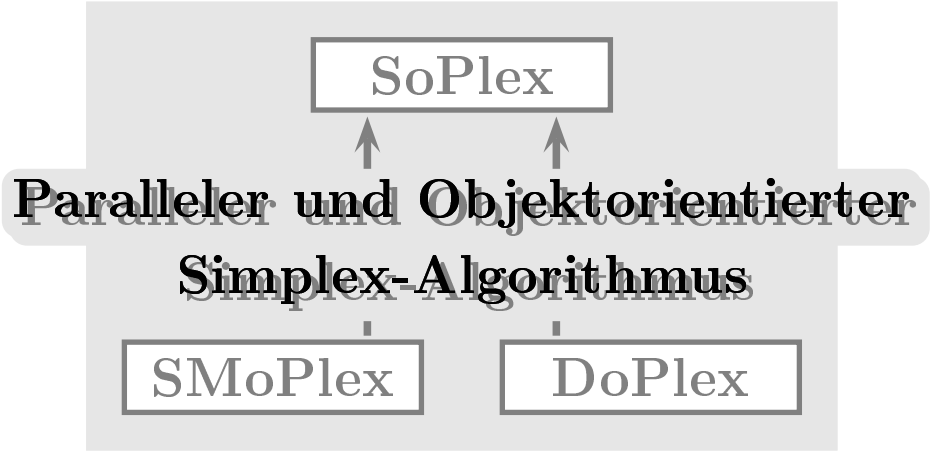
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7, D-14195 Berlin



Paralleler und Objektorientierter Simplex-Algorithmus

Roland Wunderling

Technical Report TR 96-9 (December 1996)

Paralleler und objektorientierter Simplex-Algorithmus

vorgelegt von
Dipl. Phys.
Roland Wunderling

Vom Fachbereich Mathematik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
— Dr. rer. nat. —
genehmigte Dissertation

Promotionsausschuß:

Vorsiztender:	Prof. Dr. Thiele
Berichter:	Prof. Dr. Martin Grötschel
Berichter:	Prof. Dr. Günter Ziegler

Berlin 1996

D83

meiner Frau
gewidmet

Vorwort

Der Anstoß für die vorliegende Arbeit kam aus einem Forschungsprojekt mit der Firma *Cray Research Inc.* (CRI) zur Entwicklung eines parallelen Branch-and-Cut Frameworks für die Lösung großer kombinatorischer Optimierungsprobleme auf dem Parallelrechner Cray T3D. In Ermangelung eines LP-Lösers für diese Hardware, begann ich am *Konrad Zuse-Institut für Informationstechnik Berlin* (ZIB) mit der Entwicklung eines eigenen Simplex-Codes, der die Basis dieser Forschungsarbeit darstellt.

Mein besonderer Dank gilt *Prof. Dr. Martin Grötschel*. Er führte mich in das Gebiet der mathematischen Optimierung und insbesondere der Linearen Programmierung ein, und er leitete mich bei dieser Forschungsarbeit an. Von *Prof. Dr. Robert E. Bixby* lernte ich viel über die Implementierung von Simplex-Algorithmen, und er stellte mir schwierige Problembeispiele zur Verfügung, die mir zu weiteren Einsichten und entscheidenden Verbesserungen des Codes verhalfen. Ihm sei dafür gedankt. Viel zum Gelingen dieser Arbeit hat auch mein Freund *Martin Grammel* beigetragen, dem ich an dieser Stelle meinen Dank aussprechen möchte. Er lehrte mich die Konzepte der objektorientierten Programmierung und des parallelen Rechnens und war ein wichtiger Diskussionspartner bei vielen Entwurfsentscheidungen. *Michael Ganss* danke ich für die Implementierung des parallelen Lösers für dünnbesetzte unsymmetrische lineare Gleichungssysteme. Bei meinen Kollegen *Dr. Dimitris Alevras*, *Dr. Norbert Ascheuer*, *Ralf Borndörfer*, *Andreas Eisenblätter*, *Nikola Kamin* und *Andreas Löbel* möchte ich mich für die Bereitstellung verschiedener Testprobleme bedanken.

Nicht zuletzt möchte ich meiner Frau und meinen Eltern dafür danken, daß während der gesamten Zeit mein Zuhause nicht nur eine Unterkunft, sondern ein Ort zum Tanken neuer Kraft war.

Berlin, Dezember 1996 Roland Wunderling

Abstract

Dipl. Phys. Wunderling, Roland: Paralleler und objektorientierter Simplex-Algorithmus

In der vorliegenden Arbeit werden neue Implementierungen des dualen und primalen revidierten Simplex-Algorithmus für die Lösung linearer Programme (LPs) vorgestellt. Dazu werden die Algorithmen mithilfe einer Zeilenbasis dargestellt, aus der über einen Spezialfall die übliche Darstellung mit einer Spaltenbasis folgt. Beide Darstellungen sind über die Dualität eng miteinander verbunden. Außerdem wird eine theoretische Untersuchung der numerischen Stabilität von Simplex-Algorithmen durchgeführt, und es werden verschiedene Möglichkeiten der Stabilisierung diskutiert.

Beide Darstellungen der Basis werden in den Implementierungen algorithmisch ausgenutzt, wobei der Einsatz der Zeilenbasis für LPs mit mehr Nebenbedingungen als Variablen Geschwindigkeitsvorteile bringt. Darüberhinaus werden weitere Beschleunigung gegenüber anderen state-of-the-art Implentierungen erzielt, und zwar durch den Einsatz eines Phase-1 LPs, das eine größtmögliche Übereinstimmung mit dem Ausgangs-LPs aufweist, durch eine dynamische Anpassung der Faktorisierungsfrequenz für die Basis-Matrix und durch die Optimierung der Lösung linearer Gleichungssysteme für besonders dünnbesetzte Matrizen und Vektoren.

Es wurden drei Implementierungen vorgenommen. Die erste läuft sequentiell auf einem PC oder einer Workstation. Ihre hohe numerische Stabilität und Effizienz durch die Integration der oben genannten Konzepte machen sie zu einem zuverlässigen Hilfsmittel für den täglichen Einsatz z.B. in Schnittebenenverfahren zur Lösung ganzzahliger Programme. Als Programmiersprache wurde C++ verwendet, und es wurde ein objektorientierter Software-Entwurf zugrundegelegt. Dieser leistet eine hohe Flexibilität und Anpaßbarkeit z.B. für die Integration benutzerdefinierter Pricing-Strategien.

Bei den anderen beiden Implentierungen handelt es sich um parallele Versionen für Parallelrechner mit gemeinsamem und für solche mit verteiltem Speicher. Dabei wird der objektorientierte Entwurf so genutzt, daß lediglich die zusätzlichen Aufgaben für die Parallelisierung (Synchronisation, Kommunikation und Verteilung der Arbeit) implementiert werden, während alle Algorithmen von der sequentiellen Implementierung geerbt werden.

Die Parallelisierung setzt an vier Punkten an. Der erste und einfachste ist die parallele Berechnung eines Matrix-Vektor-Produktes. Als zweites wurden beim Pricing und Quotiententest parallele Suchalgorithmen eingesetzt. Weiter werden beim steepest-edge Pricing zwei lineare Gleichungssysteme nebenläufig gelöst. Schließlich wird ein paralleles Block-Pivoting verwendet, bei dem Gleichungssysteme mehrerer aufeinanderfolgender Iterationen gleichzeitig gelöst werden. Ob und welche der Parallelisierungs-Konzepte eine Beschleunigung bewirken, ist problemabhängig. Es gelingt z.B. mit 32 Prozessoren eine Beschleunigung um mehr als einen Faktor 16 zu erzielen.

Schließlich wird die parallele Lösung dünnbesetzter linearer Gleichungssysteme mit unsymmetrischen Matrizen untersucht und eine Implementierung für den Cray T3D vorgenommen. Sie enthält ein neues Verfahren des Lastausgleichs, das keinen zusätzlichen Aufwand verursacht. Die Implementierung erzielt vergleichsweise günstige Laufzeiten.

Inhaltsverzeichnis

Einleitung

Der Anfang der Linearen Programmierung als mathematische Disziplin läßt sich genau auf das Jahr 1947 datieren, in dem G.B. Dantzig das erste Lineare Programm (LP) aufstellte und den Simplex-Algorithmus als Lösungsverfahren formulierte. Zu dieser Zeit verwendete man den Begriff Programm noch nicht in seiner heutigen Bedeutung als Ausführungscode für Computer. Vielmehr verstanden die Militärs unter einem Programm einen Plan zur Ressourcenverteilung; das erste von G.B. Dantzig aufgestellte Lineare Programm war eben ein militärisches Planungsproblem.

Bereits vor 1947 gab es vereinzelte Arbeiten, die heute der Linearen Programmierung zugeordnet werden müssen. Sie blieben jedoch ohne Auswirkung auf die Fortentwicklung der Mathematik, wie die Arbeiten von Fourier 1823 und de la Vallée Poussin 1911, oder wurden aus politisch-ideologischen Gründen nicht vorangetrieben (Kantorovich 1939) [32].

Lineare Programme sind Optimierungsprobleme, die in die Form

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

gebracht werden können. Dabei ist A eine Matrix und c, b und x sind dimensionskompatible Vektoren. Schon bald stellte sich heraus, daß Lineare Programme weitreichende Anwendungen in zahlreichen wirschaftlichen und gesellschaftlichen Gebieten haben, wie bei der Transport- und Netzwerkplanung, der Ressourcenverteilung oder beim Scheduling, um nur einige zu nennen. 1949 faßte R. Dorfman diese Gebiete mit dem Begriff der *mathematischen Programmierung* zusammen [32]. Ein zentraler Teil davon ist die Lineare Programmierung, deren Gegenstand die algorithmische Lösung von LPs ist. Solche treten oft bei der Lösung mathematischer Programme auf. Dabei dient der (im Laufe der Zeit weiterentwickelte) Simplex-Algorithums bis heute als „Arbeitspferd“. Die heutige Bedeutung von LP-Lösern zeigt sich z.B. darin, daß 1995 mindestens 29 verschiedene Implementierungen kommerziell angeboten wurden [88].

Obwohl in vielzähligen Anwendungen bewährt, ist der Simplex-Algorithmus besonders für Mathematiker nicht zufriedenstellend. Dies liegt daran, daß seine Laufzeit nicht befriedigend beschränkt werden kann. Bereits in den 50er Jahren wurde von A.J. Hoffman ein LP angegeben, für das der damalige Simplex-Algorithmus nicht terminiert [65]. Dieses

Problem des sog. *Kreiselns* konnte zwar durch spezielle Varianten umgangen werden. Für die meisten solcher Varianten gibt es aber Problembeispiele, die ein nicht polynomiales Laufzeitverhalten aufweisen. Alle bekannten Beispiele werden in [2] einheitlich zusammengefaßt. Bei vielen praktischen Problemen zeigt sich hingegen eine polynomiale Laufzeit, was auch durch statistische Analysen untermauert werden kann [21]. Ob es jedoch polynomiale Simplex-Algorithmen gibt, ist eine bis heute offene Frage [70, 104].

Deshalb wurde lange der Frage nachgegangen, ob Lineare Programme überhaupt in polynomialer Zeit gelöst werden können. Diese Frage wurde 1979 von L.G. Khachian durch Angabe der Ellipsoid-Methode positiv beantwortet [69, 54]. Trotz ihrer theoretisch überlegenen Laufzeit konnte die Ellipsoid-Methode oder deren Varianten in realen Anwendungen nicht mit dem Simplex-Algorithmus konkurieren, und so beschränkte sich die Fortentwicklung der Linearen Programmierung noch weitere fünf Jahre i.w. auf Verbesserungen des Simplex-Verfahrens.

Vielleicht durch die Ellipsoid-Methode angespornt, stellte N.K. Karmarkar im Jahr 1984 mit einem Innere-Punkte-Verfahren einen anderen polynomialen LP-Löser vor [67]. Innere-Punkte-Verfahren wurden bereits seit den 60er Jahren zur Lösung nichtlinearer Optimierungsprobleme eingesetzt [44], jedoch wurden sie erst seit 1984 für den linearen Fall spezialisiert und seither zu einer mächtigen Konkurrenz zu Simplex-Algorithmen fortentwickelt [76, 87]. Derzeit sind die Innere-Punkte-Verfahren und der Simplex-Algorithmus gleichermaßen etabliert, und für viele Anwendungen ist eine Kombination beider Methoden mittels eines sog. *Cross-overs* die beste Wahl [14, 16].

Unabhängig von den Erfolgen der Innere-Punkte-Verfahren kommt dem Simplex-Algorithmus auf absehbare Zeit eine nicht zu vernachlässigende Bedeutung zu. So liefert er zusätzlich zu einer optimalen Lösung des LPs weitere Informationen wie die Schattenpreise, die, ökonomisch interpretiert, eine bessere Beurteilung der Lösung erlauben. Darüberhinaus werden Simplex-Algorithmen in Schnittebenen-Verfahren zur Lösung ganzzahliger Optimierungsprobleme eingesetzt. Dabei wird eine Folge von jeweils leicht modifizierten LPs generiert und gelöst. Nur der Simplex-Algorithmus ist hierbei in der Lage, auf dem Ergebnis des vorigen LPs aufzusetzen, um das jeweils neue LP schneller zu lösen. Deshalb bleibt die Weiterentwicklung des Simplex-Algorithmus' ein ständig aktuelles Forschungsthema.

Die Verbesserungen des Simplex-Algorithmus' haben seit seiner Erfindung Beachtliches zustandegebracht. 1953 feierte man die Lösung eines 48×72 LPs noch mit Wein und Schmaus [65]. Heute werden auch LPs mit über 100000 Zeilen und Spalten routinemäßig gelöst, und die Entwicklung immer ausgeklügelterer Verfahren, mit denen immer größere LPs im immer kürzerer Zeit gelöst werden können, hält weiter an. Auch die vorliegende Arbeit trägt dazu bei.

G.B. Dantzig formulierte den Simplex-Algorithmus mit dem sog. *Simplex-Tableau*. Dies ist im wesentlichen eine Matrix, die in jeder Iteration des Algorithmus' vollständig aktualisiert wird. Ein bedeutender Meilenstein bei der Entwicklung des Simplex-Algorithmus' bestand in der Formulierung des *revidierten* Simplex-Algorithmus', bei dem immer nur ein

Teil des Tableaus, die sog. Basismatrix, aktualisiert wird. Dafür müssen pro Iteration zwei Gleichungssysteme mit der Basismatrix gelöst werden. Dennoch kann gezeigt werden, daß der Gesamtaufwand geringer ausfällt.

Ein weiterer wichtiger Vorteil des revidierten Simplex-Algorithmus' gegenüber der Tableauform ist, daß er es erleichtert, Datenstrukturen für dünnbesetzte Matrizen zu verwenden, also solche mit vielen Null-Elementen. Typischerweise haben die Nebenbedingungsmatrizen A heutiger LPs nur 1-20 Nicht-Null-Elemente pro Spalte oder Zeile, ggf. mit einigen Ausnahmen. Insgesamt sind weit weniger als 10% der Elemente von Null verschieden. Bei der Aktualisierung der Tableaus ohne Ausnutzung der Dünnbesetztheit würden viele Rechenoperationen mit Null-Elementen durchgeführt, ein Aufwand, der vermieden werden sollte.

Eine weitere Verbesserung bestand in der Einführung der LU-Zerlegung der Basismatrix für die Lösung der beiden Linearen Gleichungssysteme in jeder Iteration des revidierten Simplex-Algorithmus' [77]. Ursprünglich verwandte man die sog. *Produktform der Inversen* [29]. Für dünnbesetzte Matrizen wurde jedoch gezeigt, daß diese zu mehr Nicht-Null-Elementen führt als die LU-Zerlegung und somit einen höheren Rechenaufwand bedingt [11].

Bereits 1954 wurde von C.E. Lemke der duale Simplex-Algorithmus angegeben. Heute ist er fester Bestandteil aller effizienten Implementierungen, nicht zuletzt weil er eine wesentliche Grundlage für die Implementierung von Schnittebenenverfahren darstellt.

Eine wichtige Verbesserung des Simplex-Algorithmus' war seine Erweiterung auf LPs, bei denen die Variablen nicht nur untere, sondern auch obere Schranken aufweisen können. Außerdem werden für die Variablenschranken auch von Null verschiedene Werte zugelassen. Das Verfahren wurde 1955 von G.B. Dantzig selbst aufgestellt und ist als *upper-bounding Technik* bekannt.

Das Pricing ist ein wichtiger Schritt beim Simplex-Verfahren, der einige Freiheit zuläßt. Daher verwundert es nicht, daß sich eine Vielzahl von Verbesserungsvorschlägen um eine Konkretisierung bemühen, die die Iterationszahl des Algorithmus' reduzieren oder seine Termination sicherstellen sollen. Bei der ersten Kategorie sind insbesondere die Arbeiten von P.M.J. Harris [60] sowie von D. Goldfarb und J.K. Reid [52] bzw. J.J. Forest und D. Goldfarb [47] zu nennen. Sie befassen sich mit Varianten des sog. steepest-edge Pricings, für das schon lange bekannt war, daß es zwar zu einer bemerkenswert geringen Iterationszahl führt, dafür aber eines hohen Rechenaufwandes bedarf [71]. Harris stellt ein approximatives Verfahren vor, das wesentlich weniger Operationen bedarf. Goldfarb und Reid entwickeln für den primalen Algorithmus Update-Formeln, die den Rechenaufwand beim steepest-edge Pricing mindern. In [47] wurden diese auch für den dualen Algorithmus aufgestellt.

Simplex-Varianten, die eine Termination des Simplex-Algorithmus' zusichern, benutzen meist eine Kombination von Verfahren für den Pricing-Schritt und den sog. Quotiententest. Beides wird mit dem Begriff *Pivot-Regel* zusammengefaßt. Erste Ansätze zur garantierten

Termination des Simplex-Algorithmus' waren die Perturbationsmethode von Orden und Charnes [24] sowie die Lexikographische Methode von Dantzig, Orden und Wolfe [30]. Mit der Arbeit von Bland [17] begann eine verstärkte Untersuchung von Pivotregeln, die eine endliche Laufzeit des Simplex-Algorithmus' garantieren. Eine Übersicht findet sich in [94]. Insgesamt wurde noch kein theoretischer Ansatz gefunden, der auch in der Praxis befriedigt. Deshalb werden in heutigen Implementierungen Verfahren eingesetzt, die sich in der Praxis bewährt haben, aber auf keiner theoretischen Grundlage basieren [50, 16].

Der Quotiententest ist auch der Schritt bei Simplex-Algorithmen, der die numerische Stabilität des Verfahrens bestimmt. Wenn auch ohne theoretische Stabilitätsanalyse, wurde der wesentliche Ansatzpunkt zur Gewährleistung der numerischen Stabilität in [60] aufgezeigt. Er ist bis heute die Grundlage für robuste Implementierungen [50, 16].

Ein wichtiges Arbeitsfeld bei der Fortentwicklung von Simplex-Algorithmen ist deren Spezialisierung auf LPs mit ausgezeichneter Struktur. Wichtigstes Beispiel sind Netzwerkprobleme, für die der sog. Netzwerk-Simplex entwickelt wurde, der von den Datenstrukturen her kaum mehr an Dantzig's Algorithmus erinnert [26]. Ein anderer Spezialfall sind Multi-Commodity-Flow Probleme. Bei diesen Problemen zerfällt die Nebenbedingungsmatrix in Diagonalblöcke, die durch wenige Zeilen verbunden sind. Hierfür wurde von Dantzig und Wolfe eine Dekompositionsmethode entwickelt, bei der iterativ zu jedem Diagonalblock ein LP gelöst wird, um aus diesen Teillösungen eine verbesserte Lösung des Gesamt-LPs zu konstruieren [31]. Gerade in der Zeit des Aufkommens von Parallelrechnern, bei denen die Teilprobleme von verschiedenen Prozessoren berechnet werden können, wurde dies wieder aufgegriffen [74].

Es gibt noch eine Vielzahl weiterer Arbeiten mit Modifikations- oder Verbesserungsvorschlägen zum Simplex-Algorithmus. Den meisten dieser Arbeiten ist jedoch gemein, daß die evtl. angegebenen Implementierungen nicht alle oben genannten Verbesserungen enthalten, sondern oft nur auf dem Grundalgorithmus von Dantzig basieren oder sich mit diesem vergleichen (siehe z.B. [25, 37, 82, 4]). Deshalb bleiben diese Ansätze weitgehend für die Lösung praktischer Probleme irrelevant.

Um zur Fortentwicklung von Simplex-Algorithmen beizutragen, muß zunächst auf dem bisher geschilderten Stand aufgesetzt werden. Dies leisten die in der vorliegenden Arbeit entwickelten Implementierungen, und sie treiben die Fortentwicklung von Simplex-Algorithmen auf verschieden Ebenen voran:

- Auf konzeptioneller Ebene basieren die Implementierungen auf einer neuen Darstellung des primalen und dualen Simplex-Algorithmus mithilfe einer Zeilenbasis, aus der die übliche Darstellung mit einer Spaltenbasis als Spezialfall folgt. Die gewählte Darstellung ermöglicht eine einheitliche Formulierung beider Algorithmen für beide Arten der Basis und wurde den Implementierungen zugrundegelegt. Desweiteren wird eine theoretische Stabilitätsanalyse des Simplex-Verfahrens durchgeführt, und es werden Methoden der Stabilisierung aufgezeigt. Außerdem verwenden die Implementierungen ein Phase 1 LP, das eine größtmögliche Übereinstimmung mit dem zu lösenden LP aufweist.

- Auf Implementierungsebene kommt ein verbesserter Lösungsalgorithmus für lineare Gleichungssysteme mit extrem dünnbesetzten Matrizen und Vektoren zum Einsatz. Außerdem wird der Zeitpunkt der erneuten Faktorisierung der Basismatrix dynamisch so festgelegt, daß die Iterationsgeschwindigkeit maximiert wird.

- In software-technischer Hinsicht wird der Simplex-Algorithmus einem objektorientierten Entwurf unterzogen. Dieser bietet eine hohe Flexibilität und Anpaßbarkeit, durch die es z.B. möglich ist, benutzerdefinierte Pricing- oder Quotiententest-Verfahren einzusetzen.

- Als Hardware kommen auch moderne Parallelrechner zum Einsatz. Dies fügt sich in die Tradition der Entwicklung von Simplex-Algorithmen, die stets eng mit den jeweiligen Hardwarevoraussetzungen verbunden war. So entwickelte Dantzig den Simplex-Algorithmus nur wegen der in Aussicht stehenden Verfügbarkeit automatischer Rechenmaschinen [32].
  Die hier untersuchte Parallelisierung setzen an vier Punkten an: Am einfachsten gestaltet sich die Parallelisierung elementarer Operationen der linearen Algebra auf dünnbesetzten Daten, wie die Berechnung des Matrix-Vektor-Produktes oder die Vektorsumme. Dies wurde bereits in anderen Arbeiten erfolgreich durchgeführt [13]. Außerdem kommen in dem Pricing- und Quotiententest-Schritt parallele Suchalgorithmen zum Einsatz. Ferner werden mit dem *Block-Pivoting* mehrere sequentielle Iterationen zu einer parallelen zusammengefaßt. Schließlich werden Möglichkeiten zur parallelen Lösung von Gleichungssystemen betrachtet. Dabei wird sowohl die gleichzeitige Lösung verschiedener Systeme sowie die Parallelisierung eines auf der LU-Zerlegung basierenden Lösers vorgenommen.

Es werden drei Implementierungen von revidierten Simplex-Algorithmen für verschiedene Hardware-Architekturen vorgestellt. Sie heißen:

SoPlex für „**S**equential **o**bject-oriented sim**Plex**“,

DoPlex für „**D**istributed **o**bject-oriented sim**Plex**“ und

SMoPlex für „**S**hared **M**emory **o**bject-oriented sim**Plex**“

SoPlex ist also eine Implementierung für normale PCs oder Workstations mit einem Prozessor. Sie enthält alle Konzepte von state-of-the-art Simplex-Algorithmen inclusive der in dieser Arbeit geleisteten Fortentwicklungen. Wo diese greifen, können Geschwindigkeitsgewinne gegenüber anderen Implementierungen wie CPLEX [27] erzielt werden. Damit eignet sich SoPlex für die Benutzung im täglichen Einsatz.

Der objektorientierte Entwurf ermöglicht es, alle Eigenschaften von SoPlex den Implementierungen DoPlex für Parallelrechner mit verteiltem Speicher und SMoPlex für solche mit gemeinsamen Speicher zu vererben, insbesondere seine numerische Stabilität. Für beide parallele Versionen wurden alle o.g. Parallelisierungsansätze implementiert.

