                basvar++ ;
                break;
           }
        }
    }

    numbiegen -= basvar ;
    if (numbiegen) {
        theupdate = 1 ; // Berechne Zul"assigkeitsvektor neu, Update waere falsch
        thecounter2 += numbiegen;
        thecounter1 ++ ;
        numbiegen = 0;
    }


}

// ----------------------------------------------------------------------------

SoPlex::Id SPxLongStepRT::longStep( int& nr, double& max,
    double& maxabs, int sense )
{

// Variablenschranken
    const double*  ub     = thesolver->upper();
    const double*  lb     = thesolver->lower() ;
// Ungleichungsschranken
    const double*  rhs     = thesolver->rhs() ;
    const double*  lhs     = thesolver->lhs() ;
// g
    const double* pvec   = thesolver->pVec() ;
    const double* pupd   = thesolver->pVec().delta().values() ;
    const IdxSet& pidx   = thesolver->pVec().idx() ;
    const double* lpb    = thesolver->lpBound() ;
    const double* upb    = thesolver->upBound() ;
// h
```

```
    const double* cvec   = thesolver->coPvec() ;
    const double* cupd   = thesolver->coPvec().delta().values() ;
    const IdxSet& cidx   = thesolver->coPvec().idx() ;
    const double* lcb    = thesolver->lcBound() ;
    const double* ucb    = thesolver->ucBound() ;

    int cnum, pnum ;
    double val = max;
    double max2 = max;
    double x ;
    SoPlex::Id enterId, myId ;
    SPxBasis::Desc& ds = thesolver->desc() ;

    int  i, j ;
    int  indp, indc ;

// Setup thetac und thetap
    if (sense == 1) {
        indc = maxDeltaLS( max2, maxabs,
           thesolver->coPvec(), thesolver->lcBound(), thesolver->ucBound(),
           0, 1, thetac ) ;
        indp = maxDeltaLS( max2, maxabs,
           thesolver->pVec(), thesolver->lpBound(), thesolver->upBound(),
           0, 1, thetap ) ;
    }
    else {
        indc = minDeltaLS( max2, maxabs,
           thesolver->coPvec(), thesolver->lcBound(), thesolver->ucBound(),
           0, 1, thetac ) ;
        indp = minDeltaLS( max2, maxabs,
           thesolver->pVec(), thesolver->lpBound(), thesolver->upBound(),
           0, 1, thetap ) ;
    }

    do {
        pnum = cnum = -1;
        val = SPxLP::infinity ;
  // Bestimme Minimum
        for (j=0 ; j <= pidx.size()-1 ; j++ )
            if (thetap[j] < val) {
                val = thetap[j];
                pnum = j;
            }
        for (j=0 ; j <= cidx.size()-1 ; j++ )
            if (thetac[j] < val) {
                val = thetac[j];
                cnum = j;
            }

        if (val >= SPxLP::infinity) break;       // INFEASIBLE

        if (cnum >= 0) {

            i = cidx.index(cnum);
            x = cupd[i] ;
            if (x < 0) x = -x ;

            if ( (rhs[i] >= SPxLP::infinity) || (lhs[i] <= - SPxLP::infinity) )
```

```
            alpha = SPxLP::infinity;
        else alpha += x * (rhs[i] - lhs[i]);

        thetac[cnum] = SPxLP::infinity ;

        if (alpha < - epsilon){
            umbiegen[numbiegen] = - (i+1) ;
            numbiegen++ ;
        }
    }
    else {
        i = pidx.index(pnum);
        x = pupd[i] ;
        if (x < 0) x = -x ;

        if ( (ub[i] >= SPxLP::infinity) || (lb[i] <= - SPxLP::infinity) )
            alpha = SPxLP::infinity;
        else alpha += x * (ub[i] - lb[i]);

        thetap[pnum] = SPxLP::infinity ;
        if (alpha < - epsilon){
            umbiegen[numbiegen] = i ;
            numbiegen++ ;
        }
    } // cnum oder pnum

} while (alpha < - epsilon) ;


          // Heuristik
if ( (val < delta * 500) || (x < 5 * minStab) && (maxabs > 100*x) ) {
   max = max2 ;
   numbiegen = 0;
   if( indp >= 0 )
   {
      nr = indp ;
      return thesolver->id(indp) ;
   }
     if( indc >= 0 )
   {
      nr = indc ;
      return thesolver->coId(indc) ;
   }

}

max = val * sense;

if( cnum >= 0 )
{
   nr = i ;
   return thesolver->coId(i) ;
}
if( pnum >= 0 )
{
   nr = i ;
   return thesolver->id(i) ;
}
```