Die Arbeit ist in vier Kapitel gegliedert. Das erste Kapitel enthält eine mathematische Darstellung der SoPlex zugrundeliegenden Algorithmen. Nach einer kurzen Einfühung der Notation und elementaren Sätzen (Abschnitt 1.1) werden zunächst der primale und der duale Grund-Algorithmus für eine Zeilenbasis anhand einfacher LPs aufgestellt (Abschnitt 1.2.1). Anhand eines Spezialfalles werden daraus in Abschnitt 1.2.2 die entsprechenden Algorithmen für die in der Literatur übliche Spaltenbasis abgeleitet. Der Zusammenhang zwischen beiden Darstellungen der Basis wird in Abschnitt 1.2.3 hergestellt. Anschließend werden die Simplex-Algorithmen für allgemeinere LPs für beide Basisdarstellungen beschrieben (Abschnitt 1.2.4). Abschnitt 1.3 enthält eine Stabilitätsanalyse der Simplex-Algorithmen und stellt verschiedene stabilisierte Quotiententest-Methoden vor. Es folgt eine Beschreibung der Phasen der Simplex-Algorithmen, der Kreiselvermeidung-Strategien sowie der wichtigsten Pricing-Verfahren (Abschnitte 1.4 bis 1.6). In Abschnitt 1.7 werden verschiedene Verfahren zur Lösung von Gleichungssystemen vorgestellt und der für SoPlex implementierte LU-Zerlegungs-Algorithmus beschrieben. Den Abschluß dieses Kapitels bildet Abschnitt 1.8 mit einer Zusammenstellung verschiedener Tricks, mit denen die Geschwindigkeit gesteigert werden kann.

Das zweite Kapitel befaßt sich mit den Parallelisierungsansätzen, die DoPlex und SMoPlex zugrundeliegen. Zunächst wird in Abschnitt 2.1 eine kurze Einführung in das Gebiet des parallelen Rechnens gegeben. Anschließend werden die möglichen Parallelisierungsansätze beim Simplex-Algorithmus aufgezeigt (Abschnitt 2.2). Einer davon ist die parallele Lösung linearer Gleichungssysteme. Wegen der Bedeutung weit über die Grenzen der mathematischen Programmierung hinaus wird dies in einem eigenen Abschnitt (2.3) behandelt.

Der objektorientierte Software-Entwurf und die Implementierung in C++ werden im dritten Kapitel behandelt. Zunächst wird in Abschnitt 3.1 eine Einführung in die objektorientierte Programmierung und eine Gegenüberstellung mit der imperativen Programmierung gegeben. In Abschnitt 3.2 werden die Klassen, mit denen SoPlex aufgebaut ist, und deren Zusammenspiel beschrieben. Die Klassen für die parallelen Versionen DoPlex und SMoPlex werden in Abschnitt 3.3 vorgestellt.

Im letzten Kapitel werden die mit den Implementierungen anhand verschiedener Beispielprobleme erzielten Testergebnisse ausgewertet. Zunächst wird die Menge der Test-LPs vorgestellt. In Abschnitt 4.2 werden die Lösung der Test-LPs mit SoPlex diskutiert, wobei der Einfluß der verschiedenen Methoden und Parameter aufgezeigt wird. Dabei wird auch ein Vergleich mit den von CPLEX 4.07 (aktuelle Version von Dez. 1996) erzielten Ergebnissen vorgenommen. Die Auswertung der parallelen Versionen SMoPlex und DoPlex erfolgt in den Abschnitten 4.3 und 4.4. Eine Zusammenfassung bildet den Abschluß der Arbeit.

Kapitel 1

Revidierte Simplex-Algorithmen

In diesem Kapitel werden Simplex-Algorithmen, genauer revidierte Simplex-Algorithmen, von der Theorie bis hin zur praktischen Umsetzung beschrieben. Dabei werden sowohl primale als auch duale Algorithmen jeweils für zwei verschiedene Darstellungen der Basis behandelt.

Zunächst wird in Abschnitt 1.1 die verwendete Notation eingeführt, und es werden die für Simplex-Algorithmen grundlegenden Sätze der Polyedertheorie zusammengestellt. Darauf aufbauend werden in Abschnitt 1.2 Simplex-Algorithmen mit verschiedenen Darstellungen der Basis entwickelt. Um zu lauffähigen Implementierungen zu gelangen, müssen noch weitere Aspekte berücksichtigt werden. Zunächst ist dies die numerische Stabilität, die in Abschnitt 1.3 analysiert wird. Dort werden auch verschiedene Ansätze vorgestellt, um Simplex-Algorithmen stabil zu machen. Weiterhin muß ein geeigneter Start für Simplex-Algorithmen gefunden werden. Dies geschieht mit der Phase 1, die in Abschnitt 1.4 diskutiert wird. Schließlich bergen Simplex-Algorithmen das Problem, daß sie evtl. nicht terminieren. Strategien dagegen werden in Abschnitt 1.5 vorgestellt. Beim sog. Pricing bieten Simplex-Algorithmen eine große algorithmische Vielfalt; die wichtigsten Strategien dafür werden in Abschnitt 1.6 vorgestellt. Auch die numerische Lösung linearer Gleichungssysteme ist ein integraler Bestandteil jeder Implementierung von Simplex-Algorithmen. Da diesem Problem eine Bedeutung weit über Simplex-Algorithmen hinaus zukommt, wird es in einen eigenen Abschitt, nämlich 1.7, behandelt. Schließlich werden in Abschnitt 1.8 verschieden Tips und Tricks für eine effiziente und zuverlässige Implementierung beschrieben.

1.1 Notation und mathematische Grundlagen

In diesem Abschnitt werden wesentliche Begriffe und Sätze der Polyedertheorie zusammengestellt, und es wird die Notation dazu eingeführt. Dies geschieht in enger Anlehnung an [55], worauf auch für die Beweise der hier nur zitierten Sätze verwiesen wird. Zuvor sei auf folgende Schreibweisen hingewiesen:

- Die Einheitsmatrix wird mit I bezeichnet. Ihre Dimension ist im allgemeinen aus dem Zusammenhang ersichtlich; andernfalls wird sie durch einen Index angegeben. Somit bezeichnet I_n die n-dimensionale Einheitsmatrix.

- Für Vektoren, die nur aus den Werten 1 oder nur aus den Werten 0 bestehen, wird kurz 1 bzw. 0 geschrieben, wobei die Dimension wiederum aus dem Zusammenhang ersichtlich ist.

- Ferner bezeichnet e_i den i-ten Einheitsvektor, wiederum in der aus dem Zusammenhang vorgegebenen Dimension.

Lineare Programme sind Optimierungsprobleme der Form

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Dx & \geq \quad d, \end{array} \tag{1.1}$$

wobei $c, x \in \mathbb{R}^n$, $d \in \mathbb{R}^k$ und $D \in \mathbb{R}^{k \times n}$ seien. Falls es für jedes $M \in \mathbb{R}$ ein $x \in \mathbb{R}^n$ gibt, so daß $c^T x < M$ und $Dx \geq d$ gilt, so heißt das LP *unbeschränkt*. Ist hingegen $\{x \in \mathbb{R}^n : Dx \geq d\} = \emptyset$, so heißt es *unzulässig*.

$D_{i.}$ bezeichne den i-ten Zeilenvektor der Matrix D, $D_{.j}$ entsprechend ihren j-ten Spaltenvektor. Jede Nebenbedingung $D_{i.}x \geq d_i$, mit $D_{i.} \neq 0$ definiert einen *Halbraum* $\mathcal{H}_i = \{x \in \mathbb{R}^n : D_{i.}x \geq d_i\}$ des $\mathbb{R}^n$, und die Hyperebene $G_i = \{x \in \mathbb{R}^n : D_{i.}x = d_i\}$ wird die *zugehörige Hyperebene* genannt. Lösungsvektoren x von (1.1) müssen alle Nebenbedingungen erfüllen und liegen somit im Schnitt aller Halbräume $\mathcal{H}_i$, $i = 1, \ldots, k$. Der Schnitt von endlich vielen Halbräumen ist ein *Polyeder* und wird mit

$$\mathcal{P}(D, d) = \bigcap_{i=1}^{k} \mathcal{H}_i = \{x \in \mathbb{R}^n : Dx \geq d\} \tag{1.2}$$

bezeichnet. Das Polyeder $\mathcal{P}(D, d)$ heißt das zum LP (1.1) gehörende Polyeder. Ferner werden später auch Polyeder $\mathcal{P}^{=}(D, d) = \{x \in \mathbb{R}^n : Dx = d, x \geq 0\}$ und $\mathcal{P}(l, D, u) = \{x \in \mathbb{R}^n : l \leq Dx \leq u\}$ benötigt.

Eine wichtige Eigenschaft von Polyedern ist, daß sie nicht nur als Schnitt von Halbräumen dargestellt werden können. Vielmehr kann jedes Polyeder aus zwei Grundtypen von Polyedern aufgebaut werden. Dies sind für $X = \{x_1, \ldots x_m\}$

- $\text{cone}\,(X) = \{x \in \mathbb{R}^n : \sum_{i=1}^{m} \lambda_i x_i, \lambda \geq 0\}$ und
- $\text{conv}\,(X) = \{x \in \mathbb{R}^n : \sum_{i=1}^{m} \lambda_i x_i, \lambda \geq 0, 1^T \lambda = 1\}$,

wobei $x_1, \ldots, x_m$ Vektoren und $\lambda_1, \ldots, \lambda_m$ Skalare bezeichnen. $\text{cone}\,(X)$ heißt der von den Vektoren $x_i, \ldots, x_m$ aufgespannte *Kegel* und $\text{conv}\,(X)$ ihre *konvexe Hülle*. Beide Mengen sind Polyeder. Für Polyeder P_1 und P_2 bezeichnet $P_1 + P_2 = \{x = x_1 + x_2 : x_1 \in P_1, x_2 \in P_2\}$ ihre *Minkowski-Summe*.

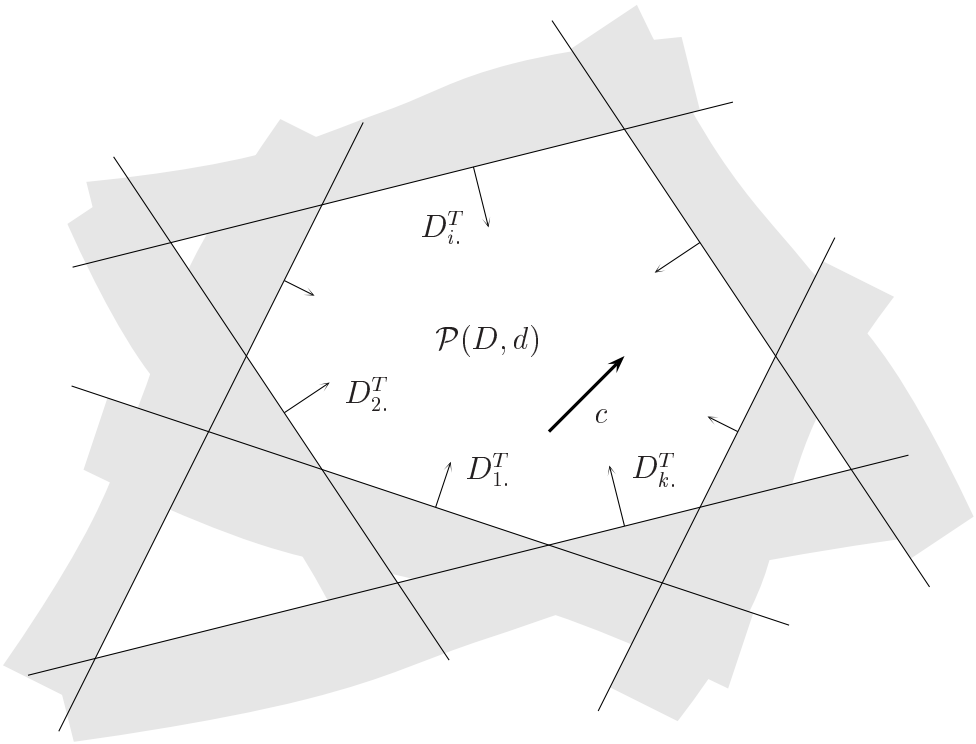


Abbildung 1.1: Geometrische Darstellung eines LPs im $\mathbb{R}^2$. Jede Ungleichung $D_{i.}x \geq d_i$, mit $D_{i.} \neq 0$ definiert einen Halbraum, wobei der Vektor $D_{i.}$ senkrecht zur Grenzebenen steht. Der unzulässige Halbraum ist jeweils grau dargestellt. Der Schnitt aller Halbräume bildet das Polyeder $\mathcal{P}(D,d)$. Es gilt den Punkt $x \in \mathcal{P}(D,d)$ zu finden, der am wenigsten weit in der durch den Zielfunktionsvektor c gegebenen Richtung liegt.

Satz 1 (Darstellungssatz)
Eine Teilmenge $P \subseteq \mathbb{R}^n$ ist genau dann ein Polyeder, wenn es endliche Mengen von Vektoren $V, E \subseteq \mathbb{R}^n$ gibt, so daß

$$P = \text{conv}\,(V) + \text{cone}\,(E). \tag{1.3}$$

Sofern P Ecken hat, ist dabei conv (V) die konvexe Hülle aller Ecken von P. Zur nachfolgenden Definition von Ecken eignet sich die Notation von *Indexvektoren* $I = (i_1, \ldots, i_m)$, wobei die Indizes $i_1, \ldots, i_m$ paarweise verschieden sein müssen. Zur Vereinfachung der Notation werden Indexvektoren auch als Mengen von Indizes aufgefaßt, und es wird die übliche Mengennotation verwendet. Zu einer Matrix $D^{m \times n}$ und Indexvektoren $I \subseteq \{1, \ldots, m\}$, $J \subseteq \{1, \ldots, n\}$ bezeichnet D_{IJ} die Untermatrix

$$D_{IJ} = \begin{pmatrix} D_{i_1 j_1} & \ldots & D_{i_1 j_{|J|}} \\ \vdots & & \vdots \\ D_{i_{|I|} j_1} & \ldots & D_{i_{|I|} j_{|J|}} \end{pmatrix}.$$

Falls $I = \{1, \ldots, m\}$ oder $J = \{1, \ldots, n\}$ verwendet man abkürzend auch '.' als Index.

DEFINITION 1 (ECKEN VON POLYEDERN)
Ein Vektor $x \in \mathcal{P}(D, d)$ heißt Ecke von $\mathcal{P}(D, d)$, *falls es einen Indexvektor $I \subseteq \{1, \ldots, m\}$ mit $|I| = n$ gibt, so daß $D_{I.}$ regulär und $D_{I.}x = d_I$ ist.*

Ein Polyeder $\mathcal{P}(D, d)$ mit $\operatorname{rang}(D) = n$ ist somit entweder leer, oder es ist *spitz*, d.h. es hat eine Ecke.

Für spitze Polyeder lassen sich die Vektoren E von (1.3) genauer beschreiben. Es sind Elemente des *Rezessionskegels* $\operatorname{rec}(\mathcal{P}(D, d)) = \mathcal{P}(D, 0) = \operatorname{cone}(E)$. E wird eine *Kegelbasis* genannt, falls für jede echten Teilmenge $E' \subsetneq E$ gilt: $\operatorname{cone}(E') \neq \operatorname{cone}(E)$. Die Elemente einer Kegelbasis heißen *Extremalen*. Sie sind bis auf Skalierung eindeutig bestimmt.

SATZ 2
Sei w Extremale eines spitzen Polyeders $\mathcal{P}(D, 0)$. Dann gibt es ein $\lambda > 0$ und einen Indexvektor I_w, d.d.

$$D_{I_w.}(\lambda w) = e_1.$$

Insbesondere ist auch $\lambda w = D_{I_w.}^{-1} e_1$ selbst Extremale von $\mathcal{P}(D, d)$. Statt des Einheitsvektors e_1 kann durch geeigneter Permutation von I_w auch jeder andere verwendet werden.

Der folgende Satz führt die Definitionen und Sätze dieses Abschnittes zusammen.

SATZ 3 (DARSTELLUNGSSATZ FÜR SPITZE POLYEDER)
Sei $\mathcal{P}(D, d)$ ein spitzes Polyeder. Dann gibt es Ecken $v_1, \ldots, v_m$ und Extremalen $w_1, \ldots, w_l$, so daß gilt

$$\mathcal{P}(D, d) = \operatorname{conv}(\{v_1, \ldots, v_m\}) + \operatorname{cone}(\{w_1, \ldots, w_l\}).$$

Dabei sind die Vektoren v_i eindeutig und die Vektoren w_j bis auf Skalierung eindeutig bestimmt.

Schließlich wird mit dem folgenden Satz die Grundlage für Simplex-Algorithmen gelegt. Wegen seiner Bedeutung für die vorliegende Arbeit wird auch ein Beweis angegeben.

SATZ 4
Gegeben sei ein LP der Form (1.1) mit $n < k$ und $\operatorname{rang}(D) = n$. *Hat das LP eine optimale Lösung, so gibt es eine Ecke von $\mathcal{P}(D, d)$, an der sie angenommen wird.*

Beweis:
Sei x^* optimale Lösung des LPs. Wegen rang $(D) = n$ ist $\mathcal{P}(D,d)$ spitz. Nach Satz 3 kann x^* dargestellt werden als

$$x^* = \sum_{i=1}^{m} \lambda_i v_i + \sum_{i=1}^{l} \mu_i w_i,$$

wobei $\lambda, \mu \geq 0$ und $1^T\lambda = 1$ gilt. Die Vektoren v_i sind die Ecken von $\mathcal{P}(D,d)$ und die Vektoren w_i bilden eine Kegelbasis von $\mathcal{P}(D,0)$. Für alle i mit $\mu_i > 0$ ist $c^T w_i = 0$, denn sonst gäbe es einen zulässigen Vektor mit geringerem Zielfunktionswert: Falls $c^T w_i < 0$ gilt nämlich für den Vektor $x^+ = x^* + w_i \in \mathcal{P}(D,d)$ die Ungleichung $c^T x^+ = c^T x^* + c^T w_i < c^T x^*$, und falls $c^T w_i > 0$ gilt für den Vektor $x^- = x^* - \mu_i w_i \in \mathcal{P}(D,d)$ die Ungleichung $c^T x^- = c^T x^* - \mu_i c^T w_i < c^T x^*$. Somit hat auch der Vektor

$$x' = \sum_{i=1}^{m} \lambda_i v_i$$

denselben Zielfunktionswert $c^T x' = c^T x^*$ wie x^*, und es gilt $x' \in \mathcal{P}(D,d)$. Wähle nun i^* so, daß für alle $1 \leq i \leq m$ gilt $c^T v_i \geq c^T v_{i^*}$. Damit gilt

$$c^T x' = \sum_{i=1}^{m} c^T \lambda_i v_i \geq \sum_{i=1}^{m} \lambda_i c^T v_{i^*} = c^T v_{i^*} \geq c^T x',$$

wobei die letzte Ungleichung wegen $v_{i^*}, x' \in \mathcal{P}(D,d)$ aus der Optimalität von x' folgt. Insgesamt gilt $c^T x' = c^T v_{i^*}$. Der optimale Zielfunktionswert wird also auch an der Ecke $v_{i^*} \in \mathcal{P}(D,d)$ angenommen, was den Beweis abschließt.

■

Aufgrund dieses Satzes ist es das Ziel von Simplex-Algorithmen eine optimale Ecke des Polyeders zu finden.

1.2 Die Grundalgorithmen

In diesem Abschnitt werden die Grundversionen von Simplex-Algorithmen beschrieben. Dabei werden sowohl primale als auch duale Algorithmen jeweils für zwei verschiedene Darstellungen der sog. Basis behandelt.

Zunächst werden einfache LPs betrachtet, bei denen nicht so viel Notation aufgewendet werden muß, dafür aber die mathematischen und geometrischen Gegebenheiten deutlicher hervortreten. Für sie wird in Abschnitt 1.2.1 die Zeilenbasis eingeführt und dafür der primale und duale Simplex-Algorithmus entwickelt. Im nächsten Abschnitt werden die Algorithmen erneut aufgestellt, diesmal jedoch in der Formulierung mithilfe einer Spaltenbasis. Dabei zeigen sich große Ähnlichkeiten zwischen der zeilenweisen und spaltenweisen Darstellung, die in der Dualität begründet liegen und in Abschnitt 1.2.3 erläutert werden. In

Abschnitt 1.2.4 werden schließlich allgemeine LPs behandelt und die Simplex-Algorithmen in einer weitgehend von der gewählten Darstellung unabhängigen Form beschrieben. Dies ist die Form, auf der die in der vorliegenden Arbeit vorgestellten Implementierungen basieren.

1.2.1 Die Zeilenbasis

Betrachte wieder das LP

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Dx & \geq \quad d, \end{array} \tag{1.1}$$

wobei $c, x \in \mathbb{R}^n$, $d \in \mathbb{R}^k$ und $D \in \mathbb{R}^{k \times n}$ und $k \geq n$ gelte und D vollen Rang habe. Wenn nichts weiter gesagt wird, werden im folgenden diese Setzungen zugrundegelegt.

Die Idee von Simplex-Algorithmen basiert auf Satz 4: Wenn das LP (1.1) eine optimale Lösung hat, so gibt es eine Ecke des zugehörigen Polyeders $\mathcal{P}(D,d) = \{x \in \mathbb{R}^n : Dx \geq d\}$, an der sie angenommen wird. Nach Definition 1 ist jede Ecke des Polyeders $\mathcal{P}(D,d)$ der Schnittpunkt von n seiner Hyperebenen. Das Ziel des Simplex-Algorithmus' ist deshalb, n (linear unabhängige) Hyperebenen von $\mathcal{P}(D,d)$ zu finden, so daß ihr Schnittpunkt x^*

- *zulässig*, d.h. $x^* \in \mathcal{P}(D,d)$, und
- *optimal*, d.h. für alle $x \in \mathcal{P}(D,d)$ gilt $c^T x \geq c^T x^*$, ist.

Zur Beschreibung einer solchen Menge von Hyperebenen wird der Begriff einer *Simplex-Basis* verwendet. Entgegen dem üblichen Vorgehen in der Literatur zum Simplex-Algorithmus führen wir zunächst eine *Zeilenbasis* ein (auch *aktive Menge* genannt). Wir tun dies aus zwei Gründen. Zum einen erlaubt die Verwendung der Zeilenbasis eine geometrisch anschaulichere Beschreibung der Simplex-Algorithmen, da sinnvolle Beispiele bereits im $\mathbb{R}^2$ angegeben werden können. Zum anderen kommt die Zeilenbasis auch in den hier vorzustellenden Implementierungen zum Einsatz. In Abschnitt 1.2.2 wird die übliche Definition einer *Spaltenbasis* aus einem Spezialfall abgeleitet.

Definition 2 (Zeilenbasis)
Ein geordnetes Paar $Z = (P,Q)$ von Indexvektoren $P, Q \subseteq \{1,\ldots,k\}$ heißt Zeilenbasis *(zum LP (1.1)), falls folgendes gilt:*

1. $P \cup Q = \{1,\ldots,k\}$,
2. $P \cap Q = \emptyset$,
3. $|P| = n$ *und*
4. $D_{P.}$ *ist nicht singulär.*

Gemäß Bedingungen 1 und 2 zerlegt eine Zeilenbasis die Menge von Ungleichungen in zwei Teilmengen P und Q. Die Stützhyperebenen $D_{P.}x = d_P$ der zu P gehörenden

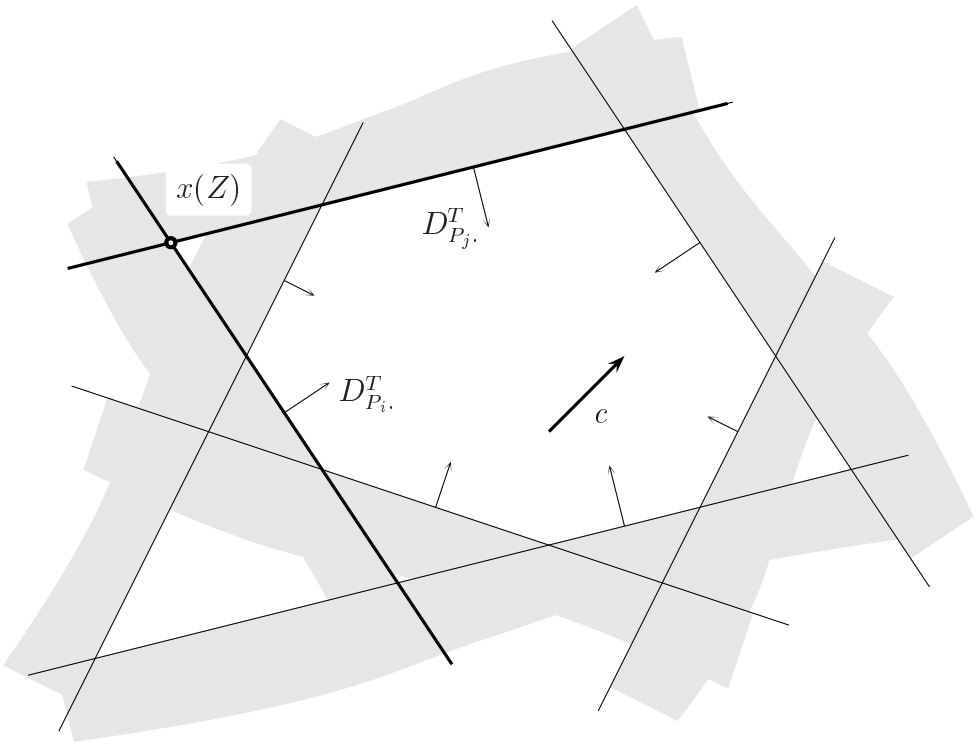


Abbildung 1.2: Eine Zeilenbasis $Z = (P, Q)$ zu einem LP zeichnet n Stützhyperebenen des Polyeders aus, die fett dargestellt sind. Ihr Schnittpunkt ist der Basislösungsvektor $x(Z)$.

Ungleichungen dienen der Bestimmung eines Schnittpunktes als mögliche optimale Lösung. Die Bedingungen 3 und 4 postulieren, daß das Gleichungssystem $D_{P.}x = d_P$ eindeutig lösbar ist (vgl. Abb. 1.2).

Zu einer Zeilenbasis werden noch weitere Begriffe und Vektoren eingeführt:

DEFINITION 3
Sei $Z = (P, Q)$ eine Zeilenbasis. Dann heißen die Ungleichungen $D_{p.}x \geq d_p$, mit $p \in P$, Basisungleichungen *(zur Basis Z); alle anderen heißen* Nichtbasisungleichungen. *Die Matrix $D_{P.}$ heißt die* Basismatrix. *Ferner wird definiert:*

der Basislösungsvektor	$x(Z) = D_{P.}^{-1} d_P,$	(1.4)
der Vektor der Schlupfvariablen	$s(Z) = D \cdot x(Z)$	(1.5)
und der Vektor der Dualvariablen	$y(Z)^T = c^T D_{P.}^{-1}.$	(1.6)

Eine Schlupfvariable wird zur Transformation einer Ungleichung $a^T x \geq \alpha$ in eine Gleichung $a^T x - s = 0$ und eine Variablenschranke $s \geq \alpha$ benutzt (vgl. Abschnitt 1.2.4). Die Definition (1.5) unterscheidet sich von der in der Literatur üblichen, bietet jedoch den Vorteil der symmetrischen Erweiterbarkeit auf sog. *Bereichsungleichungen* $\alpha' \leq a^T x \leq \alpha''$.

Simplex-Algorithmen versuchen eine Basis zu finden, deren Basislösungsvektor sowohl zulässig als auch optimal ist. Dazu wird, ausgehend von einer bestehenden Basis, iterativ je ein Index aus P mit einem aus Q ausgetauscht, bis eine Basis gefunden wird, deren Lösungsvektor beide Bedingungen erfüllt. Ein solcher Austausch wird ein *Pivot-Schritt* genannt. Dabei muß allerdings die Basismatrix regulär bleiben. Wann der Austausch einer Zeile einer reglären Matrix wieder zu einer regulären Matrix führt, wird durch den folgenden Satz beschrieben.

SATZ 5 (ZEILENTAUSCH)
Sei $D \in \mathbb{R}^{n\times n}$ regulär, $r \in \mathbb{R}^n$ und $l \in \{1,\ldots,n\}$. Die Matrix $D' = D + e_l(r^T - D_{l.})$, die durch Austausch der l-ten Zeile von D mit r^T entsteht, ist genau dann regulär, wenn

$$(r^T D^{-1})_l \neq 0. \tag{1.7}$$

Seien in diesem Fall $\rho \in \mathbb{R}$ und $f, f', \bar{f}, \bar{f}', g, g', \bar{g} \in \mathbb{R}^n$, mit $f = D^{-1}\bar{f}$, $g^T = \bar{g}^T D^{-1}$ und $\bar{f}' = \bar{f} + (\rho - \bar{f}_l)e_l$. Dann gilt für $f' = D'^{-1}\bar{f}'$:

$$f' = f + \theta \cdot D_{.l}^{-1} \quad , \text{ mit} \tag{1.8}$$

$$\theta = \frac{\rho - r^T f}{(r^T D^{-1})_l} \tag{1.9}$$

und für $g'^T = \bar{g}^T D'^{-1}$

$$g'^T = g^T + \phi \cdot (r^T D^{-1} - e_l^T), \text{ mit} \tag{1.10}$$

$$\phi = \frac{-g_l}{(r^T D^{-1})_l}. \tag{1.11}$$

BEWEIS:
Zunächst ist zu bemerken, daß D' aus D durch linksseitige Multiplikation mit

$$V = I + e_l(r^T D^{-1} - e_l^T) \tag{1.12}$$

hervorgeht, denn damit gilt $VD = D + e_l(r^T D^{-1} D - e_l^T D) = D + e_l(r^T - D_{l.}) = D'$. Nach dem Determinantenproduktsatz ist $\det(D') = \det(V) \cdot \det(D)$. Da D regulär und somit $\det(D) \neq 0$, ist D' genau dann regulär, wenn $\det(V) \neq 0$. Nun gilt aber nach dem Determinantenentwicklungssatz $|\det(V)| = |(r^T D^{-1})_l|$, woraus Behauptung (1.7) folgt.

Die Gleichungen (1.8) und (1.9) werden durch Einsetzen in $D'f'$ gezeigt:

$$\begin{aligned}
D'f' &= VD(f + \theta D_{.l}^{-1}) = V(Df + \theta D D^{-1} e_l) \\
&= Df + e_l(r^T D^{-1} - e_l^T)Df + \theta e_l + \theta e_l(r^T D^{-1} - e_l^T)e_l \\
&= \bar{f} + e_l(r^T f - \bar{f}_l) + \theta e_l + \theta(r^T D^{-1})_l e_l - \theta e_l \\
&= \bar{f} + (r^T f - \bar{f}_l)e_l + (\rho - r^T f)e_l \\
&= \bar{f} + (\rho - \bar{f}_l)e_l \\
&= \bar{f}'.
\end{aligned}$$

Entsprechend gelten (1.10) und (1.11), denn durch Einsetzen in $g'^T D'$ erhält man

$$\begin{aligned} g'^T D' &= [g^T + \phi(r^T D^{-1} - e_l^T)]VD \\ &= g^T VD + \phi(r^T D^{-1} - e_l^T)VD \\ &= g^T D + g^T e_l(r^T D^{-1} - e_l^T)D + \phi(r^T D^{-1} - e_l^T)D \\ &\quad +\phi(r^T D^{-1} - e_l^T)e_l(r^T D^{-1} - e_l^T)D \\ &= \bar{g}^T + g_l(r^T - D_{l.}) + \phi(r^T - D_{l.}) + \phi[r^T D^{-1} e_l(r^T - D_{l.}) - r^T + D_{l.}] \\ &= \bar{g}^T + g_l \left[r^T - D_{l.} - \frac{(r^T D^{-1})_l (r^T - D_{l.})}{(r^T D^{-1})_l} \right] \\ &= \bar{g}^T. \end{aligned}$$

■

Angewandt auf die Zeilenbasismatrix folgt aus diesem Satz

Satz 6 (Zeilenbasistausch)
Sei $Z = (P, Q)$ eine Zeilenbasis zum LP (1.1). Seien $i \in \{1, \ldots, n\}$ und $j \in Q$. Dann ist $Z' = (P', Q')$ mit $P' = P \setminus \{P_i\} \cup \{j\}$ und $Q' = Q \setminus \{j\} \cup \{P_i\}$ genau dann eine Zeilenbasis von (1.1), wenn

$$(D_{j.} \cdot D_{P.}^{-1})_i \neq 0. \tag{1.13}$$

Ferner sind

$$\begin{aligned} x(Z') &= x(Z) + \Theta(i,j) \cdot (D_{P.}^{-1})_{.i}, & (1.14) \\ s(Z') &= s(Z) + \Theta(i,j) \cdot D(D_{P.}^{-1})_{.i} \text{ und} & (1.15) \\ y^T(Z') &= y^T(Z) + \Phi(i,j) \cdot (D_{j.} D_{P.}^{-1} - e_i^T) & (1.16) \end{aligned}$$

wobei

$$\begin{aligned} \Theta(i,j) &= \frac{d_j - s_j(Z)}{(D_{j.} D_{P.}^{-1})_i} & (1.17) \\ \text{und } \Phi(i,j) &= \frac{-y_i(Z)}{(D_{j.} D_{P.}^{-1})_i}. & (1.18) \end{aligned}$$

Beweis:
Bis auf (1.15) folgt alles aus Satz 5, und zwar mit $D = D_{P.}$, $D' = D_{P'.}$, $\theta = \Theta(i,j)$, $\phi = \Phi(i,j)$, $f = x(Z)$, $f' = x(Z')$, $\bar{f} = d_P$, $g = y(Z)$, $g' = y(Z')$, $\bar{g} = c$, $\rho = d_j$, $r^T = D_{j.}$ und $l = i$. Gleichung (1.15) gilt dann wegen $s(Z') = Dx(Z') = s(Z) + \Theta(i,j)D(D_{P.}^{-1})_j$.

■

Ein Simplex-Algorithmus führt so lange Pivot-Schritte aus, bis eine Basis gefunden wird, dessen Lösungsvektor sowohl optimal als auch zulässig ist, bzw. bis die Unbeschränktheit oder Unzulässigkeit des LPs nachgewiesen werden kann. Wie aber erkennt man die Optimalität und Zulässigkeit des Basislösungsvektors? Letzteres erreicht man leicht durch Testen der Schranken des Schlupfvektors $s(Z) = D \cdot x(Z) \geq d$, wobei nach definition von s nur die Elemente s_Q überprüft werden müssen. Die Überprüfung der Optimalität geschieht mit Hilfe von folgendem

Satz 7 (Simplex-Kriterium für eine Zeilenbasis)
Sei $Z = (P, Q)$ eine Zeilenbasis zum LP (1.1). $x(Z)$ ist optimal, wenn

$$y(Z) \geq 0. \tag{1.19}$$

Beweis:
Nach Definition 1 ist $x(Z)$ die einzige Ecke von $\mathcal{P}(D_{P.}, d_P)$. Demnach lassen sich gemäß Satz 3 alle $x \in \mathcal{P}(D_{P.}, d_P)$ als

$$x = x(Z) + \sum_{i=1}^{n} \tau_i \cdot t_i$$

darstellen, wobei die Vektoren $t_i = D_{P.}^{-1} e_i$, $i \in \{1, \ldots, n\}$, eine Kegelbasis von $\mathcal{P}(D_{P.}, 0)$ bilden und $\tau_1, \ldots, \tau_n \geq 0$ gilt. Demnach gilt für alle $x \in \mathcal{P}(D_{P.}, d_P)$:

$$\begin{aligned} c^T x &= c^T x(Z) + c^T \sum_{i=1}^{n} \tau_i \cdot D_{P.}^{-1} e_i \\ &= c^T x(Z) + y^T(Z) \sum_{i=1}^{n} \tau_i \cdot e_i \\ &\geq c^T x(Z), \end{aligned}$$

wobei die letzte Ungleichung wegen $y^T(Z) \geq 0$ gilt. Da $\mathcal{P}(D_{P.}, d_P) \supseteq \mathcal{P}(D, d)$ folgt die Behauptung.

■

Anschaulich besagt das Simplex-Kriterium, daß eine Zeilenbasis optimal ist, wenn es keine Extremale gibt, entlang der eine Verbesserung des Zielfunktionswertes möglich wäre (vgl. Abb. 1.3).

Definition 4
Sei $Z = (P, Q)$ eine Zeilenbasis zum LP (1.1). Z heißt zulässig, *wenn*

$$s_Q(Z) \geq d \tag{1.20}$$

und Z heißt optimal, *wenn*

$$y(Z) \geq 0. \tag{1.21}$$

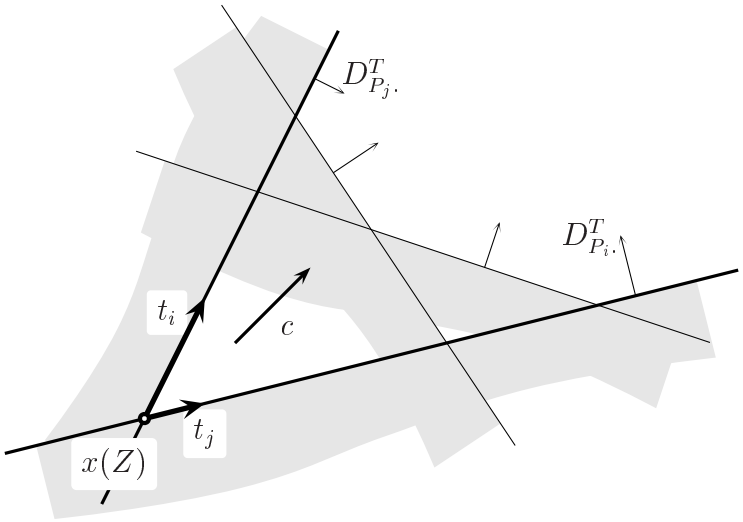


Abbildung 1.3: Der Kegel einer optimalen Basis hat keine Extremale, entlang der eine Verbesserung des Zielfunktionswertes möglich ist.

Diese Definition ermöglicht es, die Ergebnisse dieses Abschnittes mit folgendem Satz zusammenzufassen.

SATZ 8

Sei $n, k \in \mathbb{N}$, $0 < n < k$, $c, x \in \mathbb{R}^n$, $d \in \mathbb{R}^k$ und $D \in \mathbb{R}^{k \times n}$ habe vollen Rang. Sei $Z = (P, Q)$ eine Zeilenbasis des LPs

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Dx & \geq \quad d. \end{array} \tag{1.1}$$

Der Basislösungsvektor $x(Z)$ ist eine optimale Lösung von (1.1) wenn Z optimal und zulässig ist.

Dieser Satz folgt unmittelbar aus den Definitionen und Satz 7.

Der Basistausch bei Simplex-Algorithmen soll "gerichtet" erfolgen. Dabei unterscheidet man zwei Strategien: Der *primale* Algorithmus benötigt eine *zulässige* Basis und versucht bei jeder Iteration, den Zielfunktionswert zu verbessern, ohne dabei die Zulässigkeit zu verlieren. Der entsprechende Algorithmus wird im folgenden Abschnitt entwickelt. Dagegen arbeitet der *duale* Simplex auf einer *optimalen* Basis und versucht bei jeder Iteration, die Unzulässigkeit zu verringern während die Optimalität erhalten wird. Dieser Algorithmus wird in Abschnitt 1.2.1.2 beschrieben.

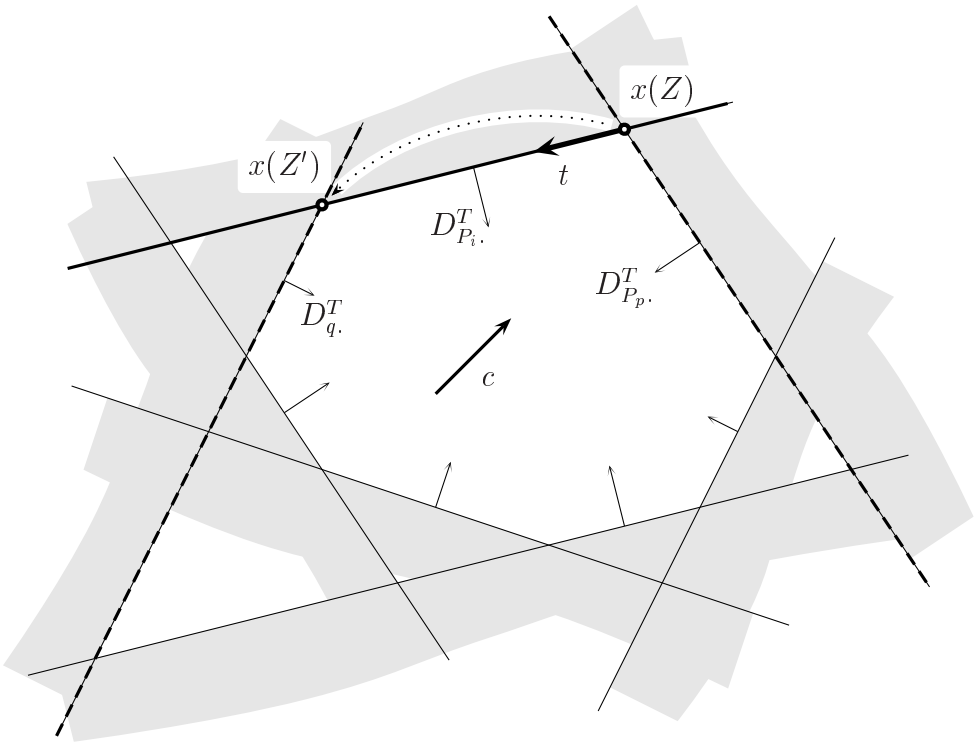


Abbildung 1.4: Bei einem Schritt des primalen Simplex wird die aktuelle Basislösung so weit entlang der Extremalen t verschoben, wie es ein Beibehalten der Zulässigkeit erlaubt. Dazu wird eine Stützhyperebene der Basis durch die neue ausgetauscht. Beide sind gestrichelt dargestellt.

1.2.1.1 Primaler Algorithmus

Es wird nun der primale Simplex-Algorithmus für eine Zeilenbasis beschrieben und seine partielle Korrektheit[1] beweisen. Im allgemeinen kann jedoch nicht gezeigt werden, daß der Algorithmus terminiert. Auf dieses Problem sowie das Generieren einer zulässigen Anfangsbasis wird in den Abschitten 1.5 und 1.4 eingegangen.

Der primale Simplex-Algorithmus arbeitet mit einer zulässigen Basis. Diese wird durch eine Folge von Pivot-Schritten solange abgeändert, bis eine optimale Basis gefunden wird oder die Unbeschränktheit des LPs nachgewiesen werden kann. Dabei wird die Zulässigkeit gewahrt, so daß bei Terminierung im beschränkten Fall eine optimale und zulässige Basis vorliegt, dessen Lösungsvektor nach Satz 8 eine optimale Lösung des LPs ist. Die jeweils neue Basis wird so gewählt, daß sich der Zielfunktionswert des Basislösungsvektors nicht verschlechtert.

Sei also $Z = (P, Q)$ eine zulässige Zeilenbasis zum LP (1.1). Ist Z auch optimal, so ist $x(Z)$ bereits ein optimaler Lösungsvektor von (1.1). Andernfalls gibt es eine Extremale,

[1] Ein Algorithmus heißt *partiell korrekt*, wenn bei seiner Termination die korrekte Lösung vorliegt. Für die vollständige Korrektheit muß auch die Termination des Algorithmus' gewährleistet sein.

etwa $t = (D_{P.}^{-1})e_p$, so daß jeder Vektor $x(\theta) = x(Z) + \theta \cdot t$ für $\theta > 0$ einen besseren Zielfunktionswert aufweist als $x(Z)$, d.h. $c^T x(\theta) < c^T x(Z)$. Nun soll θ maximal gewählt werden, so daß $x(\theta)$ noch zulässig bleibt (vgl. Abb. 1.4). Zu jedem Vektor $x(\theta)$ kann man einen Schlupfvektor $s(\theta) = D \cdot x(\theta) = s(Z) + \theta \cdot Dt$ einführen. Offenbar kann θ nur so groß gewählt werden, bis die erste Schlupfvariable, etwa $s(\theta)_q$, an ihre Grenze d_q stößt. Ein Austausch der p-ten Basisungleichung mit Ungleichung q führt dann wieder zu einer zulässigen Basis. Dies ist der Inhalt von folgendem

Satz 9 (Primaler Quotiententest für eine Zeilenbasis)
Sei $Z = (P, Q)$ eine zulässige Zeilenbasis zum LP (1.1), $p \in \{1, \ldots, N\}$, mit $y_p(Z) < 0$, und $t = D_{P.}^{-1} e_p$.
Gilt $Dt \geq 0$, so ist das LP (1.1) unbeschränkt. Andernfalls ist für

$$q \in \arg\min \left\{ \frac{d_i - s_i(Z)}{D_{i.}t} : D_{i.}t < 0, 1 \leq i \leq k \right\} \tag{1.22}$$

$Z' = (P', Q')$, mit $P' = P \setminus \{P_p\} \cup \{q\}$ und $Q' = Q \setminus \{q\} \cup \{P_p\}$ eine zulässige Zeilenbasis von (1.1). Ferner gilt

$$\Theta = \frac{d_q - s_q(Z)}{D_{q.}t} \geq 0 \tag{1.23}$$

$$\text{und} \quad c^T x(Z') = c^T x(Z) + \Theta \cdot y_p(Z). \tag{1.24}$$

Beweis:
Fall 1 ($Dt \geq 0$):
Es ist zu zeigen, daß es für alle $M \in \mathbb{R}$ ein $x \in \mathcal{P}(D, d)$ gibt, so daß $c^T x < M$. Für $M > c^T x(Z)$ leistet $x(Z)$ das gewünschte. Sei also $M \leq c^T x(Z)$. Setze $x = x(Z) + \frac{M-1-c^T x(Z)}{y_p} t$. Dann ist $c^T x = c^T x(Z) + \frac{M-1-c^T x(Z)}{y_p} c^T t = c^T x(Z) + (M - 1 - c^T x(Z)) = M - 1 < M$ und wegen $y_p < 0$, $Dt \geq 0$ und $M - 1 - c^T x(Z) < 0$ gilt $Dx = Dx(Z) + \frac{M-1-c^T x(Z)}{y_p} Dt \geq Dx(Z) = s(Z) \geq d$.

Fall 2 ($Dt \not\geq 0$):
Zunächst gilt $q \in Q$, denn $D_{P.}t = D_{P.}D_{P.}^{-1}e_p = e_p \geq 0$. Nach Satz 6 ist Z' eine Zeilenbasis mit Schlupfvariablen $s(Z') = s(Z) + \Theta Dt$. Da Z zulässig ist, d.h. $s(Z) \geq d$, und $D_{q.}t < 0$ gilt $\Theta \geq 0$. Wegen (1.22) gilt für $i \in \{1, \ldots, k\}$ mit $D_{i.}t < 0$: $s_i(Z') = s_i(Z) + \Theta D_{i.}t \geq s_i(Z) + \frac{d_i - s_i(Z)}{D_{i.}t} D_{i.}t = d_i$. Für $i \in \{1, \ldots, k\}$ mit $D_{i.}t \geq 0$ gilt $s_i(Z') = s_i(Z) + \Theta D_{i.}t \geq s_i(Z) \geq d_i$. Damit ist auch die Zulässigkeit von Z' bewiesen. Gleichung (1.24) folgt schließlich aus (1.14).

■

Satz 9 bietet die Grundlage für den primalen Simplex-Algorithmus. Bei seiner Darstellung verwenden wir andere Bezeichner für die relevanten Vektoren (vgl. Initialisierung in Schritt 0), um später die Ähnlichkeit der Algorithmen für Zeilen- und Spaltenbasis zu verdeutlichen.

ALGORITHMUS 1 (PRIMALER SIMPLEX-ALGORITHMUS IN ZEILENDARSTELLUNG)

Sei $Z = (P, Q)$ eine zulässige Zeilenbasis zum LP (1.1).

Schritt 0 (Initialisierung):

$$\begin{array}{lcll} A & \leftarrow & D^T & \\ B & \leftarrow & P & \\ h^T & \leftarrow & d_B^T A_{B.}^{-1} & = \quad x^T(Z) \\ g^T & \leftarrow & h^T \cdot A & = \quad s^T(Z) \\ f & \leftarrow & A_{.B}^{-1} c & = \quad y(Z) \end{array}$$

Schritt 1 (Pricing):
Falls $f \geq 0$ terminiere: h ist optimaler Lösungsvektor des LPs;
sonst wähle p, mit $f_p < 0$.

Schritt 2: $\Delta h^T \leftarrow e_p^T A_{B.}^{-1}$
$\Delta g^T \leftarrow \Delta^T \cdot hA$

Schritt 3 (Quotiententest):
Falls $\Delta g \geq 0$ terminiere: Das LP ist unbeschränkt.
Sonst wähle $q \in \arg\min\{(d_i - g_i)/\Delta g_i : \Delta g_i < 0\}$

Schritt 4: $\Delta f \leftarrow A_{.B}^{-1} A_{.q}$

Schritt 5 (Update):

$$\begin{array}{lcl} B_p & \leftarrow & q \\ \Theta & \leftarrow & (d_q - g_q)/\Delta g_q \\ \Phi & \leftarrow & -f_p/\Delta f_p \\ h & \leftarrow & h' = h + \Theta \cdot \Delta h \\ g & \leftarrow & g' = g + \Theta \cdot \Delta g \\ f & \leftarrow & f' = f + \Phi(\Delta f - e_p) \end{array}$$

Schritt 6: Gehe zu Schritt 1

SATZ 10 (PARTIELLE KORREKTHEIT VON ALGORITHMUS 1)
Algorithmus 1 arbeitet partiell korrekt.