```
    nr = -1 ;
    return enterId ;
}

// ----------------------------------------------------------------------------

SoPlex::Id SPxLongStepRT::selectEnterCol(double& val, int leaveIdx)
{
// x_j_q
    const double  vec    = (thesolver->fVec())[leaveIdx] ;
    const double fub    = (thesolver->ubBound())[leaveIdx] ;
    const double flb    = (thesolver->lbBound())[leaveIdx] ;

    SoPlex::Id enterId ;
    double max, sel ;
    double maxabs ;
    int nr ;
    int cnt = 0 ;

    resetTols() ;
    sel = 0 ;

    if( val > epsilon )
    {
        do
        {
            maxabs = 0 ;
            max = val ;

            alpha = vec - flb;
            enterId = longStep( nr, max, maxabs, 1 ) ;
            if( !enterId.isValid() )
                return enterId ;

            if( !shortEnter( enterId, nr, max, maxabs ) )
            {
                if (numbiegen) {
                    max = umbiegen[0] ;
                    numbiegen = 0 ;
                }

                double bestDelta, stab ;
                stab = minStability( minStab, maxabs ) ;
                enterId = maxSelect( nr, sel, stab, bestDelta, max ) ;
                if( bestDelta  <  DELTA_SHIFT*TRIES )
                   cnt++ ;
                else
                    cnt += TRIES ;
            }
            if( !maxReenter( sel, max, maxabs, enterId, nr ) )
                break ;
            relax() ;
        } while( cnt < TRIES ) ;
    }

    else if( val < -epsilon )
    {
```

```
        do
        {
            maxabs = 0 ;
            max = val ;
            enterId = longStep( nr, max, maxabs, - 1 ) ;
            if( !enterId.isValid() )
                return enterId ;
            assert( max <= 0 ) ;

            if( !shortEnter( enterId, nr, max, maxabs ) )
            {
                double bestDelta, stab ;
                stab = minStability( minStab, maxabs ) ;
                enterId = minSelect( nr, sel, stab, bestDelta, max ) ;
                if( bestDelta  <  DELTA_SHIFT*TRIES )
                    cnt++ ;
                else
                    cnt += TRIES ;
            }
            if( !minReenter( sel, max, maxabs, enterId, nr ) )
                break ;
            relax() ;
        } while( cnt < TRIES ) ;
    }

    if( enterId.isValid()  ||  minStab > 2*epsilon )
    {
        val = sel ;
        if( enterId.isValid() ) {
            tighten() ;
            umbasteln() ;
        }
    }
    return enterId ;
}