Beweis:
Terminiert Algorithmus 1 in Schritt 1 dann ist h nach Satz 8 ein optimaler Lösungsvektor von LP (1.1). Wenn Algorithmus 1 in Schritt 3 terminiert, ist das LP nach Satz 9 unbeschränkt. Da $\Delta g_q = (A^T(A^T)^{-1}_{B.}e_p)_q = (A^T)_{q.}(A^T)^{-1}_{B.}e_p = \Delta f_p$ gilt schließlich nach den Sätzen 6 und 9, daß in Schritt 6 Z wieder eine zulässige Basis mit dualen Variablen f, Schlupfvariablen g, und Lösungsvektor h ist, so daß die Voraussetzungen für Schritt 1 wieder erfüllt sind.

■

Aufgrund von Gleichung (1.24) hat nach jeder Iteration der neue Lösungsvektor h' höchstens denselben Zielfunktionswert wie h. Für $d_q \neq g_q$ ist er kleiner, während sonst $h = h'$ gilt, d.h. derselbe Lösungsvektor wird von dem neuen Satz von Basishyperebenen definiert. Solch ein Pivot-Schritt heißt *degeneriert*. Er ist nur möglich, wenn sich beim Basislösungsvektor mehr als n Stützhyperebenen von $\mathcal{P}(D,d)$ schneiden. Solch eine Ecke heißt *primal degeneriert*.

1.2.1.2 Dualer Algorithmus

Im Gegensatz zum primalen Algorithmus arbeitet der duale auf einer optimalen Basis, d.h. Gleichung (1.21) ist erfüllt. Sofern die Basis nicht auch zulässig ist, gibt es eine verletzte Ungleichung, etwa q. Sie soll in die Basis aufgenommen werden. Dazu muß eine andere die Basis verlassen und zwar so, daß die neue Basis wieder optimal ist. Anschaulich ist es die Basisungleichung p, mit der der neue Lösungsvektor einen minimalen Zielfunktionswert aufweist (vgl. Abb. 1.5). Nach den Gleichungen (1.14) und (1.17) muß p also so gewählt werden, daß $c^T x(Z') = c^T x(Z) + \Theta(p,q)(c^T D^{-1}_{P.})_p$ und somit $\Theta(p,q)\cdot(c^T D^{-1}_{P.})_p = y(Z)_p(d_q - s_q(Z))/(D_{q.}D^{-1}_{P.})_p$ minimiert wird. Da aber $d_q > s_q(Z)$, ist dies gleichbedeutend mit der Minimierung von $y(Z)_p/(D_{q.}D^{-1}_{P.})_p$. Dies wird von folgenden Satz präzisiert.

Satz 11 (Dualer Quotiententest für eine Zeilenbasis)
Sei $Z = (P,Q)$ eine optimale Zeilenbasis zum LP (1.1), $q \in Q$ der Index einer verletzten Ungleichung, also $s(Z)_q < d_q$, und $t^T = D_{q.}D^{-1}_{P.}$.
Gilt $t \leq 0$, so ist das LP (1.1) unzulässig. Andernfalls ist für

$$p \in \arg\max\left\{\frac{-y_i(Z)}{t_i} : t_i > 0, 1 \leq i \leq n\right\} \tag{1.25}$$

$Z' = (P',Q')$, mit $P' = P \setminus \{P_p\} \cup \{q\}$ und $Q' = Q \setminus \{q\} \cup \{P_p\}$ eine optimale Zeilenbasis von (1.1). Ferner gilt

$$\Phi = \frac{-y_p(Z)}{t_p} \leq 0 \tag{1.26}$$

und

$$c^T x(Z') = c^T x(Z) + \Phi \cdot (s_q(Z) - d_q). \tag{1.27}$$

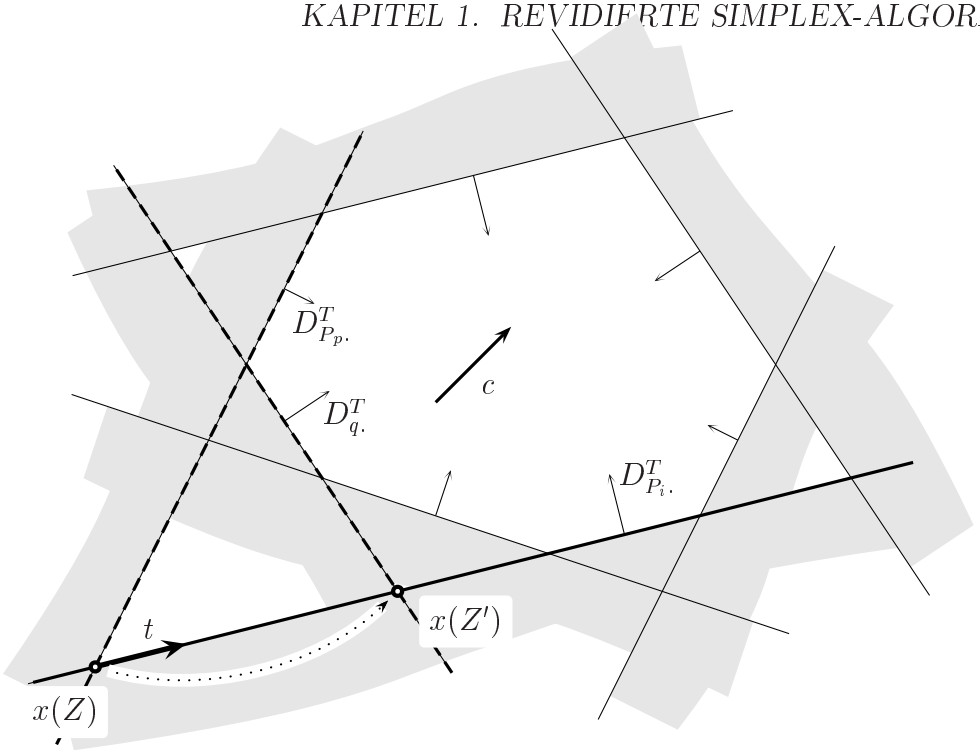


Abbildung 1.5: Bei einem Schritt des dualen Simplex wird eine verletzte Ungleichung in die Basis aufgenommen, wofür eine andere die Basis verlassen muß. Die beiden betroffenen Stützhyperebenen sind gestrichelt dargestellt.

Beweis:

Fall 1 ($t \leq 0$)**:**
Nach Satz 1 ist $x(Z)$ die einzige Ecke von $\mathcal{P}(D_{P\cdot}, d_P)$. Demnach lassen sich gemäß Satz 3 und 2 alle $x \in \mathcal{P}(D_{P\cdot}, d_P)$ als

$$x = x(Z) + \sum_{i=1}^{n} \tau_i \cdot D_{P\cdot}^{-1} e_i$$

darstellen, wobei $\tau_1, \ldots, \tau_n \geq 0$ gilt. Für alle $x \in \mathcal{P}(D_{P\cdot}, d_P)$ gilt somit $D_{q\cdot}x = D_{q\cdot}x(Z) + \sum_{i=1}^{n} \tau_i D_{q\cdot} D_{P\cdot}^{-1} e_i < d_q + \sum_{i=1}^{n} \tau_i t_i \leq d_q$, d.h. es gibt kein $x \in \mathcal{P}(D_{P\cdot}, d_P)$, mit $D_{q\cdot}x \geq d_q$. Da also $\emptyset = \mathcal{P}(D_{P\cdot}, d_P) \cap \{x : D_{q\cdot}x \geq d_q\} \supseteq \mathcal{P}(D, d)$ folgt die Behauptung.

Fall 2 ($t \not\leq 0$)**:**
Wegen $t_p = (D_{q\cdot} D_{P\cdot}^{-1})_p > 0$ ist nach Satz 6 Z' eine Zeilenbasis mit dualen Variablen $y^T(Z') = y^T(Z) + \Phi(t^T - e_p^T)$. Da $y(Z) \geq 0$ und $t_p > 0$ ist $\Phi \leq 0$. Für $i \in \{1, \ldots, n\} \setminus \{p\}$ gilt $y_i(Z') = y_i(Z) + \Phi t_i$. Falls $t_i \leq 0$ gilt somit $y_i(Z') \geq y_i(Z) \geq 0$. Andernfalls gilt wegen (1.25) $y_i(Z') = y_i(Z) + \Phi t_i \geq y_i(Z) + (-y_i(Z))/t_i \cdot t_i = 0$. Ferner gilt $y_p(Z') = y_p(Z) - y_p(Z)/t_p \cdot (t_p - 1) = y_p(Z)/t_p = -\Phi \geq 0$, was die Optimalität von Z' zeigt. Schließlich gilt nach (1.14) $c^T x(Z') = c^T x(Z) + (d_q - s_q(Z))/t_p \cdot c^T D_{P\cdot}^{-1} e_p = c^T x(Z) - (s_q(Z) - d_q)/t_p \cdot y_p = c^T x(Z) + \Phi(s_q(Z) - d_q)$.

■

Mit diesem Satz kann nun auch der duale Simplex-Algorithmus in Zeilendarstellung aufgestellt werden.

ALGORITHMUS 2 (DUALER SIMPLEX-ALGORITHMUS IN ZEILENDARSTELLUNG)

Sei $Z = (P, Q)$ eine optimale Zeilenbasis von (1.1).

Schritt 0 (Initialisierung):

$$\begin{array}{lcll} A & \leftarrow & D^T & \\ B & \leftarrow & P & \\ h^T & \leftarrow & d_B^T A_{B.}^{-1} & = \quad x^T(Z) \\ g^T & \leftarrow & h^T \cdot A & = \quad s^T(Z) \\ f & \leftarrow & A_{.B}^{-1} c & = \quad y(Z) \end{array}$$

Schritt 1 (Pricing):
Falls $g \geq d$ terminiere: h ist optimaler Lösungsvektor des LPs;
sonst wähle q, mit $g_q < d_q$.

Schritt 2: $\Delta f \leftarrow A_{.B}^{-1} A_{.q}$

Schritt 3 (Quotiententest):
Falls $\Delta f \leq 0$ terminiere: Das LP ist unzulässig;
sonst wähle $p \in \arg\max\{-f_i/\Delta f_i : \Delta f_i > 0\}$

Schritt 4: $\Delta h^T \leftarrow e_p^T A_{B.}^{-1}$
$\Delta g^T \leftarrow \Delta^T \cdot hA$

Schritt 5 (Update):

$$\begin{array}{lcl} B_p & \leftarrow & q \\ \Theta & \leftarrow & (d_q - g_q)/\Delta g_q \\ \Phi & \leftarrow & -f_p/\Delta f_p \\ h & \leftarrow & h' = h + \Theta \cdot \Delta h \\ g & \leftarrow & g' = g + \Theta \cdot \Delta g \\ f & \leftarrow & f' = f + \Phi(\Delta f - e_p) \end{array}$$

Schritt 6: Gehe zu Schritt 1

SATZ 12 (PARTIELLE KORREKTHEIT VON ALGORITHMUS 2)

Algorithmus 2 arbeitet partiell korrekt.

BEWEIS:

Terminiert Algorithmus 2 in Schritt 1 dann ist nach Satz 8 h ein optimaler Lösungsvektor von LP (1.1). Wenn Algorithmus 2 in Schritt 3 terminiert, ist das LP nach Satz 11 unzulässig. Schließlich gilt nach den Sätzen 6 und 11, daß in Schritt 6 Z wieder eine optimale Basis mit dualen Variablen f, Schlupfvariablen g und Lösungsvektor h ist, so daß die Voraussetzungen für Schritt 1 wieder erfüllt sind.

■

Anstelle die Vektoren g und h in den Schritten 4 und 5 zu aktualisieren, kann man sie ebenso neu berechnen (dies wird auch z.B. bei partiellem Pricing in der Praxis durchgeführt). Die gewählte Darstellung verdeutlicht jedoch besser die Änlichkeit von primalem und dualem Algorithmus und ist bei vollständigem Pricing auch effizienter [16]. Alle numerischen Berechnungen in den Schritten 2, 4 und 5 treten in beiden Algorithmen in gleicher Weise auf. Lediglich die Auswahl der Indizes p und q wurde vertauscht: Der primale Algorithmus (in Zeilendarstellung) wählt beim Pricing zunächst einen Vektor, der die Basismatrix verlassen soll, während der duale Algorithmus im Pricing einen Vektor auswählt, der in die Basis eintreten soll. Wir nennen daher Algorithmus 1 auch den *entfernenden* und Algorithmus 2 den *einfügenden* Algorithmus.

Wie beim primalen Algorithmus ändert sich auch beim dualen der Zielfunktionswert des Basislösungsvektors, falls der Quotiententest einen von Null verschiedenen Wert liefert. Der Fall $\Phi = 0$ tritt auf, wenn der neue und alte Lösungsvektor auf gleicher Höhe (in bezug auf c) liegen, d.h. $y(Z)_p = 0$. Solch einen Pivot-Schritt nennt man wieder *degeneriert*.

1.2.2 Die Spaltenbasis

In diesem Abschnitt werden sowohl der duale als auch primale Simplex-Algorithmus erneut vorgestellt, diesmal jedoch mit einer Spaltenbasis. Dies ist die Darstellung, wie man sie üblicherweise in der Literatur findet. Sie wird hier als Umformulierung eines Spezialfalls der zeilenweisen Darstellung der Basis vorgestellt.

Sofern nichts weiteres gesagt wird, sei in diesem Abschnitt folgendes vorausgesetzt: $m, n \in \mathbb{N}$, mit $m < n$, $c, l, x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ und $A \in \mathbb{R}^{m \times n}$ hat vollen Rang. Betrachte das LP

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & Ax & = & b \\ & x & \geq & 0. \end{array} \tag{1.28}$$

Dieses LP ist offenbar der (umgeschriebene) Spezialfall eines LPs der Form (1.1), mit $k = 2m + n$, $D = (I, A^T, -A^T)^T$ und $d = (0^T, b^T, -b^T)^T$, nämlich:

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & Ix & \geq & 0 \\ & Ax & \geq & b \\ & -Ax & \geq & -b. \end{array} \tag{1.29}$$

Für jede zulässige Zeilenbasis von (1.29) müssen die Ungleichungen $\{n+1, \ldots, n+2m\}$ mit Gleichheit erfüllt sein. Da A vollen Rang hat, kann man zu jeder zulässigen Basislösung x eine zulässige Zeilenbasis $Z' = (P', Q')$ von (1.29) finden, bei der alle Ungleichungen $Ax \geq b$ zur Basis gehören, d.h. $P' \supseteq \{n+1, \ldots, n+m\}$ und $x = x(Z')$.

SATZ 13

Sei $Z = (P, Q)$ eine zulässige Zeilenbasis von (1.29), mit Basislösungsvektor $x(Z)$. Dann gibt es eine zulässige Zeilenbasis $\bar{Z} = (\bar{P}, \bar{Q})$ von (1.29), mit $x(\bar{Z}) = x(Z)$ und $\bar{P} \supseteq \{n+1, \ldots, n+m\}$.

BEWEIS:

Sei $Z = (P, Q)$ eine zuläsige Basis von (1.29) mit Lösungsvektor $x(Z)$. Setze $M = \{n+1, \ldots, n+m\}$, $R(P) = P \cap M$ und $S(P) = M \setminus R(P)$, d.h. $S(P)$ bezeichnet die Menge aller Nichtbasisungleichungen von $Ax \geq b$. Gilt $|S(P)| = 0$, so erfüllt Z bereits das Gewünschte. Andernfalls konstruieren wir eine Basis $Z' = (P', Q')$ mit $x(Z') = x(Z)$, für die $|S(P')| = |S(P)| - 1$ gilt. Durch $|S(P)|$-fache Anwendung dieser Konstruktion erhält man somit eine Basis $\bar{Z}$, die das Gewünschte leistet.

Betrachten wir für ein $k \in S(P)$ den Vektor $x^T = A_{k.} D_{P.}^{-1}$. Wir wählen ein $j \in \{1, \ldots, n\}$, so daß $P_j \notin R(P)$ und $x_j \neq 0$. Ein solches muß es geben, denn sonst hätte wegen $A_{k.} = \sum_{P_i \notin R(P)} x_i A_{i.}$ die Matrix A nicht vollen Rang. Nach Satz 6 ist $Z' = (P', Q')$ mit $P' = P \setminus \{P_j\} \cup \{k\}$ und $Q' = Q \setminus \{k\} \cup \{P_j\}$ eine Zeilenbasis von (1.29) und wegen $s_k(Z) = b_k$ gilt $\Phi(j, k) = 0$. Somit ist nach (1.14) $x(Z) = x(Z')$ und Z' zulässig. Offenbar gilt $|S(P')| = |S(P)| - 1$, was den Beweis beendet. ■

Dieser Satz erlaubt es, sich bei der Suche nach einer optimalen und zulässigen Basis von (1.29) auf solche Basen $Z = (P, Q)$ zu beschränken, für die $P \supseteq M$ gilt. Die Zeilen $\{n+1, \ldots, n+m\}$ können somit stets in der Basis verbleiben, während die Ungleichungen $\{n+m+1, \ldots, n+2m\}$ nie in die Basis gelangen (sie würde sonst singulär) und daher ignoriert werden können. Beides kann man durch eine angepaßte Verwaltung der Indexvektoren modellieren, bei der nur noch Zeilen $\{1, \ldots, n\}$ ausgetauscht werden. Um dies effizient durchführen zu können, eignet sich die Definition einer Spaltenbasis.

DEFINITION 5 (SPALTENBASIS)

Ein geordnetes Paar $S = (B, N)$ von Indexvektoren $B, N \subseteq \{1, \ldots, n\}$ heißt Spaltenbasis *(zum LP (1.28)), falls folgendes gilt:*

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

Die Indizes aus B heißen Basisindizes *und die Indizes aus N* Nichtbasisindizes. *Die Zeilenbasis $Z = (P, Q)$ zu (1.29), mit $P = N \cup \{n+1, \ldots, n+m\}$ heißt die* zu S gehörende *Zeilenbasis. Die Vektoren $x(S) = x(Z)$, $s(S) = s(Z)$ und $y(S) = y(Z)$ heißen der* Basislösungsvektor*, der* Vektor der Schlupfvariablen *respektive der* duale Vektor *von S. Eine Spaltenbasis S heißt* zulässig*, wenn $x(S) \in \mathcal{P}^=(A, b)$, und* optimal*, falls $c^T x \geq c^T x(S)$ für alle $x \in \mathcal{P}^=(A, b)$.*

Die Definition einer Spaltenbasis zerlegt also die Variablen in Basis- und Nichtbasisvariablen. Die Variablenschranken der Nichtbasisvariablen bilden zusammen mit den Ungleichungen $Ax \geq b$ die Basisungleichungen der zugehörigen Zeilenbasis zu (1.29). Dabei ist a priori nicht kar, ob Z wirklich eine Zeilenbasis ist, sofern S eine Spaltenbasis ist. Es bedarf daher des folgenden Satzes.

SATZ 14

Die zu einer Spaltenbasis gehörende Zeilenbasis ist wohldefiniert.

BEWEIS:

Es muß gezeigt werden, daß $A_{.B}$ genau dann regulär ist, wenn es $D_{P.}$ ist. Nach geeigneter Permutation hat die Zeilenbasismatrix folgende Gestalt:

$$D_{P.} = \begin{pmatrix} 0 & I_{|N|} \\ A_{.B} & A_{.N} \end{pmatrix}. \tag{1.30}$$

Also gilt nach dem Determinantenentwicklungssatz

$$|\det(D_{P.})| = |\det(A_{.B})|,$$

woraus die Behauptung folgt.

■

Die Struktur der Zeilenbasismatrix (1.30) läßt sich auch für die Berechnung der Vektoren $x(S)$, $s(S)$ und $y(S)$ ausnutzen:

$$\begin{aligned} x_B(S) &= A_{.B}^{-1} b, \text{ mit } x_N(S) = 0_N && (1.31)\\ s^T(S) &= (x^T(S), b^T, -b^T) && (1.32)\\ y_B^T(S) &= c_B^T A_{.B}^{-1} \text{ und } y_N^T(S) = c_N^T - y_B^T(S) A_{.N} && (1.33) \end{aligned}$$

Diese Struktur der Vektoren erlaubt es, die Optimalität und Zulässigkeit einer Spaltenbasis mit einem geringeren Rechenaufwand zu überprüfen als mit der zugehörigen Zeilenbasis.

SATZ 15

Sei S eine Spaltenbasis und

$$t^T(S) = y_B^T(S) A. \tag{1.34}$$

S ist zulässig, wenn

$$x_B(S) \geq 0, \tag{1.35}$$

und S ist optimal, wenn

$$t(S) \leq c. \tag{1.36}$$

Beweis:
Sei Z die zu S gehörende Zeilenbasis. Die Zulässigkeitsbedingungen (1.35) und (1.20) sind äquivalent, weil nach (1.31) und (1.32) $s^T(Z) =_{perm} (x_N^T(S), x_B^T(S), b, -b) = (0_N^T, x_B^T(S), b, -b)$ gilt. Dabei bezeichnet $=_{perm}$ Gleichheit bei geeigneter Permutation.

Für die Optimalitätsbedingung betrachten wir $y^T(Z) = c^T D_{P.}^{-1} =_{perm} (y_B^T(S), c_N^T - y_B^T(S)A_{.N})$. Wenn Z optimal also $y(Z) \geq 0$ ist, gilt nach Gleichung (1.33) und (1.34) $c_N^T - t_N^T(S) = c_N^T - y_B^T(S)A_{.N} = y_N^T(Z) \geq 0$ und $t_B^T(S) = c_B^T A_{.B}^{-1} A_{.B} = c_B^T$, woraus Gleichung (1.36) folgt.

Gilt nun (1.36) aber $y(Z) \ngeq 0$, so konstruieren wir eine optimale Zeilenbasis Z' mit demselben Lösungsvektor $x(Z) = x(Z')$. Zunächst gilt nach Voraussetzung (1.36) und wegen der Gleichungen (1.33) und (1.34) $y(Z)_N \geq 0$. Setze $Z' = (P', Q')$, mit $P' = P \setminus \{i + n : y(Z)_i < 0\} \cup \{i + 2n : y(Z)_i < 0\}$. Z' erhält man also aus Z, indem man alle Zeilen mit negativen dualen Variablen durch die entsprechenden Zeilen aus $-Ax \geq -b$ ersetzt. Demnach ist $D_{P'.} = \mathrm{diag}\,(\sigma) \cdot D_{P.}$ und $d_{P'} = \mathrm{diag}\,(\sigma) \cdot d_P$, wobei $\sigma_i = -1$ für $y(Z)_i < 0$ und sonst $\sigma_i = +1$ gilt. Z' ist also eine Basis mit Lösungsvektor $x(Z') = D_{P'.}^{-1} d_{P'} = D_{P.}^{-1} \cdot \mathrm{diag}\,(\sigma)^{-1} \cdot \mathrm{diag}\,(\sigma) d_P = x(Z)$, und für den dualen Vektor gilt $y^T(Z') = c^T D_{P.}^{-1} \mathrm{diag}\,(\sigma)^{-1} = y^T(Z) \cdot \mathrm{diag}\,(\sigma) \geq 0$ nach der Konstruktion von σ.

■

Es liegt somit eine Asymmetrie zwischen einer Spaltenbasis S und der zugehörigen Zeilenbasis Z vor. Mit Z ist stets auch S optimal; die Umkehrung gilt jedoch nicht. Der obige Beweis legt aber nahe, wie diese Asymmetrie aufgehoben werden kann. Dazu wird in Abschnitt 1.2.4 die Definition einer Zeilenbasis auch auf Gleichungen erweitert. Wie soeben gezeigt, unterliegen die zu Gleichungen gehörenden dualen Variablen keinen Beschränkungen.

Eine Spaltenbasismatrix besteht aus Spalten von A. Bei jedem Pivot-Schritt wird eine Spalte ausgetauscht. Dementsprechend müssen auch die Vektoren $x_B(S)$ und $t(S)$ aktualisiert werden. Wie dies zu geschehen hat, beschreibt der folgende Satz.

Satz 16 (Spaltentausch)
Sei $S = (B, N)$ eine Spaltenbasis von (1.28), $j \in \{1, \ldots, m\}$ und $i \in N$. Es ist $S' = (B', N')$, mit $B' = B \setminus \{B_j\} \cup \{i\}$ und $N' = N \setminus \{i\} \cup \{B_j\}$, genau dann eine Spaltenbasis von (1.28), wenn

$$e_j^T A_{.B}^{-1} A_{.i} \neq 0. \tag{1.37}$$

Ferner sind

$$\begin{aligned} y_{B'}^T(S') &= y_B^T(S) + \Theta(i,j) \cdot e_j^T A_{.B}^{-1}, && (1.38)\\ t^T(S') &= t^T(S) + \Theta(i,j) \cdot e_j^T A_{.B}^{-1} A, && (1.39)\\ x_{B'}(S') &= x_B(S) + \Phi(i,j) \cdot (A_{.B}^{-1} A_{.i} - e_i) && (1.40) \end{aligned}$$

wobei

$$\Theta(i,j) = \frac{c_i - t(S)_i}{e_j^T A_{.B}^{-1} A_{.i}} \tag{1.41}$$

und

$$\Phi(i,j) = \frac{-x_{B_j}(S)}{e_j^T A_{.B}^{-1} A_{.i}} \tag{1.42}$$

gilt.

Beweis:
Bis auf (1.39) folgen alle Aussagen aus Satz 5, wenn man $D = (A_{.B}^T)$, $l = j$, $r = A_{.i}$, $\rho = c_i$, $f = y_B(S)$, $f' = y_{B'}(S')$, $\bar{f} = c_B$, $\theta = \Theta(i,j)$, $g = x_B(S)$, $g' = x_{B'}(S')$, $\bar{g} = b$ und $\phi = \Phi(i,j)$ setzt. Gleichung (1.39) folgt damit nach Defintion (1.34), denn es ist $t^T(S') = y_{B'}^T(S')A = y_B^T(S)A + \Theta(i,j) \cdot e_j^T A_{.B}^{-1} A = t^T(S) + \Theta(i,j) \cdot e_j^T A_{.B}^{-1} A$.

■

1.2.2.1 Primaler Algorithmus

Nunmehr sind die Vorbereitungen für die Formulierung der Simplex-Algorithmen für Spaltenbasen abgeschlossen. Beim primalen Algorithmus wird die Zulässigkeit von S verlangt, d.h. $x_B(S) \geq 0$ muß gelten. Solange die Optimalitätsbedingung nicht gilt, wird ein Vektor in die Basis hineingetauscht.

Algorithmus 3 (Primaler Simplex-Algorithmus in Spaltendarstellung)

Sei $S = (B, N)$ eine zulässige Spaltenbasis von (1.28).

Schritt 0 (Initialisierung):

$$\begin{array}{lcll} h^T & \leftarrow & c_B^T A_{.B}^{-1} & = & y^T(S) \\ g^T & \leftarrow & h^T \cdot A & = & t^T(S) \\ f & \leftarrow & A_{.B}^{-1} b & = & x_B(S) \end{array}$$

Schritt 1 (Pricing): Falls $g \leq c$ terminiere:
x, mit $x_B = f^T$ und $x_N = 0$ ist optimaler Lösungsvektor des LPs.
Sonst wähle q, mit $g_q > c_q$.

Schritt 2: $\Delta f \leftarrow A_{.B}^{-1} A_{.q}$

Schritt 3 (Quotiententest):
Falls $\Delta f \leq 0$ terminiere: Das LP ist unbeschränkt;
sonst wähle $p \in \arg\max\{-f_i/\Delta f_i : \Delta f_i > 0\}$

Schritt 4: $\Delta h^T \leftarrow e_p^T A_{.B}^{-1}$
$\Delta g^T \leftarrow \Delta h^T \cdot A$

Schritt 5 (Update):

$$\begin{array}{lcl} B_p & \leftarrow & q \\ \Theta & \leftarrow & (c_p - g_p)/\Delta g_p \\ \Phi & \leftarrow & -f_q/\Delta f_q \\ h & \leftarrow & h' = h + \Theta \cdot \Delta h \\ g & \leftarrow & g' = g + \Theta \cdot \Delta g \\ f & \leftarrow & f' = f + \Phi \cdot (\Delta f - e_p) \end{array}$$

Schritt 6: Gehe zu Schritt 1

Satz 17 (Partielle Korrektheit von Algorithmus 3)
Algorithmus 3 arbeitet partiell korrekt.

Beweis:
Bei Terminierung in Schritt 1 folgt die Behauptung aus Satz 15. Wegen Satz 16 ist in Schritt 6 B wieder eine Basis, die wegen Schritt 3 auch zulässig ist.

Für die Unbeschränktheit des LPs bei Terminierung in Schritt 3 betrachte das LP (1.29) und die zu S gehörende Zeilenbasis $Z = (P, Q)$. Wir identifizieren nun die Größen aus Algorithmus 3 mit denen aus Satz 9. Nach (1.33) gilt $y_N(Z) = c_N - t_N(S)$, und somit ist $y_q(Z) = c_q - g_q < 0$. Ferner gilt nach (1.30) für den Vektor $t = D_{P,\cdot}^{-1} e_q$: $t_N = e_q$ und $t_B = -A_{\cdot B}^{-1} A_{\cdot q} = -\Delta f$. Deshalb ist

$$D \cdot t = \begin{pmatrix} t \\ At \\ -At \end{pmatrix} = \begin{pmatrix} t \\ A_{\cdot B}(-A_{\cdot B}^{-1} A_{\cdot q}) + A_{\cdot N} e_q \\ -A_{\cdot B}(-A_{\cdot B}^{-1} A_{\cdot q}) - A_{\cdot N} e_q \end{pmatrix} = \begin{pmatrix} t \\ 0 \\ 0 \end{pmatrix}.$$

Gilt also in Schritt 3 $\Delta f \leq 0$, so folgt $Dt \geq 0$ und damit nach Satz 9 die Unbeschränktheit des LPs.

■

Algorithmus 3 ist *mathematisch* eine Umformulierung von Algorithmus 1 in die Notation einer Spaltenbasis. *Algorithmisch* entspricht Algorithmus 3 dem dualen Algorithmus 2: Bei entsprechender Initialisierung aller relevanten Vektoren und Schranken unterscheidet sich lediglich das Terminationskriterium in Schritt 1 gemäß der umgekehrten Ungleichungsrichtung in (1.36).

1.2.2.2 Dualer Algorithmus

Wie zuvor der primale Algorithmus wird nun auch der duale für eine Spaltenbasis umformuliert.

Algorithmus 4 (Dualer Simplex-Algorithmus in Spaltendarstellung)

Sei $S = (B, N)$ eine optimale Zeilenbasis von (1.28).

Schritt 0 (Initialisierung):

$$\begin{array}{lcll} h^T & \leftarrow & c_B^T A_{.B}^{-1} & = & y^T(S) \\ g^T & \leftarrow & h^T \cdot A & = & t^T(S) \\ f & \leftarrow & A_{.B}^{-1} b & = & x_B(S) \end{array}$$

Schritt 1 (Pricing):
Falls $f \geq 0$ terminiere:
x, mit $x_B = f^T$ und $x_N = 0$ ist optimaler Lösungsvektor des LPs.
Sonst wähle p, mit $f_p < 0$.

Schritt 2: $\Delta h^T \leftarrow e_p^T A_{.B}^{-1}$
$\Delta g^T \leftarrow \Delta h^T \cdot A$

Schritt 3 (Quotiententest):
Falls $\Delta g \geq 0$ terminiere: Das LP ist unzulässig;
sonst wähle $q \in \arg\max\{(c_i - g_i)/\Delta g_i : \Delta g_i < 0\}$

Schritt 4: $\Delta f \leftarrow A_{.B}^{-1} A_{.q}$

Schritt 5 (Update):

$$\begin{array}{lcl} B_p & \leftarrow & q \\ \Theta & \leftarrow & (c_p - g_p)/\Delta g_p \\ \Phi & \leftarrow & -f_q/\Delta f_q \\ h & \leftarrow & h' = h + \Theta \cdot \Delta h \\ g & \leftarrow & g' = g + \Theta \cdot \Delta g \\ f & \leftarrow & f' = f + \Phi \cdot (\Delta f - e_p) \end{array}$$

Schritt 6: Gehe zu Schritt 1

Satz 18 (Partielle Korrektheit von Algorithmus 4)
Algorithmus 4 arbeitet partiell korrekt.

Beweis:
Bei Terminierung in Schritt 1 folgt die Behauptung aus Satz 15. Wegen Satz 16 ist in Schritt 6 B wieder eine Basis, die wegen Schritt 3 auch dual zulässig ist.

Für den Fall der Termination in Schritt 3 betrachte das LP

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & I_N x & \geq & 0 \\ & Ax & = & b. \end{array} \tag{1.43}$$

$\mathcal{P}$ bezeichne das zugehörige Polyeder, und es seien $D = \binom{I_N}{A}$ sowie $d = \binom{0}{b}$. Aus den Sätzen 2 und 3 folgt, daß $x(S)$ die einzige Ecke von $\mathcal{P}$ ist und alle $x \in \mathcal{P}$ als

$$x = x(S) + \sum_{i=1}^{n-m} \tau_i D^{-1} e_i,$$

mit $\tau_1, \ldots, \tau_{n-m} \geq 0$, dargestellt werden können. Für alle $x \in \mathcal{P}$ gilt deshalb $x_{B_p} = e_{B_p}^T x = x_{B_p}(S) + \sum_{i=1}^{n-m} \tau_i (e_{B_p}^T D^{-1})_i$. Für $1 \leq i \leq n-m$ ist $(e_{B_p}^T D^{-1})_i = -e_p^T A_{.B}^{-1} A_{.N_i} = -\Delta g_{N_i}^T \leq 0$, so daß $e_{B_p}^T x \leq x_{B_p}(S) < 0$ folgt. Nun muß aber $x_{B_q} \geq 0$ für alle $x \in \mathcal{P}^=(A,b)$ gelten, so daß wegen $\mathcal{P} \supseteq \mathcal{P}^=(A,b)$ die Unzulässigkeit $\mathcal{P}^=(A,b) = \emptyset$ bei Termination in Schritt 3 folgt. ■

Wiederum sei auf die algorithmische Ähnlichkeit des dualen Algorithmus' mit Spaltenbasis 4 zum primalen mit Zeilenbasis 1 hingewiesen. Bis auf die unterschiedliche Initialisierung der Vektoren und ihrer Schranken liegt der einzige Unterschied in Schritt 3. Dieser ist wieder auf die Ungleichungsrichtung (1.36) zurückzuführen.

Somit vertauscht sich beim Übergang von der zeilenweisen zur spaltenweisen Darstellung der Basis der Typ der Algorithmen, wie es die folgende Tabelle angibt.

	Spaltenbasis	Zeilenbasis
einfügend	primal	dual
entfernend	dual	primal

Tabelle 1.1: Zusammenhang zwischen der Basisdarstellung und algorithmischem und mathematischem Typ von Simplex-Algorithmen

Dies liegt am Zusammenhang der Algorithmen mit den entsprechenden Algorithmen auf dem *dualen LP*, der im folgenden Abschnitt beschrieben wird.

1.2.3 Dualität

In Abschnitt 1.2.2 wurde eine Ähnlichkeit der Simplex-Algorithmen in Zeilen- und Spaltendarstellung der Basis aufgezeigt. Diese ist in der nun zu erörtenden Dualität begründet. Dazu gehen wir zurück auf das LP

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Dx & \geq \quad d. \end{array} \tag{1.1}$$

Multipliziert man jede Ungleichung $i \in \{1, \ldots, k\}$ mit einer nichtnegativen Zahl $y_i \geq 0$, so daß $y^T D = c^T$ ergibt, erhält man eine untere Schranke für den optimalen Zielfunktionswert des LPs, nämlich $c^T x = y^T D x \geq y^T d$. Dies gilt für alle $y \geq 0$ mit $y^T D = c^T$, also insbesondere auch für die bestmögliche Schranke, die man als Lösung des sog. *dualen LPs* erhält:

$$\begin{array}{lrcl} \max & d^T y & & \\ \text{s.t.} & D^T y & = & c \\ & y & \geq & 0. \end{array} \tag{1.44}$$

Dieses LP hat die geeignete Form, um eine Spaltenbasis $S = (B, N)$ zu definieren. Wenn $(D^T)_{.B}^{-1} c \geq 0$ ist S zulässig (1.35), und wenn $d^T (D^T)_{.B}^{-1} D^T \leq d^T$ ist S optimal (1.36).

Bis auf Transposition sind das dieselben Beziehungen wie die Zulässigkeits- und Optimalitätsbedinungen für die Zeilenbasis $Z = (B, N)$ zum LP (1.1). Eine Zeilenbasis ist somit äquivalent zu einer Spaltenbasis auf dem dualen LP. Diese Äquivalenz verwenden wir zum Beweis des folgenden zentralen Satzes der Linearen Programmierung, wobei vorausgesetzt wird, daß es Simplex-Algorithmen gibt, die vollständig korrekt arbeiten (vgl. Abschnitt 1.5).

Satz 19 (Dualitätssatz der Linearen Programmierung)
Sei $n, k \in \mathbb{N}$, $0 < n < k$, $c, x \in \mathbb{R}^n$, $d \in \mathbb{R}^k$ und $D \in \mathbb{R}^{k \times n}$ habe vollen Rang. Das Lineare Programm

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Dx & \geq \; d. \end{array} \tag{1.1}$$

hat genau dann eine optimale Lösung, wenn sein duales LP

$$\begin{array}{lll} \max & d^T y & \\ \text{s.t.} & D^T y & = \; c \\ & y & \geq \; 0. \end{array} \tag{1.44}$$

eine optimale Lösung hat. In diesem Fall stimmen beide Zielfunktionswerte überein.

Beweis:
$Z = (B, N)$ ist genau dann eine optimale, zulässige Zeilenbasis von (1.1), wenn $S = (B, N)$ eine optimale und zulässige Spaltenbasis von (1.44) ist. In diesem Fall ist $c^T x(Z) = c^T D_{B.}^{-1} d_B = d_B^T (D^T)_{.B}^{-1} c = d_B^T (D^T)_{.B}^{-1} c + d_N^T 0_N = d^T y(S)$.
■

Dieser Satz erklärt die in Tabelle 1.1 zusammengestellten Ähnlichkeiten zwischen beiden Simplex-Algorithmen und beiden Basisdarstellungen. Der Vorzeichenwechsel bei dem Optimalitätskriterium rührt von der unterschiedlichen Optimierungsrichtung von (1.44) und (1.28).

1.2.4 Allgemeine Basis

Die Beziehung zwischen beiden Simplex-Algorithmen mit Zeilen- und Spaltenbasis über die Dualität läßt die Frage aufkommen, ob zwei verschiedene Darstellungen sinnvoll sind. Mathematisch gesehen, ist dieser Einwand gerechtfertigt, kann man doch, anstelle eine Zeilenbasisbasis zu verwenden, zum dualen LP übergehen und eine Spaltenbasis benutzen. Vom Standpunkt der Implementierung für tatsächlich auftretende LPs ergibt sich jedoch ein anderes Bild.

Oft trifft man auf LPs, bei denen obere und untere Schranken sowohl für die Variablen als auch für „echte“ Bedingungen, sog. *Bereichsungleichungen*, vorliegen. Betrachte also das LP

$$\begin{array}{lccccc} \min & & & c^T x & & \\ \text{s.t.} & l' & \leq & x & \leq & u', \\ & l'' & \leq & A'x & \leq & u'', \end{array} \tag{1.45}$$

wobei $A' \in \mathbb{R}^{m \times n}$, alle Vektoren dimensionskompatibel seien und $l' \leq u'$ sowie $l'' \leq u''$ gelte. Dabei seien für l' und l'' auch Werte $-\infty$ sowie für u' und u'' auch Werte $+\infty$ zulässig. Um das zu diesem LP duale anzugeben, müssen die Bereichsungleichungen $l_i'' \leq A_i' x \leq u_i''$ in ein Paar von Ungleichungen aufgespalten werden. Dann ergibt sich das folgende duale LP

$$\begin{array}{lccccccccc} \max & (l')^T s & - & (u')^T t & + & (l'')^T y & - & (u'')^T z & & \\ \text{s.t.} & s & - & t & + & A'^T y & - & A'^T z & = & c \\ & s & , & t & , & y & , & z & \geq & 0. \end{array} \tag{1.46}$$

Die Bedingungsmatrix dieses LPs ist doppelt so groß wie die des Ausgangs-LPs (1.45) und damit auch der Rechenaufwand in jeder Simplex-Iteration. Dies kann jedoch vermieden werden, indem man anstelle einer Spaltenbasis auf dem dualen LP (1.46) eine (geeignet erweiterte) Zeilenbasis auf dem primalen LP (1.45) benutzt.

Im folgenden geht es darum, eine Zeilen- und Spaltenbasis für LPs der Form (1.45) zu definieren und dafür die Simplex-Algorithmen zu formulieren. Dazu werden Bezeichnungen eingeführt, die es ermöglichen, sowohl den einfügenden als auch den entfernenden Algorithmus weitgehend unabhängig von der Basisdarstellung zu formulieren. Dadurch „zählt“ jeder Algorithmus gemäß Tabelle 1.1 für zwei.

Eine Zeilenbasis kann direkt für ein LP der Form (1.45) definiert werden. Zur Vereinfachung der Notation hilft es, dieses in

$$\begin{array}{lccccc} \min & & & c^T x & & \\ \text{s.t.} & l^Z & \leq & A^Z x & \leq & u^Z \end{array} \tag{1.47}$$

umzuschreiben, wobei $l^Z = \binom{l'}{l''}$, $u^Z = \binom{u'}{u''}$ und $A^Z = \binom{I}{A'}$ gesetzt wurde. Man nennt (1.47) das *zu (1.45) gehörende Zeilen-LP*.

Für die Definition einer Spaltenbasis zum LP (1.45) muß es in das LP

$$\begin{array}{lccccccc} \min & & & c^T x & & & & \\ \text{s.t.} & & & A'x & - & s & = & 0, \\ & l' & \leq & x & & & \leq & u', \\ & l'' & \leq & & & s & \leq & u''. \end{array} \tag{1.48}$$

durch Einführung der Schlupfvariablen $s = A'x$ transformiert werden. Hier zeigt sich der Vorteil dieser Definition gegenüber der in der Literatur üblichen, die sich nicht auf so symmetrische Weise auf Bereichsungleichungen erweitern läßt. Das LP (1.48) ist in folgendem Sinn äquivalent zum Ausgangs-LP (1.45).

SATZ 20

Für jede Lösung $\binom{x}{s}$ von (1.48) ist x eine Lösung von (1.45) mit demselben Zielfunktionswert. Entsprechend ist für jede Lösung x von (1.45) $\binom{x}{s}$, mit $s = A'x$, eine Lösung von (1.48) mit demselben Zielfunktionswert.

BEWEIS:
Sei $\binom{x}{s}$ eine Lösung von (1.48). Dann ist x auch Lösung von (1.45), denn $l' \leq x \leq u'$ und $l'' \leq s = A'x \leq u''$. Sei nun x eine Lösung von (1.45), dann gilt für $\binom{x}{s}$, mit $s = A'x$, offenbar $l'' \leq s = A'x \leq u''$. Außerdem gilt wieder $l' \leq x \leq u'$, d.h. $\binom{x}{s}$ ist Lösung von (1.48). Ferner gilt jeweils für den Zielfunktionswert $c^T x = (c^T, 0^T)\binom{x}{s}$.

■

Insbesondere folgt aus diesem Satz, daß zur Lösung von (1.45) ersatzweise auch das LP (1.48) gelöst werden kann.

Wie für eine Zeilenbasis wird zur Verkürzung der Notation (1.48) umgeschrieben zu

$$\begin{array}{lrcl} \min & c^{S^T} x^S & & \\ \text{s.t.} & A^S x^S & = & 0, \\ L^S \leq & x^S & \leq & U^S, \end{array} \tag{1.49}$$

wobei $c^{S^T} = (c^T, 0^T)$, $A^S = (A', -I)$, $L^S = \binom{l'}{l''} = l^Z$ und $U^S = \binom{u'}{u''} = u^Z$. Das LP (1.49) heißt das *zu (1.45) gehörende Spalten-LP.*

DEFINITION 6 (ALLGEMEINE SIMPLEX-BASIS)

Seien $P_f, P_l, P_u, P_x, Q_f, Q_l, Q_u, Q_x$ Indexvektoren. Eine geordnete Menge $\mathcal{B} = (P, Q)$, mit $P = (P_f, P_l, P_u, P_x)$ und $Q = (Q_f, Q_l, Q_u, Q_x)$ heißt Basis *von (1.45), falls folgendes gilt:*

1. $P \cup Q = \{1, \ldots, n+m\}$,
2. *P_f, P_l, P_u, P_x, Q_f, Q_l, Q_u und Q_x sind paarweise disjunkt,*
3. $|P| = n$,
4. *$A^Z_{P.}$ ist nicht singulär und*
5. $$\begin{array}{rcrcl} i \in P_f \cup Q_f & \Rightarrow & -\infty = l_i^Z & \wedge & u_i^Z = \infty \\ i \in P_l \cup Q_l & \Rightarrow & -\infty < l_i^Z & \neq & u_i^Z \\ i \in P_u \cup Q_u & \Rightarrow & l_i^Z & \neq & u_i^Z < \infty \\ i \in P_x \cup Q_x & \Rightarrow & -\infty < l_i^Z & = & u_i^Z < \infty. \end{array}$$

$A^Z_{P.}$ heißt die Zeilenbasismatrix *(oder* Basismatrix in Zeilendarstellung*) und $A^S_{.Q}$ die* Spaltenbasismatrix *(oder* Basismatrix in Spaltendarstellung*).*

Die Matrix A^Z hat $n+m$ Zeilen. Somit entsprechen die Bedingungen 1-4 denen aus Definition 2. Die Matrix A^S hat hingegen $n+m$ Spalten, und man erwartet, daß die

Bedingungen 1-4 auch denen aus Definition 5 entsprechen. Dies ist der Inhalt von folgendem Satz.

SATZ 21
Seien $P \subseteq \{1,\ldots,m+n\}$ und $Q = \{1,\ldots,m+n\} \setminus P$ Indexvektoren mit $|P| = n$. Die Matrix $A^Z_{P.}$ ist genau dann regulär, wenn es $A^S_{.Q}$ ist.

BEWEIS:
Setze $P_1 = P \cap \{1,\ldots,n\}$, $P_2 = P \setminus P_1$, $Q_1 = Q \cap \{1,\ldots,n\}$ und $Q_2 = Q \setminus Q_1$. Damit ist $P_1 \cup Q_1 = \{1,\ldots,n\}$ und $P_2 \cup Q_2 = \{n+1,\ldots,m+n\}$. Für eine Menge von Indizes I bezeichne $I - n = \{i - n : i \in I\}$. Nach geeigneter Permutation hat die Zeilenbasismatrix die Gestalt

$$A^Z_{P.} = \begin{pmatrix} I_{|P_1|} & 0 \\ A'_{P_2-nP_1} & A'_{P_2-nQ_1} \end{pmatrix} \tag{1.50}$$

während die Spaltenbasismatrix wie folgt aussieht:

$$A^S_{.Q} = \begin{pmatrix} A'_{Q_2-nQ_1} & -I_{|Q_2|} \\ A'_{P_2-nQ_1} & 0 \end{pmatrix} \tag{1.51}$$

Nach dem Determinantenproduktsatz gilt

$$|\det(A^S_{.Q})| = |\det(A'_{P_2-nQ_1})| = |\det(A^Z_{P.})|.$$

Da eine Matrix genau dann singulär ist, wenn ihre Determinante 0 ist, folgt die Behauptung.

■

Somit definiert eine allgemeine Basis sowohl eine Zeilenbasis auf dem zugehörigen Zeilen-LP als auch eine Spaltenbasis auf dem zugehörigen Spalten-LP. Die Basisbedingung 5 dient nun dazu, die zu einer Basis gehörenden Vektoren zu definieren. Wir tun dies zunächst für eine Basis in Zeilendarstellung für das Zeilen-LP (1.47). Die Indizes aus P werden wie gewohnt zur Bestimmung eines Basislösungsvektors herangezogen. Nun definiert eine Bereichsungleichung $l \leq a^T x \leq u$ im allgemeinen zwei parallele Seitenflächen des LP-Polyeders, von denen aber nur eine zur Definition des Lösungsvektors herangezogen werden kann. Deshalb wird P in die vier Teilmengen unterteilt, nämlich

- P_x wenn die Bereichsungleichung wegen $l = u$ eine Gleichung beschreibt, die direkt verwendet werden kann (x steht für fi**x**ed),
- P_l wenn die Seitenfläche zu $l \leq a^T x$ verwendet werden soll (l steht für **l**ower),
- P_u wenn die Seitenfläche zu $a^T x \leq u$ verwendet werden soll (u steht für **u**pper) und

P_f wenn wegen $l = -\infty$ und $u = +\infty$ keine Seitenfläche definiert wird (f steht für **f**ree). Stattdessen wird künstlich $a^T x = 0$ verwendet.

Dies wird in der folgenden Definition präzisiert, in der auch die Teilmengen Q_ξ, $\xi \in \{x, u, l, f\}$ für die dualen Variablen verwendet werden.