// ----------------------------------------------------------------------------

SoPlex::Id SPxLongStepRT::selectEnter(double& val, int leaveIdx=0) {

     theupdate = 0 ; // Fuehre normales Update aus
     if (thesolver->rep() == SoPlex::COLUMN)
         return selectEnterCol( val, leaveIdx ) ;
     else
         return selectEnterRow( val ) ;
}
```

Abbildungsverzeichnis

1.1 Hyperebene und Halbräume 9
1.2 Unbeschränktes Polyeder im $\mathbb{R}^2$ 9
1.3 Höhenlinien der Zielfunktion in zulässiger Menge 10
1.4 Dreidimensionales Polytop 11
1.5 Ecke eines Polytops (Zeilenbasis) 46

2.1 Statisches Klassendiagramm von SoPlex 59
2.2 Transformation der Variablen bei Basisupdates 63

Tabellenverzeichnis

1.1 Zusammenhang zwischen g und c 40
1.2 Zusammenhang zwischen Basisdarstellung und Algorithmustyp 48
1.3 Initialisierung der allgemeinen Basis 50

2.1 Zusammenhang zwischen Variablenstatus und Basen 61
2.2 Variablenstatus in SoPlex und Variablengrenzen 61
2.3 Variablen in SoPlex 64
2.4 Variablendimensionen in SoPlex 64

3.1 Übersicht Zulässigkeiten der Algorithmen 77

4.1 Beschreibung der netlib Programme mit beschränkten Variablen 91
4.2 netlib-Test aller ratio tests, Phase I: ENTER 92
4.3 netlib-Test aller ratio tests, Phase I: LEAVE 93
4.4 netlib-Tests für Verfahren 3.4, Vergleich der Auswirkungen von Updates auf die Laufzeit 95
4.5 netlib-Tests für Verfahren 3.5 96
4.6 netlib-Tests für Verfahren 3.6, Phase I = ENTER 97
4.7 netlib-Tests für Verfahren 3.6, Phase I = LEAVE. 98
4.8 Beschreibung und Numerierung der miplib Programme 99
4.9 miplib-Tests für Verfahren 3.6, Phase I = ENTER 100
4.10 miplib-Tests für Verfahren 3.6, Phase I = LEAVE 102

Algorithmenverzeichnis

1.1 Primaler Simplex für Spaltenbasis 16
1.2 Dualer Simplex für Spaltenbasis 22
1.3 Dualer Simplex für Spaltenbasis, gleichungsbeschränktes LP 43
1.4 Entfernender Algorithmus bei Allgemeiner Basis 49

3.1 Entfernender Algorithmus mit LongStep ratio test, Spaltenbasis 75
3.2 Einfügender Algorithmus mit LongStep ratio test, Zeilenbasis 79
3.3 LongStep ratio test ohne Stabilisierung 83
3.4 LongStep ratio test mit δ 84
3.5 Stabilisierter LongStep ratio test 84
3.6 Modifizierter stabilisierter LongStep ratio test 85

Literaturverzeichnis

[BG69] R. H. Bartels, G. H. Golub, "The simplex method for linear programming using LU decomposition", Communications of the ACM 12, (1969)

[Bix94] R. E. Bixby, "Progress in linear programming", ORSA J. Comp. 6, 1, (1994)

[BJN93] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, P. H. Vance, "Branch-and-price: Column generation for solving huge integer programs", Technical Report COC-9403, Georgia Institute of Technology, Atlanta, USA, (1993)

[BRJ99] Grady Booch, James Rumbaugh, Ivar Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, (1999)

[Chv83] Vasek Chvátal, "Linear Programming", Freeman and Company, New York, (1983)

[DER89] I. S. Duff, A. M. Erisman, J. K. Reid, "Direct methods for sparse matrices", Clarendon press, (1989)

[FT72] J. J. H. Forrest, J. A. Tomlin, "Updating triangular factors of the basis to maintain sparsity in the product form of the simplex method", Mathematical Programming 2, (1972)

[Gab93] R. Gabasov, "Adaptive method of linear programming", Preprints of the University of Karlsruhe, Institute of Statistics and Mathematics, Karlsruhe, (1993)

[GKK79] R. Gabasov, F. M. Kirillova, O. I. Kostyukova, "A method of solving general linear programming problems"(russisch), Doklady AN BSSR 23, N 3, (1979)

[GMS87] Philip E. Gill, Walter Murray, Michael A. Saunders, "Maintaining LU Factors of a General Sparse Matrix", Linear Algebra and its Applications, 88/89, (1989)

[GR77] D. Goldfarb, J. K. Reid, "A practicable steepest-edge simplex algorithm", Math. Prog. 12, (1977)

[Har73] P. M. J. Harris, "Pivot selection methods for the Devex LP code", Math. Prog. 5, (1973)

[Kos00] E. Kostina, "The long step rule in the bounded-variable dual simplex method: numerical experiments", Preprint 2000-28 (SFB 359), Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Heidelberg, (2000)

[MIP01] Robert E. Bixby et al., "Updated Mixed Integer Programming Library 3.0", http://www.caam.rice.edu/~bixby/miplib/miplib.html, (2001)

[Mey88] Bertrand Meyer, "Object-Oriented Software Construction", Prentice Hall, (1988)

[NET01] Jack Dongarra, Eric Grosse et al., "Netlib linear programming test problems", http://netlib.bell-labs.com/netlib/master/readme.html, (2001)

[NKT89] G. L. Nemhauser, A. H. G. Rinnooy Kan, M. J. Todd, "Handbooks in Operations Research and Management Science, Volume 1: Optimization", NORTH-HOLLAND, (1989)

[Pad99] M. Padberg, "Linear Optimization and Extensions", Springer Heidelberg, (1999)

[JRT95] M. Jünger, G. Reinelt, S. Thienel, "Practical Problem Solving with Cutting Plane Algorithms in Combinatorial Optimization", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol.20, 1995, 111-152

[Sop96] Roland Wunderling, "SoPlex Softwarepaket", http://www.zib.de/Optimization/index.en.html, (1996)

[Sto94] Josef Stoer, "Einführung in die Numerische Mathematik I", Springer Verlag, (1994)

[Suh89] Uwe H. Suhl, Leena M. Suhl, "Computing Sparse LU Factorizations for Large-Scale Linear Programming Bases", ORSA Journal on Computing, Vol.2, No.4, (1989)

[Tim97] Christian Timpe, "Zwei Varianten des Simplex-Verfahrens", Diplomarbeit an der Uni Heidelberg, (1997)

[Wol65] P. Wolfe, "The composite simplex algorithm", SIAM Review 7, 1, (1965)

[Wun96] Roland Wunderling, "Paralleler und objektorientierter Simplex-Algorithmus", Technical Report TR 96-9, Konrad-Zuse-Zentrum für Informationstechnik Berlin, (1996)

[Zus45] Konrad Zuse, "Theorie der angewandten Logistik - 2. Buch.", (1945)

[Zus99] Horst Zuse, "Geschichte der Programmiersprachen", Bericht 1999-1, TU Berlin, FB Informatik, (1999)