DEFINITION 7

Sei $\mathcal{B}$ eine Basis von (1.45) mit Zeilenbasismatrix $B = (A^Z)_{P.}$. Für $i \in \{1, \ldots, m+n\}$ setze

$$L_i^Z(\mathcal{B}) = \begin{cases} 0 & , \textit{falls } i \in P_f \cup Q_f \\ 0 & , \textit{falls } i \in P_l \cup Q_l \\ -\infty & , \textit{falls } i \in P_u \cup Q_u \\ -\infty & , \textit{falls } i \in P_x \cup Q_x \end{cases}, \tag{1.52}$$

$$U_i^Z(\mathcal{B}) = \begin{cases} 0 & , \textit{falls } i \in P_f \cup Q_f \\ \infty & , \textit{falls } i \in P_l \cup Q_l \\ 0 & , \textit{falls } i \in P_u \cup Q_u \\ \infty & , \textit{falls } i \in P_x \cup Q_x \end{cases} \tag{1.53}$$

und

$$R_i^Z(\mathcal{B}) = \begin{cases} 0 & , \textit{falls } i \in Q_x \\ U_i^Z(\mathcal{B}) & , \textit{falls } i \in Q_u \\ L_i^Z(\mathcal{B}) & , \textit{falls } i \in Q_l \\ L_i^Z(\mathcal{B}) & , \textit{falls } i \in Q_f \\ 0 & , \textit{sonst.} \end{cases} \tag{1.54}$$

Setze ferner für $i \in \{1, \ldots, n\}$

$$r_i^Z(\mathcal{B}) = \begin{cases} 0 & , \textit{falls } P_i \in P_f \\ u_{P_i}^Z & , \textit{falls } P_i \in P_u \\ l_{P_i}^Z & , \textit{falls } P_i \in P_l \\ l_{P_i}^Z & , \textit{falls } P_i \in P_x \end{cases}. \tag{1.55}$$

Dann heißt

$$y^T(\mathcal{B}) = \left(c^T - R^Z(\mathcal{B}) A^Z\right) B^{-1} \tag{1.56}$$

der Vektor der Dualvariablen*,*

$$x(\mathcal{B}) = B^{-1} r^Z(\mathcal{B}) \tag{1.57}$$

der Basislösungsvektor *und*

$$s(\mathcal{B}) = A^Z x(\mathcal{B}) \tag{1.58}$$

der Vektor der Schlupfvariablen *von $\mathcal{B}$.*

Offenbar ist $R^Z(\mathcal{B})$ ein Nullvektor, so daß $z^T(\mathcal{B}) = c^T B^{-1}$ gilt. Die Form (1.56) zeigt jedoch deutlich die Ähnlichkeit zwischen den Vektoren für eine Zeilen- und den noch zu definierenden Vektoren für eine Spaltenbasis. Außerdem wird in den Abschnitten 1.3, 1.4 und 1.5 gezeigt, daß es für reale Implementierungen sehr wohl nötig werden kann, $R^Z(\mathcal{B})$ von Null verschiedene Werte zuzuweisen. In diesem Fall muß $y(\mathcal{B})$ in der Tat gemäß (1.56) berechnet werden.

Es sollen nun das Zulässigkeits- und das Optimalitätskriterium für allgemeine LPs aufgestellt werden. Dabei ist die Zulässigkeit wiederum einfach: $x(\mathcal{B})$ ist genau dann zulässig, wenn $l^Z \leq s(\mathcal{B}) \leq u^Z$. Wegen (1.57) und (1.58) gilt $s_P(\mathcal{B}) = r^Z(\mathcal{B})$, so daß sich die Zulässigkeitsbedingung auf

$$l_Q^Z \leq s_Q(\mathcal{B}) \leq u_Q^Z. \tag{1.59}$$

reduziert.

Das Optimalitätskriterium ist wiederum nicht unmittelbar einsichtig und wird deshalb in folgendem Satz formuliert.

Satz 22 (Allgemeines Simplex-Kriterium)
Sei $\mathcal{B}$ eine Basis von (1.45). Der Basislösungsvektor $x(\mathcal{B})$ ist optimal, wenn

$$L_P^Z(\mathcal{B}) \leq y(\mathcal{B}) \leq U_P^Z(\mathcal{B}) \tag{1.60}$$

gilt.

Beweis:
Es gelte (1.60). Sei $B = A_{P,}^Z$, $P_+ = \{i \in P_x : y_i(\mathcal{B}) \geq 0\}$ und $P_- = P_x \setminus P_+$. Für alle $F \subseteq P_f$ setze

$$\begin{aligned} \mathcal{P}_F = \{x \in \mathbb{R}^n \quad : \quad & B_{P_l} x \geq l_{P_l}^Z,\ -B_{P_u} x \geq -u_{P_u}^Z, \\ & B_{P_+} x \geq l_{P_+}^Z,\ -B_{P_-} x \geq -u_{P_-}^Z, \\ & B_F x \geq l_F^Z,\ -B_{P_f \setminus F}^Z x \geq -u_{P_f \setminus F}^Z\}. \end{aligned}$$

Nach Satz 7 und Definition 7 gilt für alle $F \subseteq P_f$ die Optimalität von $x(\mathcal{B})$ über $\mathcal{P}_F$, d.h. $c^T x(\mathcal{B}) \leq c^T x$ für alle $x \in \mathcal{P}_F$. Nun ist aber

$$\bigcup_{F \subseteq P_f} \mathcal{P}_F \supseteq \mathcal{P}(l^Z, A^Z, u^Z),$$

woraus die Optimalität auch für $\mathcal{P}(l^Z, A^Z, u^Z)$ folgt.

■

Eine Basis, für die (1.59) gilt, heißt *zulässig* und, falls (1.60) gilt, *optimal*.

Es werden nun das Optimalitäts- und Zulässigkeitskriterium für eine Spaltenbasis aufgestellt.

DEFINITION 8
Sei $\mathcal{B}$ eine Basis von (1.45) mit Spaltenbasismatrix $B = A^S_{\cdot Q}$. Für $i \in \{1, \ldots, m+n\}$ setze

$$R_i^S(\mathcal{B}) = \begin{cases} 0 & , \textit{falls } i \in P_f \\ U_i^S & , \textit{falls } i \in P_u \\ L_i^S & , \textit{falls } i \in P_l \\ L_i^S & , \textit{falls } i \in P_x \\ 0 & , \textit{sonst} \end{cases}, \tag{1.61}$$

und

$$l_i^S(\mathcal{B}) = \begin{cases} c^S & , \textit{falls } i \in P_f \\ c^S & , \textit{falls } i \in P_u \\ -\infty & , \textit{falls } i \in P_l \\ -\infty & , \textit{falls } i \in P_x \\ c^S & , \textit{sonst} \end{cases} \tag{1.62}$$

$$u_i^S(\mathcal{B}) = \begin{cases} c^S & , \textit{falls } i \in P_f \\ \infty & , \textit{falls } i \in P_u \\ c^S & , \textit{falls } i \in P_l \\ \infty & , \textit{falls } i \in P_x \\ c^S & , \textit{sonst.} \end{cases} \tag{1.63}$$

Setze ferner für $i \in \{1, \ldots, m\}$

$$r_i^S(\mathcal{B}) = \begin{cases} l^S_{Q_i}(\mathcal{B}) & , \textit{falls } Q_i \in Q_x \\ l^S_{Q_i}(\mathcal{B}) & , \textit{falls } Q_i \in Q_l \\ u^S_{Q_i}(\mathcal{B}) & , \textit{falls } Q_i \in Q_u \\ c^S & , \textit{falls } Q_i \in Q_f. \end{cases} \tag{1.64}$$

Definiere schließlich die Vektoren

$$t^T(\mathcal{B}) = {r^S}^T(\mathcal{B})B^{-1}, \tag{1.65}$$

$$d^T(\mathcal{B}) = t^T(\mathcal{B})A^S \tag{1.66}$$

und

$$s_Q(\mathcal{B}) = B^{-1}\left(0 - A^S R^S(\mathcal{B})\right). \tag{1.67}$$

Sofern keine Verwechslung möglich ist, wird im Folgenden die Abhängigkeit von der Basis ($\mathcal{B}$) nicht explizit notiert.

Es muß nun gezeigt werden, daß Definition (1.67) nicht im Widerspruch zur früheren Definition der Schlupfvariablen (1.58) steht.

SATZ 23
s_Q ist wohldefiniert.

BEWEIS:
Durch Multiplikation von (1.67) mit B erhält man $Bs_q = -A^S R^S = -A^S_{.P} R^S_P$, woraus durch Einsetzen von (1.51)

$$\begin{aligned} B \cdot s_Q &= \begin{pmatrix} A'_{Q_2-nQ_1} & -I_{|Q_2|} \\ A'_{P_2-nQ_1} & 0 \end{pmatrix} \cdot \begin{pmatrix} s_{Q_1} \\ s_{Q_2} \end{pmatrix} = \begin{pmatrix} A'_{Q_2-nQ_1} \cdot s_{Q_1} - s_{Q_2} \\ A'_{P_2-nQ_1} \cdot s_{Q_1} + 0 \end{pmatrix} \\ &= -\begin{pmatrix} (A^S_{.P} R^S_P)_{Q_2-n} \\ (A^S_{.P} R^S_P)_{P_2-n} \end{pmatrix} \end{aligned}$$

folgt. Für die rechte Seite gilt

$$A^S_{.P} R^S_P = A^S_{.P_1} R^S_{P_1} + A^S_{.P_2} R^S_{P_2} = A'_{.P_1} R^S_{P_1} - (e_{P_{2_1}-n}, \ldots, e_{P_{2_{|P_2|}}-n}) R^S_{P_2}.$$

Dann kann der $(P_2 - n)$-Teil geschrieben werden als $A^S_{P_2-nP} R^S_P = A'_{P_2-nP_1} R^S_{P_1} - R^S_{P_2}$, so daß

$$s_{Q_1} = (A'_{P_2-nQ_1})^{-1}(R^S_{P_2} - A'_{P_2-nP_1} R^S_{P_1}) \tag{1.68}$$

gilt. Entsprechend folgt aus $A^S_{Q_2-nP} R^S_P = A'_{Q_2-nP_1} R^S_{P_1}$

$$s_{Q_2} = A'_{Q_2-nP_1} R^S_{P_1} + A'_{Q_2-nQ_1} \cdot s_{Q_1}. \tag{1.69}$$

Es werden nun die letzten beiden Beziehungen aus der Definition des Schlupfvektors für eine Zeilenbasis hergeleitet. Da B regulär ist, folgt daraus die Wohldefiniertheit. Durch Einsezten von (1.50) in $r^Z = A^Z_{P.} x$ erhält man

$$\begin{pmatrix} r^Z_{P_1} \\ r^Z_{P_2} \end{pmatrix} = \begin{pmatrix} I_{|P_1|} & 0 \\ A'_{P_2-nP_1} & A'_{P_2-nQ_1} \end{pmatrix} \cdot \begin{pmatrix} x_{P_1} \\ x_{Q_1} \end{pmatrix}.$$

Somit ist $x_{P_1} = r^Z_{P_1}$ und $x_{Q_1} = (A'_{P_2-nQ_1})^{-1}(r^Z_{P_2} - A'_{P_2-nP_1} r^Z_{P_1})$. Für $1 \le i \le m$ gilt deshalb $s_{Q_i} = A^Z_{Q_i.} \cdot x = A^Z_{Q_i P_1} \cdot x_{P_1} + A^Z_{Q_i Q_1} \cdot x_{Q_1} = A^Z_{Q_i P_1} \cdot r^Z_{P_1} + A^Z_{Q_i Q_1} \cdot (A'_{P_2-nQ_1})^{-1}(r^Z_{P_2} - A'_{P_2-nP_1} r^Z_{P_1})$. Für $Q_i \in Q_1$ ist $A^Z_{Q_i P_1} = 0$ und $A^Z_{Q_i Q_1} = e^T_{Q_i}$, so daß gilt

$$s_{Q_1} = (A'_{P_2-nQ_1})^{-1}(r^Z_{P_2} - A'_{P_2-nP_1} r^Z_{P_1}).$$

Damit folgt für s_{Q_2}

$$s_{Q_2} = A^Z_{Q_2 P_1} \cdot r^Z_{P_1} + A^Z_{Q_2 Q_1} \cdot s_{Q_1}.$$

Nun ist aber $R^S_i = r^Z_i$ für $1 \le i \le n$, was den Beweis der Wohldefiniertheit von (1.67) vollendet.

■

Mit den Vektoren zur allgemeinen Spaltenbasis werden nun das Zulässigkeits- und das Optimalitätskriterium aufgestellt. Dazu bedarf es auch des Vektors der Dualvariablen.

SATZ 24

Sei $\mathcal{B}$ eine Basis von (1.45). Dann gilt für die Dualvariablen

$$y^T = (r^S)_P^T - d_P^T. \tag{1.70}$$

Ferner ist $\mathcal{B}$ zulässig, wenn

$$L_Q^S \leq s_Q \leq U_Q^S \tag{1.71}$$

und optimal, wenn

$$l_P^S \leq d_P \leq u_P^S. \tag{1.72}$$

BEWEIS:

Als erste wird Gleichung (1.70) bewiesen. Es gilt $r^S = c_P^S$, so daß $y = c_P^S - d_P$ zu zeigen ist. Definiere $\bar{y} \in \mathbb{R}^{n+m}$ so, daß $\bar{y}_P = y$. Setzt man $\bar{y}_P$ und (1.50) in (1.56) ein und multipliziert mit B, so gilt wegen $R^Z = 0$:

$$(\bar{y}_{P_1}^T, \bar{y}_{P_2}^T) \cdot \begin{pmatrix} I_{|P_1|} & 0 \\ A'_{P_2-nP_1} & A'_{P_2-nQ_1} \end{pmatrix} = (\bar{y}_{P_1}^T + \bar{y}_{P_2}^T \cdot A'_{P_2-nP_1}, \bar{y}_{P_2}^T \cdot A'_{P_2-nQ_1}) = (c_{P_1}^T, c_{Q_1}^T),$$

woraus die Gleichungen

$$\bar{y}_{P_2}^T = c_{Q_1}^T \cdot (A'_{P_2-nQ_1})^{-1} \tag{1.73}$$

$$\bar{y}_{P_1}^T = c_{P_1}^T - \bar{y}_{P_2}^T \cdot A'_{P_2-nP_1} \tag{1.74}$$

folgen. Entsprechend folgt aus (1.65) mit der Darstellung (1.51) der Spaltenbasis nach Multiplikation mit B

$$\begin{aligned} (t_{Q_2-n}^T, t_{P_2-n}^T) \cdot \begin{pmatrix} A'_{Q_2-nQ_1} & -I_{|Q_2|} \\ A'_{P_2-nQ_1} & 0 \end{pmatrix} &= (t_{Q_2-n}^T \cdot A'_{Q_2-nQ_1} + t_{P_2-n}^T \cdot A'_{P_2-nQ_1}, -t_{Q_2-n}^T) \\ &= ((c^S)_{Q_1}^T, (c^S)_{Q_2}^T) = (c_{Q_1}^T, 0), \end{aligned}$$

woraus $t_{Q_2-n}^T = 0$ und $t_{P_2-n}^T = c_{Q_1}^T \cdot (A'_{P_2-nQ_1})^{-1}$ folgt. Damit ist $d^T = t^T A^S = t_{P_2-n}^T A_{P_2-n.}^S$, und man erhält schließlich

$$\begin{aligned} (c_{P_2}^S)^T - d_{P_2}^T &= 0 - d_{P_2}^T = -t_{P_2-n}^T(-I_{|P_2-n|}) = t_{P_2-n}^T = c_{Q_1}^T \cdot (A'_{P_2-nQ_1})^{-1} = \bar{y}_{P_2}^T \\ (c_{P_1}^S)^T - d_{P_1}^T = c_{P_1}^T - d_{P_1}^T &= c_{P_1}^T - t_{P_2-n}^T A'_{P_2-nP_1} = c_{P_1}^T - \bar{y}_{P_2}^T A'_{P_2-nP_1} = \bar{y}_{P_1}^T, \end{aligned}$$

was nach der Definition von $\bar{y}$ den Beweis von Gleichung (1.70) beendet.

Die Zulässigkeitsbedingung (1.71) folgt wegen $L^S = l^Z$ und $U^S = u^Z$ direkt aus (1.59). Schließlich folgt das Optimalitätskriterium (1.72) wegen $l_P^S = c_P^S - U_P^Z$ und $u_P^S = c_P^S - L_P^Z$ aus Satz 22.

■

Vergleicht man (1.56) mit (1.67), (1.57) mit (1.65) und (1.58) mit (1.66), so erkennt man, daß die Rechenoperationen bis auf Transposition jeweils übereinstimmen. Diese Beobachtung wird von der folgenden Definition ausgenutzt, mit deren Hilfe alle für Simplex-Algorithmen relevanten Vektoren in eine möglichst darstellungsunabhängige Form gebracht werden.

DEFINITION 9

Sei $\mathcal{B}$ eine Basis von (1.45) und bezeichne mit $\mathcal{D} \in \{Z, S\}$ eine Darstellung der Basismatrix. Je nach Wahl von $\mathcal{D}$ definiere die Bezeichungen gemäß folgender Tabelle:

$\mathcal{D}$	Z	S
B, B_f, B_l, B_u, B_x	P, P_f, P_l, P_u, P_x	Q, Q_f, Q_l, Q_u, Q_x
N, N_f, N_l, N_u, N_x	Q, Q_f, Q_l, Q_u, Q_x	P, P_f, P_l, P_u, P_x
A, b	$(A^Z)^T, c$	$A^S, 0$
L	L^Z	L^S
U	U^Z	U^S
l	l^Z	l^S
u	u^Z	u^S
R	R^Z	R^S
r	r^Z	r^S

Dann heißt $A_{.B}$ die Basismatrix,

$$f = A_{.B}^{-1}(b - AR) \tag{1.75}$$

der Zulässigkeitsvektor,

$$h^T = r^T A_{.B}^{-1} \tag{1.76}$$

der Hilfsvektor *und*

$$g^T = h^T A \tag{1.77}$$

der Pricing-Vektor.

Die Namen Zulässigkeitsvektor für f und Pricing-Vektor für g beziehen sich auf den ursprünglichen Simplex-Algorithmus, nämlich den primalen Simplex mit Spaltenbasis. Dort bezeichnet g die reduzierten Kosten, die beim Pricing ausgewertet werden, wohingegen f die (primale) Zulässigkeit der Basis bestimmt.

Man beachte, daß alle Größen direkt für LPs der Form (1.45) bestimmt werden können und keine Datenstrukturen benötigen, die von der gewählten Basisdarstellung abhängen. Insbesondere brauchen die Matrizen A^Z und A^S nicht expliziet konstruiert zu werden. Statt

dessen kann der jeweilige I-Teil der Matrix bei der Berechnung von f und g algorithmisch ausgenutzt werden. Für den Vektor g etwa stimmt dieser Teil direkt mit h überein.

Die Bezeichungen wurden so gewählt, daß folgender Satz unabhängig von der Darstellung gilt.

SATZ 25

Mit den Bezeichnungen aus Definition 9 ist $\mathcal{B}$ optimal und zulässig, wenn

$$l_N \leq g_N \leq u_N \quad \textit{und} \tag{1.78}$$
$$L_B \leq f \leq U_B \tag{1.79}$$

gilt.

BEWEIS:
Dieser Satz folgt unmittelbar aus den Sätzen 22 und 24 sowie Definition 9.

■

Nun können Simplex-Algorithmen formuliert werden, die nur die Größen aus Definition 9 verwenden und somit i.w. unabhängig von der gewählten Basisdarstellung sind. Das Ziel der Algorithmen ist, eine Basis zu finden, die beide Ungleichungen (1.78) und (1.79) erfüllt. Dabei wird wieder zwischen zwei algorithmischen Varianten unterschieden. Eine arbeitet mit Basen, die (1.78) erfüllen, während die andere Basen benötigt, für die (1.79) gilt.

Da g_N und f in verschiedenen Basisdarstellungen jeweils dualen Vektoren zugeordnet sind, kann einer algorithmischen Variante erst nach Wahl der Basisdarstellung ein mathematischer Algorithmus zugeordnet werden. Die Zuordnung erfolgt wieder nach Tabelle 1.1

1.2.4.1 Einfügender Algorithmus

Es wird nun der einfügende Simplex-Algorithmus in den von der Basisdarstellung unabhängigen Größen aus Definition 9 aufgestellt. Er kann weitgehend von Algorithmus 2 oder 3 übernommen werden, wobei das Vorhandensein oberer und unterer Schranken eingearbeitet werden muß. Wenn eine Spaltenbasis verwendet wird, handelt es sich um einen primalen, andernfalls um einen dualen Simplex-Algorithmus (vgl. Tablelle 1.1). Zur Vereinfachung der Notation wird bei der Formulierung des Algorithmus die Abhängigkeit der Vektoren von der Basis ...($\mathcal{B}$) nicht explizit aufgeführt. Statt dessen werden Größen, die zur aktualisierten Basis einer Iteration gehören, mit einem Strich gekennzeichnet.

ALGORITHMUS 5 (EINFÜGENDER SIMPLEX-ALGORITHMUS)

Sei $\mathcal{B}$ eine Basis zum LP (1.45) und es gelte (1.79). Alle Größen aus Definition 9 seien entsprechend initialisiert.

Schritt 1 (Pricing):
Falls $l_N \leq g_N \leq u_N$ terminiere:
x^Z ist optimaler Lösungsvektor des LPs.
Sonst wähle q derart, daß $g_q > \phi$ mit $\phi = u_q$ oder $g_q < \phi$ mit $\phi = l_q$.

Schritt 2: Setze
$\Delta f \leftarrow A_{.B}^{-1} A_{.q}$ und

$$\Phi_0 \leftarrow \begin{cases} \infty & \text{, falls Zeilenbasis und } g_q < l_q \\ -\infty & \text{, falls Zeilenbasis und } g_q > u_q \\ L_q - U_q & \text{, falls Spaltenbasis und } g_q > u_q \\ U_q - L_q & \text{, falls Spaltenbasis und } g_q < l_q. \end{cases}$$

Schritt 3 (Quotiententest):

Falls $\Phi_0 > 0$ setze:
$\Phi_+ = \frac{U_{B_{p_+}} - f_{p_+}}{\Delta f_{p_+}}$, mit $p_+ \in \arg\min\{\frac{U_{B_i} - f_i}{\Delta f_i} : \Delta f_i > 0\}$,
$\Phi_- = \frac{L_{B_{p_-}} - f_{p_-}}{\Delta f_{p_-}}$, mit $p_- \in \arg\min\{\frac{L_{B_i} - f_i}{\Delta f_i} : \Delta f_i < 0\}$ und
$\Phi = \min\{\Phi_0, \Phi_+, \Phi_-\}$.

Falls $\Phi_0 < 0$ setze:
$\Phi_+ = \frac{L_{B_{p_+}} - f_{p_+}}{\Delta f_{p_+}}$, mit $p_+ \in \arg\max\{\frac{L_{B_i} - f_i}{\Delta f_i} : \Delta f_i > 0\}$,
$\Phi_- = \frac{U_{B_{p_-}} - f_{p_-}}{\Delta f_{p_-}}$, mit $p_- \in \arg\max\{\frac{U_{B_i} - f_i}{\Delta f_i} : \Delta f_i < 0\}$ und
$\Phi = \max\{\Phi_0, \Phi_+, \Phi_-\}$.

Ist $\Phi = \pm\infty$ terminiere:
Das LP ist unzulässig oder unbeschränkt.

Sonst setze $p = \begin{cases} p_- & \text{, falls } \Phi = \Phi_- \\ p_+ & \text{, sonst} \end{cases}$
und für $\Phi \neq \Phi_0$ setze ρ derart, daß $\Phi = \frac{\rho - f_p}{\Delta f_p}$.

Schritt 4: Falls $\Phi \neq \Phi_0$ setze:
$\Delta h^T \leftarrow e_p^T A_{.B}^{-1}$ und
$\Delta g^T \leftarrow \Delta h^T \cdot A$

Schritt 5 (Update):

Falls $\Phi = \Phi_0$ Setze:
$f \leftarrow f' = f + \Phi \cdot \Delta f$ und
$\mathcal{B} \leftarrow \mathcal{B}' = (B, N')$, mit $N_i' = N_i$, für $i \neq q$ und
$N_u' = N_u \cup q$ und $N_l' = N_u \setminus q$, falls $q \in N_l$
$N_u' = N_u \setminus q$ und $N_l' = N_u \cup q$, sonst.

Falls $\Phi \neq \Phi_0$ Setze:

$$\begin{array}{lcl} \Theta & \leftarrow & (\phi - g_q)/\Delta g_q \\ h & \leftarrow & h' = h + \Theta \cdot \Delta h \\ g & \leftarrow & g' = g + \Theta \cdot \Delta g \\ f & \leftarrow & f' = f + \Phi \cdot \Delta f + (R_q - \Phi - \rho)e_p \\ \mathcal{B} & \leftarrow & \mathcal{B}' = (B', N'), \text{ mit} \end{array}$$

$$B'_i = \begin{cases} B_i & \text{, falls } i \neq p \\ q & \text{, falls } i = p \end{cases} \text{ und}$$

$$N'_i = \begin{cases} N_i & \text{, falls } i \neq q \\ B_p & \text{, falls } i = q \end{cases},$$

wobei die Einordnung in die Mengen $B'_f, \ldots, N'_x$ gemäß Definition erfolgt.

Aktualisiere die Vektoren L, U, l, u, R, r.

Schritt 6: Gehe zu Schritt 1

Satz 26 (partielle Korrektheit von Algorithmus 5)
Algorithmus 5 arbeitet partiell korrekt.

Beweis:

1. Bei Terminierung in Schritt 1 folgt die Behauptung aus Satz 25.

2. Es wird nun gezeigt, daß $\mathcal{B}'$ in Schritt 6 eine Basis mit Vektoren f', g' und h' ist, für die $L'_{B'} \leq f' \leq U'_{B'}$ gilt. Der Beweis erfolgt in vier Teilschritten:

   (a) $\mathcal{B}'$ ist eine Basis:
   Falls in Schritt 5 $\Phi \neq \Phi_0$ ist, gelten die Basisbedingungen 1-3 und 5 für $\mathcal{B}'$, da sie es auch für $\mathcal{B}$ tun. Die Basisbedingung 4 folgt aus Satz 16, da nach Schritt 3 $\Delta f_p = e_p^T A_{.B}^{-1} A_{.q} \neq 0$ gilt.
   Ist in Schritt 5 hingegen $\Phi = \Phi_0$, so folgt nach Schritt 2, daß es sich um eine Spaltenbasis mit $-\infty < L_q \neq U_q < \infty$ handelt und deshalb $q \in N_l \cup N_u$ gilt. Wie schon $\mathcal{B}$ erfüllt deshalb auch $\mathcal{B}'$ die Basisbedingungen 1-3 und 5. Da $B' = B$ gilt auch die Basisbedingung 4.

   (b) Die Vektoren g' und h' gehören zu $\mathcal{B}'$:
   Für den Fall $\Phi = \Phi_0$ ist $r' = r$ und $B' = B$, woraus $g' = g$ und $h' = h$ folgt.
   Betrachte nun den Fall $\Phi \neq \Phi_0$. Da $r' = r + (\phi - r_p)e_p$ gilt nach (1.8) und (1.9) aus Satz 5 $(h')^T = h^T + \theta e_p^T A_{.B}^{-1} = h^T + \theta \Delta h^T$, wobei $\theta = \frac{\phi - h^T A_{.q}}{(A_{.B}^{-1} A_{.q})_p} = \frac{\phi - g_q}{\Delta g_p} = \Theta$ ist. Schließlich ist $(g')^T = (h')^T A = h^T A + \Theta \Delta h^T A = g^T + \Theta \Delta g^T$.

(c) Der Vektor f' gehört zu $\mathcal{B}'$, d.h. es ist $A_{.B'}f' = b - AR'$ zu zeigen.
Im Fall von $\Phi \neq \Phi_0$ sei $j = B_p$. Damit gilt $A_{.B'} = A_{.B} + (A_{.q} - A_{.j})e_p^T$, $R' = R - R_q e_q + \rho e_j$ und $f' = f + \Phi \cdot \Delta f + (R_q - \Phi - \rho)e_p$. Somit ist

$$\begin{aligned}
A_{.B'}f' &= [A_{.B} + (A_{.q} - A_{.j})e_p^T] \cdot [f + \Phi \cdot \Delta f + (R_q - \Phi - \rho)e_p] \\
&= A_{.B}f + \Phi A_{.B}\Delta f + (R_q - \Phi - \rho)A_{.j} + \\
&\quad (A_{.q} - A_{.j})f_p + (A_{.q} - A_{.j})\Phi\Delta f_p + (A_{.q} - A_{.j})(R_q - \Phi - \rho) \\
&= b - AR + \Phi A_{.q} + (R_q - \Phi - \rho)A_{.j} + \\
&\quad (A_{.q} - A_{.j})f_p + (A_{.q} - A_{.j})(\rho - f_p) + (A_{.q} - A_{.j})(R_q - \Phi - \rho) \\
&= b - AR + (\Phi + f_p + \rho - f_p + R_q - \Phi - \rho)A_{.q} + \\
&\quad (R_q - \Phi - \rho - f_p - \rho + f_p - R_q + \Phi + \rho)A_{.j} \\
&= b - (AR - A_{.q}R_q + A_{.j}\rho) \\
&= b - A(R - e_q R_q + e_j \rho) \\
&= b - AR'.
\end{aligned}$$

Im Fall $\Phi = \Phi_0$ ist $R' = R - \Phi_0 e_q$, $B' = B$ und $f' = f + \Phi_0 \Delta f$. Deshalb gilt:

$$\begin{aligned}
A_{.B'}f' &= A_{.B}(f + \Phi_0 \Delta f) \\
&= b - AR + \Phi_0 A_{.q} \\
&= b - A(R - \Phi_0 e_q) \\
&= b - AR'.
\end{aligned}$$

(d) $L'_{B'} \leq f' \leq U'_{B'}$:
Betrachte zunächst für $\Phi \neq \Phi_0$ einen beliebigen Index $i \neq p$. Dafür gilt $L'_i = L_i$ und $U'_i = U_i$, und es sind zwei Fälle zu unterscheiden:

i. $\Phi_0 \cdot \Delta f_i > 0$:

$$f'_i = f_i + \Phi \Delta f_i \begin{cases} \leq f_i + \frac{U_{B_i} - f_i}{\Delta f_i}\Delta f_i = U_{B_i} \\ \geq f_i \geq L_{B_i} \end{cases}$$

ii. $\Phi_0 \cdot \Delta f_i < 0$:

$$f'_i = f_i + \Phi \Delta f_i \begin{cases} \geq f_i + \frac{L_{B_i} - f_i}{\Delta f_i}\Delta f_i = L_{B_i} \\ \leq f_i \geq U_{B_i} \end{cases}$$

Der obere Fall folgt jeweils wegen Schritt 3. Dieselben Beziehungen gelten auch im Fall $\Phi = \Phi_0$ für alle Indizes i.

Für den verbleibenden Fall $i = p$ und $\Phi \neq \Phi_0$ gilt zunächst $f'_p = f_p + \Phi\Delta f_p + R_q - \Phi - \rho = f_p + \rho - f_p + R_q - \Phi - \rho = R_q - \Phi$. Bei einer Zeilenbasis ist dabei $R_q = 0$, so daß $f'_p = \Phi$ gilt. Für $\Phi_0 > 0$ gilt dann $L'_{B_p} = 0 \leq \Phi = f'_p \leq \infty = U'_{B_p}$ und für $\Phi_0 < 0$ gilt $L'_{B_p} = -\infty \leq \Phi = f'_p \leq 0 = U'_{B_p}$.
Im Fall einer Spaltenbasis betrachten wir 3 Fälle: Für $q \in P_f$ ist $-\infty = L_q \leq f'_q \leq U_q = +\infty$. Falls $q \in P_u$ folgt $\Phi_0 > 0$ und $R_q = U_q$. Deshalb ist $f'_q = R_q - \Phi \geq U_q - (U_q - L_q) = L_q$ sowie $f'_q = R_q - \Phi \leq U_q$. Schließlich folgt für

$q \in P_l$: $\Phi_0 < 0$ und $R_q = L_q$ und somit $f'_q = R_q - \Phi \leq L_q - (L_q - U_q) = U_q$ sowie $f'_q = R_q - \Phi \geq L_q$.

3. Termination in Schritt 3:
Die Interpretation der Lösung bei Termination in Schritt 3 hängt von der gewählten Basisdarstellung ab; für eine Zeilenbasis ist das LP unzulässig, während es für eine Spaltenbasis unbeschränkt ist.

   (a) Zeilenbasis: Es ist die Unzulässigkeit des LPs zu zeigen.
   Betrachte für alle $F \subseteq B_f$ das LP

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & D^F x & \geq \quad d^F, \end{array} \tag{1.80}$$

   mit $D^F \in \mathbb{R}^{(n+1)\times n}$ und $d^F \in \mathbb{R}^{n+1}$. Es sei für $i \in \{1, \ldots, n\}$

$$D^F_{i.} = \begin{cases} -A^Z_{B_i.} & \text{, falls } B_i \in B_u \cup F \\ A^Z_{B_i.} & \text{, sonst} \end{cases} \quad \text{und} \quad d^F_i = \begin{cases} -r_i & \text{, falls } B_i \in B_u \cup F \\ r_i & \text{, sonst} \end{cases}$$

   sowie

$$D^F_{n+1.} = \begin{cases} -A^Z_{q.} & \text{, falls } g_q > u_q \\ A^Z_{q.} & \text{, sonst} \end{cases} \quad \text{und} \quad d^F_{n+1} = \begin{cases} -u_q & \text{, falls } g_q > u_q \\ l_q & \text{, sonst} \end{cases} .$$

   Da für jedes dieser LPs $Z = (\{1, \ldots, n\}, \{n+1\})$ eine zulässige Zeilenbasis und $n+1$ eine verletzte Ungleichung ist, sind sie nach Satz 11 alle unzulässig. Weil $P(l^Z, A^Z, u^Z) \subseteq \bigcup_{F \subseteq B_f} P(D^F, d^F)$, folgt die Behauptung.

   (b) Spaltenbasis:
   In diesem Fall gilt es zu beweisen, daß das LP unbeschränkt ist, d.h. daß es zu jedem $M \in \mathbb{R}$ ein $\bar{x} \in P^=(L^S, A^S, U^S)$ mit $(c^S)^T \bar{x} < M$ gibt. Sei also ein $M \in \mathbb{R}$ gegeben. Sei x der Basislösungsvektor, also $x_N = R_N$ und $x_B = f$. Gilt $(c^S)^T x < M$, so ist nichts zu zeigen.
   Andernfalls definiere Δx gemäß

$$\Delta x_i = \begin{cases} \Delta f_{B_j} & \text{, falls } i = B_j \\ -1 & \text{, falls } i = q \\ 0 & \text{, sonst.} \end{cases}$$

   Sei $\gamma = (c^S)^T x$ und $\lambda = \frac{M-1-\gamma}{g_q - c^S_q}$. Für den Vektor $\bar{x} = x + \lambda \Delta x$ gilt

$$\begin{array}{rcl} (c^S)^T \bar{x} & = & (c^S)^T x + \lambda (c^S)^T \Delta x = \gamma + \lambda((c^S)^T_B \Delta f - c^S_q) \\ & = & \gamma + \lambda(g_q - c^S_q) = \gamma + M - 1 - \gamma \\ & = & M - 1. \end{array}$$

   Außerdem ist nach Schritt 3 $L^S \leq \bar{x} \leq U^S$, da L^S und U^S nicht von der Basis abhängen, was den Beweis beendet.

■

Dieser Algorithmus ist nicht vollständig von der gewählten Basisdarstellung unabhängig. In Schritt 2 geht sie noch in die Definition von Φ_0 ein, und in Schritt 5 wird sie zur Aktualisierung der Vektoren L, U, l und u benötigt. In allen andern Schritten tritt sie jedoch nicht mehr auf. Dies umfaßt insbesondere die für verschiedene Implementierungen variierbaren Schritte, nämlich das Pricing und den Quotiententest. Bei SoPlex wird dies so ausgenutzt, daß jede Implementierung eines Pricing- oder Quotiententest-Verfahrens für beide Basisdarstellungen funktioniert, und somit gleich für zwei Algorithmen, je einen primalen und einen dualen, bereitsteht.

Was die Übersichtlichkeit von Algorithmus 5 beeinträchtigt, ist die Definition von Φ_0 und die zugehörige Fallunterscheidung in Schritt 5. Wie im obigen Beweis gezeigt, kann dieser Fall nur für eine Spaltenbasis, also bei einem primalen Algorithmus eintreten. Hier kann es nämlich vorkommen, daß eine Variable von ihrer aktuellen Schranke entfernt werden soll, es aber gerade wieder diese Variable ist, die als erste auf ihre zweite Schranke trifft. Die betreffende Variable wird also nur von ihrer einen Schranke auf ihre andere gesetzt, während die Basismatrix unverändert bleibt.

1.2.4.2 Entfernender Algorithmus

Es wird nun der entfernende Simplex-Algorithmus in den darstellungsunabhängigen Größen aus Definition 9 formuliert. Gemäß Tabelle 1.1 handelt es sich im Fall einer Zeilenbasis um den primalen und im Fall einer Spaltenbasis um den dualen Algorithmus.

Algorithmus 6 (Entfernender Simplex-Algorithmus)
Sei $\mathcal{B}$ eine Basis zum LP (1.45) und es gelte (1.78). Alle Größen aus Definition 9 seien entsprechend initialisiert.

Schritt 1 (Pricing):
Falls $L_B \leq f \leq U_B$ terminiere:
x^Z ist optimaler Lösungsvektor des LPs.
Sonst wähle p, mit $f_p > U_{B_p}$ oder $f_p < L_{B_p}$.

Schritt 2: Setze
$\Delta h^T \leftarrow e_p^T A_{.B}^{-1}$ und
$\Delta g^T \leftarrow \Delta h^T \cdot A$

Falls Spaltenbasis setze:
$\Theta_0 = +\infty$, $\rho = U_p$ und $l_{B_q} = +\infty$, wenn $f_p > U_{B_p}$, und
$\Theta_0 = -\infty$, $\rho = L_p$ und $u_{B_q} = -\infty$, sonst.
Falls Zeilenbasis setze:
$\Theta_0 = -\infty$ und $\rho = U_p$, wenn $f_p > U_{B_p}$, und
$\Theta_0 = +\infty$ und $\rho = L_p$, sonst.

Schritt 3 (Quotiententest):

Falls $\Theta_0 > 0$ setze:
$\Theta_+ = \frac{u_{q_+} - g_{q_+}}{\Delta g_{q_+}}$, mit $q_+ \in \arg\min\{\frac{u_i - g_i}{\Delta g_i} : \Delta g_i > 0\}$,
$\Theta_- = \frac{l_{q_-} - g_{q_-}}{\Delta g_{q_-}}$, mit $q_- \in \arg\min\{\frac{l_i - g_i}{\Delta g_i} : \Delta g_i < 0\}$ und
$\Theta = \min\{\Theta_+, \Theta_-\}$.

Falls $\Theta_0 < 0$ setze:
$\Theta_+ = \frac{l_{q_+} - g_{q_+}}{\Delta g_{q_+}}$, mit $q_+ \in \arg\max\{\frac{l_i - g_i}{\Delta g_i} : \Delta g_i > 0\}$,
$\Theta_- = \frac{u_{q_-} - g_{q_-}}{\Delta g_{q_-}}$, mit $q_- \in \arg\max\{\frac{u_i - g_i}{\Delta g_i} : \Delta g_i < 0\}$ und
$\Theta = \max\{\Theta_+, \Theta -\}$.

Ist $\Theta = \pm\infty$ terminiere:
Das LP ist unzulässig oder unbeschränkt.

Sonst setze $q = \begin{cases} q_- & \text{, falls } \Theta = \Theta_- \\ q_+ & \text{, sonst} \end{cases}$
und ϕ derart, daß $\Theta = \frac{\phi - g_q}{\Delta g_q}$.

Schritt 4: Falls $q \notin B$ setze $\Delta f \leftarrow A_{\cdot B}^{-1} A_{\cdot q}$

Schritt 5 (Update):
$h \leftarrow h' = h + \Theta \cdot \Delta h$
$g \leftarrow g' = g + \Theta \cdot \Delta g$

Falls $q \in B$ setze:
$\mathcal{B} \leftarrow \mathcal{B}' = (B', N)$, mit $B'_i = B_i$ falls $i \neq q$ und
$B'_u = B_u \cup B_p$ und $B'_l = B_u \setminus B_p$, falls $B_p \in B_l$
$B'_u = B_u \setminus B_p$ und $B'_l = B_u \cup B_p$, sonst.

Falls $q \notin B$ setze:
$\Phi \leftarrow (\rho - f_p)/\Delta f_p$
$f \leftarrow f' = f + \Phi \cdot \Delta f + (R_q - \Phi - \rho) e_p$
$\mathcal{B} \leftarrow \mathcal{B}' = (B', N')$, mit
$B'_i = \begin{cases} B_i & \text{, falls } i \neq p \\ q & \text{, falls } i = p \end{cases}$ und
$N'_i = \begin{cases} N_i & \text{, falls } i \neq q \\ B_p & \text{, falls } i = q \end{cases}$,
wobei die Einordnung in die Mengen $B'_f, \ldots, N'_x$ gemäß Definition erfolgt.

Aktualisiere die Vektoren L, U, l, u, R, r.

Schritt 6: Gehe zu Schritt 1

Satz 27 (partielle Korrektheit von Algorithmus 6)
Algorithmus 6 arbeitet partiell korrekt.

Beweis:

1. Bei Terminierung in Schritt 1 folgt die Behauptung aus Satz 25.

2. Es wird nun gezeigt, daß $\mathcal{B}'$ in Schritt 6 eine Basis mit Vektoren f', g' und h' ist, für die wieder (1.78) gilt.

   Daß $\mathcal{B}'$ eine Basis ist, folgt für $q \notin B$ analog zum Beweis von Satz 26. Für $q \in B$ beachte man zunächst, daß $q = B_p$, denn es gilt $\Delta g_B^T = \Delta h^T A_{.B} = e_p^T A_{.B}^{-1} A_{.B} = e_p^T$. Ferner ist in diesem Fall $-\infty < l_q \neq u_q < \infty$, so daß $B_q \in B_u \cup B_l$ folgt. Damit ist $\mathcal{B}'$ eine Basis, da $\mathcal{B}$ eine Basis ist.

   Entsprechend wie im Beweis von Satz 26 folgt nach Satz 5, daß die Vektoren f', g' und h' zu $\mathcal{B}'$ gehören.

   Daß in Schritt 6 die Ungleichung $l' \leq g' \leq u'$ gilt, sieht man für eine Zeilenbasis wie folgt. Da in diesem Fall $l' = l$ und $u' = u$, gilt für alle i für $\Theta \Delta g_i > 0$:

$$g_i' = g_i + \Theta \Delta g_i \begin{cases} \leq g_i + \frac{u_i - g_i}{\Delta g_i} \Delta g_i = u_i' \\ \geq g_i \geq l_i' \end{cases}$$

   und für $\Theta \Delta g_i < 0$:

$$g_i' = g_i + \Theta \Delta g_i \begin{cases} \geq g_i + \frac{l_i - g_i}{\Delta g_i} \Delta g_i = l_i' \\ \leq g_i \leq u_i'. \end{cases}$$

   Für eine Spaltenbasis wird zunächst l_p bzw. u_p in Schritt 2 gemäß der zu erwartenden neuen Basis $\mathcal{B}'$ für den Fall $q \notin B$ initialisiert. Daher kann in Schritt 6 der Fall $q \in B$ nicht eintreten. Mit diesen Schranken l und u gelten wieder die obigen Beziehungen für alle $i \neq q$. Schließlich ist $g_q' = g_q + \frac{\phi - g_q}{\Delta g_q} \Delta g_p = \phi$ und $u_q' = l_q' = \phi$, so daß insgesamt $l' \leq g' \leq u'$ gilt.

3. Zuletzt wird die Termination in Schritt 3 behandelt, wobei wieder zwischen beiden Basisdarstellungen unterschieden werden muß.

   (a) Zeilenbasis: Das LP ist unbeschränkt.
   Es ist zu zeigen, daß es für jede Zahl $M \in \mathbb{R}$ einen Vektor $\bar{x} \in \mathcal{P}(l^Z, A^Z, u^Z)$ gibt mit $c^T \bar{x} < M$. Gilt dies bereits für den Basislösungsvektor $x = h$, so ist nichts zu zeigen. Andernfalls sei $\bar{x} = x + \theta \Delta h$, mit $\theta = \frac{M - 1 - c^T x}{c^T \Delta h}$. Damit ist $c^T \bar{x} = c^T x + \theta c^T \Delta h = M - 1 < M$ und nach Schritt 3 gilt $l \leq x \leq u$.

   (b) Spaltenbasis: Das LP ist unzulässig.
   Betrachte zu derselben Basis $\mathcal{B} = (P, Q)$ das Zeilen-LP (1.47). Alle dazu gehörenden Größen markieren wir mit dem Exponent $...^Z$, während wir die Größen zum Spalten-LP mit $...^S$ kennzeichnen. Nach den Definitionen 7, 8 und

9 gelten folgende Zusammenhänge:

$$\begin{aligned} N^Z &= B^S = Q \\ B^Z &= N^S = P \\ U_P^Z &= c_P^S - l_P^S \\ L_P^Z &= c_P^S - u_P^S \\ f^Z &= c_P^S - g_P^S \\ u^Z &= U^S \\ l^Z &= L^S \\ g_Q^Z &= f^S. \end{aligned}$$

Demnach gilt für die Zeilenbasis $l_Q^Z \leq g_Q^Z \leq u_Q^Z$ (1.78) und somit kann sie für Algorithmus 5 verwendet werden. Desweiteren gilt $g_{Q_p}^Z = f_p^S > U_{Q_p}^S = u_{Q_p}^Z$ bzw. $g_{Q_p}^Z = f_p^S < L_{Q_p}^S = l_{Q_p}^Z$ je nach Fall in Schritt 1 von Algorithmus 6. Somit eignet sich $q = Q_p^Z$ als eintretender Index von Algorithmus 5. Wenn wir noch $-\Delta g_P^S = \Delta f^Z$ zeigen, folgt mit den oberen Beziehungen, daß Algorithmus 5 in Schritt 3 terminiert, da dies auch Algorithmus 6 tut. Nach Satz 26 bedeuted dies aber die Unzulässigkeit des LPs und, da Zeilen- und Spalten-LP äquivalent sind, die Behauptung.
Es muß also noch $(\Delta f^Z)^T = A_{q.}^Z \cdot (A^Z)_{P.}^{-1} \stackrel{!}{=} -e_p^T \cdot (A^S)_{.Q}^{-1} \cdot A_{.P}^S = -(\Delta g_P^S)^T$ gezeigt werden. Wir weisen die dazu äquivalente Beziehung

$$-A_{q.}^Z = e_p^T \cdot (A^S)_{.Q}^{-1} \cdot A_{.P}^S \cdot (A^Z)_{P.} \tag{1.81}$$

nach. Dazu berechnen wir nach geeigneter Permutation mit (1.50)

$$\begin{aligned} A_{.P}^S \cdot A_{.P}^Z &= \begin{pmatrix} A'_{Q_2-nP_1} & 0 \\ A'_{P_2-nP_1} & -I_{|P_2|} \end{pmatrix} \cdot \begin{pmatrix} I_{|P_1|} & 0 \\ A'_{P_2-nP_1} & A'_{P_2-nQ_1} \end{pmatrix} \\ &= \begin{pmatrix} A'_{Q_2-nP_1} & 0 \\ 0 & -A'_{P_2-nQ_1} \end{pmatrix}. \end{aligned} \tag{1.82}$$

Ferner gilt für $x^T = e_p^T \cdot (A^S)_{.Q}^{-1}$ nach (1.51)

$$e_p^T = x^T \cdot A_{.Q}^S = (x_{Q_2-n}^T \cdot A'_{Q_2-nQ_1} + x_{P_2-n}^T \cdot A'_{P_2-nQ_1}, -x_{Q_2-n}^T). \tag{1.83}$$

Nun sind zwei Fälle zu unterscheiden. Falls $q = Q_p \in Q_1$ folgt $x_{Q_2-n}^T = 0$ und deshalb $x_{P_2-n}^T \cdot A'_{P_2-nQ_1} = e_q^T$. Damit ist $x^T \cdot (A_{.P}^S \cdot A_{.P}^Z) = (x_{Q_2-n} A'_{Q_2-nP_1}, x_{P_2-n} A'_{P_2-nQ_1}) = -e_p^T = -A_{q.}^Z$, d.h. (1.81) wurde bewiesen. Im anderen Fall $q = Q_p \in Q_2$ ist $x_{Q_2-n}^T = -e_{q-n}^T$ woraus $x_{P_2-n}^T \cdot A'_{P_2-nQ_1} = e_{q-n}^T \cdot A'_{Q_2-nQ_1}$ folgt. Damit gilt nun $x^T \cdot (A_{.P}^S \cdot A_{.P}^Z) = (-e_{q-n}^T \cdot A'_{Q_2-nP_1}, -e_{q-n}^T \cdot A'_{P_2-nQ_1}) = -A'_{q-n.} = -A_{q.}^Z$ was den Beweis beendet.

■

Der Beweis zum Fall der Termination in Schritt 3 für eine Spaltenbasis basiert auf der Dualitätsbeziehung zwischen beiden Basisdarstellungen und beiden Algorithmustypen. Auch der Fall einer Zeilenbasis hätte so bewiesen werden können. Dazu würde man die Bedeutung von Zeilen- und Spalten-LP vertauschen, und es müßte noch der Fall $\Theta = \Theta_0$ ausgeschlossen werden. Der direkte Beweis schien da einfacher, zumal ähnliche Beweise bereits vorgestellt wurden.

1.3 Stabilität des Simplex-Verfahrens

Die Aufgabe numerischer Algorithmen ist die Berechnung einer Abbildung $\mathbb{R}^n \rightarrow \mathbb{R}^m$. Bei der Implementierung auf realen Computern ist man stets mit dem Problem endlicher Genauigkeit konfrontiert. Da ein Rechner nur ein Gitter von endlich vielen Zahlenwerten darstellen kann, wird jede reelle Zahl durch die nächstliegende Zahl in diesem Gitter approximiert. Also repräsentiert jede im Computer dargestellte Gleitkommazahl ein Intervall und ist somit a priori fehlerbehaftet. Der relative Fehler heutiger 64-Bit-Gleitkommaarithmetik liegt in der Größenordnung von 10^{-16}.

Ein numerischer Algorithmus besteht aus einer Reihe von Rechenoperationen, bei denen sich Fehler akkumulieren können. Dies kann so gravierende Ausmaße annehmen, daß das berechnete Resultat keine vernünftige Aussage über die zu lösende Aufgabe mehr erlaubt. Auch der Simplex-Algorithmus bleibt von diesem Problem nicht verschont. Deshalb wird er in diesem Abschnitt auf seine numerischen Eigenschaften hin untersucht.

In den Abschnitten 1.3.1 und 1.3.2 werden die Begriffe der *Kondition eines numerischen Problems* und der *Stabilität eines numerischen Algorithmus* eingeführt. Wir orientieren uns dabei an den Darstellungen in [34] und [39]. Anschließend wird in Abschnitt 1.3.3 die Stelle im Simplex-Algorithmus aufgezeigt, die numerische Schwierigkeiten in sich birgt, und worauf zu achten ist, um eine Instabilität zu umgehen. Schließlich werden in Abschnitt 1.3.4 drei verschiedene Stabilisierungen vorgestellt und diskutiert, die für SoPlex implementiert wurden.

1.3.1 Kondition

Ein *numerisches Problem* $r = f(e)$ besteht darin, einen Eingabevektor $e \in \mathbb{R}^n$ auf einen Resultatvektor $r \in \mathbb{R}^m$ abzubilden. Nun kann der Eingabevektor auf keinem Computer exakt dargestellt sondern nur durch Gleitkommazahlen approximiert werden. Also repräsentiert jeder Gleitkommavektor in Wirklichkeit eine Umgebung E von Eingabevektoren, die *Fehlermenge* von e. Der Computer ist nicht in der Lage, zwischen verschiedenen Elementen von E zu unterscheiden. Daher ist es sinnvoll zu studieren, wie E durch f abgebildet wird. Dies wird von dem Begriff der *Kondition* eines Problems beschrieben.

DEFINITION 10 (KONDITION)

Sei $f(e)$ ein numerisches Problem, E die Fehlermenge von $e \in E$ und $R = f(E)$. Dann ist mit der Bezeichnung[2] $[X] = \sup_{x' \in X} \frac{\|x-x'\|}{\|x\|}$ *für* $x \in X$

$$\kappa = \frac{[E]}{[R]}, \tag{1.84}$$

die Kondition *des Problems f. Ist $\kappa \approx 1$, so nennt man das Problem* gut konditioniert, *für $\kappa \gg 1$ nennt man es* schlecht konditioniert *und für $\kappa = \infty$* unsachgemäß gestellt.

Ferner definiere die Kondition einer regulären Matrix A *als*

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|. \tag{1.85}$$

Die Kondition eines numerischen Problems hängt somit nicht von dem gewählten Algorithmus ab, sondern ist eine Eigenschaft des Problems selbst.

Wir betrachten die Kondition der Lösung von linearen Gleichungssystemen, da diese für Simplex-Algorithmen von zentraler Bedeutung ist (Schritte 2 und 4).

SATZ 28 (KONDITION EINER REGULÄREN MATRIX)

Für $n \in \mathbb{N}$ sei $x, b, \tilde{x}, \tilde{b}, \Delta x, \Delta b \in \mathbb{R}^n$, $A, \tilde{A}, \Delta A \in \mathbb{R}^{n \times n}$ und A nicht singulär, derart daß $\tilde{A} = A + \Delta A$, $\tilde{b} = b + \Delta b$, und $\tilde{x} = x + \Delta x$, sowie $Ax = b$ und $\tilde{A}\tilde{x} = \tilde{b}$. Die Größen mit $\tilde{}$ bezeichnen fehlerbehaftete Größen und die Werte Δ. den Fehler. Ferner sei $\epsilon(x) = \frac{\|\Delta x\|}{\|x\|}$ und $\epsilon(A) = \frac{\|\Delta A\|}{\|A\|}$. Falls $\kappa(A)\epsilon(A) < 1$ ist, gilt

$$\epsilon(x) \leq \frac{\kappa(A)}{1 - \kappa(A)\epsilon(A)}(\epsilon(A) + \epsilon(b)). \tag{1.86}$$

BEWEIS:

Nach Voraussetzung gilt $(A + \Delta A)(x + \Delta x) = b + \Delta b$ und somit $\|\Delta x\| = \|A^{-1}(b + \Delta b - Ax - \Delta Ax - \Delta A\Delta x)\| \leq \|A^{-1}\| \cdot \|\Delta b\| + \|A^{-1}\| \cdot \|\Delta A\| \cdot \|x\| + \|A^{-1}\| \cdot \|\Delta A\| \cdot \|\Delta x\|$. Wegen $\|A^{-1}\| \cdot \|\Delta A\| = \|A^{-1}\| \cdot \|A\| \cdot \|\Delta A\|/\|A\| = \kappa(A) \cdot \epsilon(A) < 1$ gilt $(1 - \kappa(A) \cdot \epsilon(A))\|\Delta x\| \leq \|A^{-1}\| \cdot \|\Delta b\| + \kappa(A) \cdot \epsilon(A) \cdot \|x\|$. Daraus folgt

$$\begin{aligned}
\epsilon(x) &= \|\Delta x\|/\|x\| \\
&\leq \frac{1}{1 - \kappa(A) \cdot \epsilon(A)} \left(\frac{\|A^{-1}\| \cdot \|A\| \cdot \|\Delta b\|}{\|A\| \cdot \|x\|} + \kappa(A) \cdot \epsilon(A) \right) \\
&\leq \frac{1}{1 - \kappa(A) \cdot \epsilon(A)} \left(\kappa(A) \frac{\|\Delta b\|}{\|b\|} + \kappa(A) \cdot \epsilon(A) \right) \\
&= \frac{\kappa(A)}{1 - \kappa(A) \cdot \epsilon(A)}(\epsilon(A) + \epsilon(b)).
\end{aligned}$$

■

[2] Anstelle des hier verwendeten relativen Fehlers kann man für $[X]$ auch ein anderes Maß wählen. Man kommt dann jeweils zu einer etwas anderen Definition der Kondition.

Satz 28 gibt eine Abschätzung für den Fehler des Resultats $\epsilon(x)$ durch den Fehler der Eingabedaten $\epsilon(A) + \epsilon(b)$. Insbesondere ist für gut konditionierte Matrizen, mit $\kappa(A) \approx 1$ und $\epsilon(A) \ll 1$, die Matrixkondition $\kappa(A)$ eine gute Approximation für die Kondition der Lösung des linearen Gleichungssystems $Ax = b$, denn dann ist $\epsilon(x) \leq \kappa(A) \cdot (\epsilon(A) + \epsilon(b))$. Da die Kondition $\kappa(A)$ nicht von der rechten Seite b abhängt, repräsentiert sie die Kondition für alle lineare Gleichungssysteme mit der Matrix A und einer beliebigen rechten Seite b.

1.3.2 Stabilität

Bei der Implementierung eines Algorithmus zur Lösung eines numerischen Problems, werden im allgemeinen Zwischenresultate berechnet, bei denen zusätzliche Fehler auftreten können. Das bedeutet, daß eine Implementierung eines Algorithmus nicht exakt die Abbildung f berechnet, sondern eine etwas andere Abbildung $\tilde{f}$. Diese bildet nun e auf $\tilde{r} = \tilde{f}(e)$ statt auf $r = f(e)$ ab. Wie dieser Fehler zu bewerten ist, wird durch die *Stabilität* eines Algorithmus beschrieben.

Es gibt zwei Ansätze zur Behandlung der Stabilität von Algorithmen, die Vorwärts- und die Rückwärtsanalyse. Wir beschränken uns hier auf den zweiten Ansatz, der für die Stabilitätsanalyse der LU-Faktorisierung zur Lösung linearer Gleichungssysteme verwendet wird. Die Grundidee dabei ist, den Fehler des Algorithmus $\tilde{f}$ durch einen erweiterten Fehler der Eingabedaten zu modellieren. Dabei repräsentiert eine Implementierung des Algorithmus eine ganze Familie F von Abbildungen, mit $f \in F$ (sonst wäre die Implementierung nicht korrekt).

Definition 11 (Stabilität)
Sei $f(e)$ ein numerisches Problem und F eine Implementierung zur Lösung von f. Setze $\tilde{E} = \{\tilde{e} : f(\tilde{e}) = \tilde{f}(e), e \in E, \tilde{f} \in F\}$. F heißt stabil, *falls $[\tilde{E}] \approx [E]$ und* instabil, *falls $[\tilde{E}] \gg [E]$.*

1.3.3 Analyse der Simplex-Algorithmen

Da bei der Definition einer Basis nur ganzzahlige Werte auftreten, ist die Kondition des Problems "Löse ein LP mit dem Simplex-Algorithmus" gerade die Kondition des Gleichungssystems der optimalen, zulässigen Basis[3]. Die numerisch stabile Lösung von dünnbesetzten linearen Gleichungssystemen — und Basismatrizen sind typischerweise dünnbesetzt — wird in Abschnitt 1.7 separat behandelt.

[3]Dieses numerische Problem ist nicht äquivalent zu dem Problem "Löse ein LP", etwa mit einem Innere-Punkte Verfahren. Dies liegt daran, daß der Simplex-Algorithmus stets noch andere Resultate liefert, nämlich die Basis und alle damit zusammenhängenden Vektoren, auch wenn man sich nur den Lösungsvektor und den Zielfunktionswert ausgeben läßt.

Im Laufe des Simplex-Algorithmus werden immer neue Basismatrizen generiert, mit denen jeweils Gleichungssysteme gelöst werden müssen. Damit der Simplex-Algorithmus nicht aufgrund explodierender numerischer Fehler zusammenbricht, muß darauf geachtet werden, daß alle auftretenden Basismatrizen gut konditioniert bleiben. Dazu untersuchen wir die Kondition von Basismatrizen nach einem Basistausch (vgl. Satz 5).

SATZ 29

Sei D eine Basismatrix und $D' = VD$, mit $V = I + e_p(r^T D^{-1} - e_p^T)$, die Basismatrix nach einem Basistausch, bei dem die i-te Zeile durch r^T ersetzt wird. Dann ist die Kondition von D' beschränkt durch

$$\kappa(D') \leq \frac{1}{|r'|} \cdot (1 + \|r^T D^{-1}\|) \cdot (1 + \|r^T D^{-1}\| + |r'|) \cdot \kappa(D), \tag{1.87}$$

wobei $r' = (r^T D^{-1})_p$ ist.

BEWEIS:

Nach (1.85) ist die Kondition von D' gegeben durch

$$\begin{array}{rcl} \kappa(D') & = & \kappa(VD) = \|VD\| \cdot \|(VD)^{-1}\| \\ & \leq & \|V\|\|V^{-1}\| \cdot \|D\|\|D^{-1}\| = \kappa(D) \cdot \kappa(V). \end{array} \tag{1.88}$$

Es muß also die Kondition von V abgeschätzt werden. Dazu bedarf es der Inversen

$$V^{-1} = I + \frac{e_p}{(r^T D^{-1})_p}(r^T D^{-1} - e_p^T), \tag{1.89}$$

denn es ist, mit $t^T = r^T D^{-1}$

$$\begin{array}{rcl} VV^{-1} & = & [I + e_p(t^T - e_p^T)] \cdot [I - \frac{e_p}{t_p}(t^T - e_p^T)] \\ & = & I + e_p t^T - e_p e_p^T - \frac{1}{t_p}(e_p t^T - e_p e_p^T + e_p t_p t^T - e_p t_p e_p^T - e_p t^T + e_p e_p^T) \\ & = & I + e_p t^T - e_p e_p^T - e_p t^T + e_p e_p^T \\ & = & I. \end{array}$$

Also sind

$$\begin{array}{rcl} \|V\| & = & \sup\limits_{\|x\|=1} \|Vx\| \\ & = & \sup\limits_{\|x\|=1} \|x + e_p r^T D^{-1} x - x_p e_p\| \\ & \leq & \sup\limits_{\|x\|=1} \|x - x_p e_p\| + \sup\limits_{\|x\|=1} \|e_p r^T D^{-1} x\| \\ & \leq & 1 + \|r^T D^{-1}\| \end{array}$$

und

$$\begin{aligned}
\|V^{-1}\| &= \sup_{\|x\|=1} \|V^{-1}x\| \\
&= \sup_{\|x\|=1} \|x + \frac{1}{r'} e_p r^T D^{-1} x - \frac{1}{r'} x_p e_p\| \\
&\leq \sup_{\|x\|=1} \|x\| + \frac{1}{|r'|} \sup_{\|x\|=1} \|x_p e_p\| + \sup_{\|x\|=1} \|\frac{1}{r'} e_p r^T D^{-1} x\| \\
&\leq 1 + \frac{1}{|r'|} + \frac{\|r^T D^{-1}\|}{|r'|},
\end{aligned}$$

woraus mit (1.88) die Behauptung folgt.

■

Dieser Satz gibt lediglich eine obere Schranke für die Kondition der neuen Basis an. Damit ist nicht gesagt, daß die Kondition der Basismatrizen stetig anwächst. Hingegen nimmt aber die Stabilität der Inversen (vgl. Abschnitt 1.7.3) stetig ab, wenn man sie mithilfe der Updatematrizen V^{-1} berechnet. Deshalb ist es unerläßlich, die Inverse (besser die LU-Zerlegung) nach einigen Updates immer wieder neu zu berechnen. Wie in Abschnitt 1.8.5 gezeigt wird, ist dies auch aus Geschwindigkeitsgründen sinnvoll.

1.3.4 Stabile Implementierungen

Nach Satz 29 müssen "kleine" Werte r' vermieden werden. Dies leuchtet auch unmittelbar ein, denn kleine Werte r' können immer auch von numerischen Fehlern bei der Berechnung von r resultieren. In diesem Fall könnte in mathematischer Genauigkeit $r' = 0$ sein, was nach den Sätzen 5 und 16 nicht erlaubt ist. Aber auch schon "kleine" Werte bergen numerische Schwierigkeiten in sich.

Wird bei den Simplex-Algorithmen (1) und (2) der Vektor $D_{q\cdot}$ mit der p-ten Basiszeile getauscht, so führt dies zu dem Wert $r' = (D_{q\cdot} D_{P\cdot}^{-1})_p$. Beim einfügenden Simplex-Algorithmus wird p und beim entfernenden wird q beim Quotiententest bestimmt. Mathematisch gesehen, ist die Auswahl (bei Abwesenheit von Degeneriertheit) eindeutig bestimmt. Vom numerischen Standpunkt aus muß man sich jedoch Alternativen schaffen, indem man die Optimalitäts- oder Zulässigkeitsbedingung leicht relaxiert. Dieser Grundgedanke wurde 1973 erstmals von P. Harris vorgeschlagen [60].

Im folgenden werden drei Stabilisierungs-Strategien am Beispiel von Algorithmus 1 vorgestellt; ihre Übertragung auf andere Simplex-Algorithmen ist offensichtlich. Sei dazu $s \geq d$ der Vektor der Schlupfvariablen. Er soll zu $s' = s + (d_q - s_q)/\Delta s_q \cdot \Delta s$ mit $\Delta s = D D_{P\cdot}^{-1} e_p$ aktualisiert werden, so daß anschließend weiterhin $s' \geq d$ gilt. Beim mathematischen Quotiententest erfolgt dies nach Satz 9 über

$$q = \arg\max \left\{ \frac{d_i - s_i}{\Delta s_i} : \Delta s_i < 0 \right\}.$$

Alle Varianten von stabilisierten Quotiententests benötigen einen Toleranzparameter δ, um den s seine Schranken d verletzen darf. Bei SoPlex beträgt der voreingestellter Wert $\delta = 10^{-6}$.

1. Ansatz („Textbook“ Quotiententest)
Die einfachste Stabilisierung besteht darin, daß man zur Auswahl des Index q, die Schranken $d - \delta$ statt d verwendet[4]:

$$q = \arg\max \left\{ \frac{d_i - s_i - \delta}{\Delta s_i} : \Delta s_i < 0 \right\}.$$

Die Idee dabei ist, daß Indizes mit großem $|\Delta s_i|$ eine größere Chance haben, an die um δ erweiterten Schranken zu gelangen. Für den Update wird der Wert $\Theta = (d_q - s_q)/\Delta s_q$ wie gewohnt nach (1.23) ohne δ berechnet, damit s' auch zur neuen Basis gehört. Dieser Quotiententest sichert das Einhalten der Schranken $x \geq d - \delta$ zu (vgl. Abb. 1.6). Es können jedoch zweierlei Probleme auftreten:

1. Da Θ mit den originalen Schranken d gebildet wird, können Pivot-Schritte geschehen, bei denen $\Theta < 0$ ist, also ein Rückschritt bei der Lösung gemacht wird. Dadurch wird ein Kreiseln als Folge von leichten Verbesserungen und Verschlechterungen möglich. Dieses Problem wird in Abschnitt 1.5 wieder aufgegriffen.

2. Für alle i, für die (nach einigen Iterationen) $s_i = d_i - \delta$ gilt, bietet auch dieser Quotiententest keine Vorzüge mehr gegenüber dem mathematischen. Dies tritt allerdings nicht so häufig auf.

Trotz dieser Probleme ist dieser Quotiententest für numerisch unproblematische LPs gut geeignet. Aufgrund des geringeren Rechenaufwands gegenüber den folgenden Ansätzen sollte er ihnen bei einfachen Problemen vorgezogen werden.

2. Ansatz (Harris Quotiententest)
Der von P. Harris vorgeschlagenen Quotiententest [60] besteht aus zwei Phasen. In der ersten Phase wird ein maximaler Wert Θ_{max} bestimmt, bis zu dem $s' \geq d - \delta$ bleibt:

$$\Theta_{max} = \max \left\{ \frac{d_i - s_i - \delta}{\Delta s_i} : \Delta s_i < 0 \right\}.$$

In der zweiten Phase wählt man unter allen Indizes i, die zu einem $\Theta < \Theta_{max}$ führen, dasjenige aus, für das $|\Delta s_i|$ maximal ist (vgl. Abb. 1.6):

$$q = \arg\max \left\{ |\Delta s_i| : \frac{d_i - s_i}{\Delta s_i} < \Theta_{max}, \Delta x_i < 0 \right\}. \tag{1.90}$$

[4] Aus numerischen Gründen (Auslöschung führender Stellen), wird δ als letzter Wert im Zähler addiert.

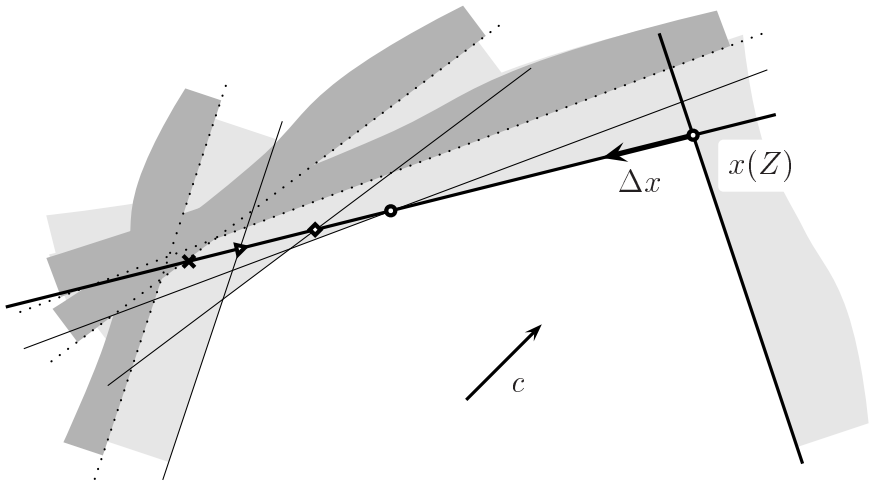


Abbildung 1.6: Stabilisierungen des Quotiententests. Das mathematische Verfahren nach (1.22) würde die neue Basislösung auf den Kreis positionieren. Aufgrund des geringen Winkels der Stützhyperebenen an diesem Schnittpunkt ist die zugehörige Basis jedoch schlecht konditioniert. Zur Stabilisierung werden daher die um einen Betrag δ verschobenen Schranken betrachtet, die gepunktet und mit dunklerer Schraffur dargestellt sind. Die Stabilisierung gemäß Ansatz 1 liefert den mit der Raute gekennzeichneten Schnittpunkt als neue Basislösung, da in diesem die zugehörige verschobene Schranke als erste erreicht wird. Dieser Schnittpunkt ist durch das Kreuz gekennzeichnet. Die Ansätze 2 und 3 wählen unter allen Schnittpunkten, die vor dem Kreuz liegen, den numerisch stabilsten aus. Dieser ist durch das Dreieck markiert.

Dieses Verfahren ist offenbar das "stabilste", das mit der Idee der Toleranz δ möglich ist. Dennoch birgt auch dieses Verfahren dieselben Probleme wie das zuvor beschriebene. Varianten des Quotiententests von Harris werden z.B. in MPSX und CPLEX eingesetzt [12, 13].

3. Ansatz (SoPlex Quotiententest)
Es folgt eine Beschreibung des für SoPlex standardmäßig empfohlenen Verfahrens. Wie das von Harris benötigt es auch zwei Phasen, wovon die zweite Phase mit der von Harris übereinstimmt. Zur Bestimmung von Θ_{max} wird jedoch erlaubt, daß sich die Schrankenüberschreitung um δ verschärft. Setze also

$$\delta_i = \begin{cases} \delta & \text{, falls } s_i \geq d_i \\ s_i - \delta & \text{sonst.} \end{cases}$$

Damit wird Θ_{max} bestimmt gemäß

$$\Theta_{max} = \max \left\{ \frac{d_i - s_i - \delta_i}{\Delta s_i} : \Delta s_i < 0 \right\}.$$

Anschließend wird q gemäß (1.90) bestimmt.

Dieses Verfahren schließt das zweite Problem aus. Das erste kann sich jedoch verschärfen. Da nun über mehrere Pivot-Schritte die Verletzung von Schranken deutlich größer als δ werden kann, ist es auch möglich, daß ein substantieller Rückschritt auftritt. Um dies zu vermeiden, wird für den Fall $s_q < d_q$ die Schranke auf $d_q = s_q$ *geschiftet*. Wie in Abschnitt 1.5 beschrieben wird, werden solche Shifts u.U. ohnehin zur Vermeidung von Kreiseln vorgenommen. Dort wird auch erklärt, wie man dennoch zu einer Lösung des ungeshifteten LPs gelangt.

1.4 Die Phasen des Simplex-Algorithmus

Um einen Simplex-Algorithmus starten zu können, bedarf es im Falle des primalen Algorithmus einer zulässigen und im Falle des dualen einer optimalen Basis. Es gibt verschiedene Methoden solche Basen zu generieren, von denen nun einige für eine zulässige Spaltenbasis zu folgendem LP vorgestellt werden:

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & Ax & = & b \\ & x & \geq & 0. \end{array} \tag{1.91}$$

Dabei gelte o.B.d.A. $b \geq 0$; sonst kann man die Zeile mit -1 multiplizieren.

Generell unterscheidet man zwischen Ein- und Zwei-Phasen-Algorithmen. Bei Ein-Phasen-Algorithmen wird das LP so modifiziert, daß eine zulässige Basis direkt angegeben und aus der Lösung des modifizierten LPs eine Lösung des Ausgangs-LPs konstruiert werden kann. Für eine "geeignet große" Zahl M bildet man das LP [8]

$$\begin{array}{lrcrcl} \min & c^T x & + & M \cdot 1^T s & & \\ \text{s.t.} & Ax & + & Is & = & b \\ & x & & & \geq & 0 \\ & & & s & \geq & 0. \end{array} \tag{1.92}$$

Für dieses LP ist die aus den Variablen s bestehende Spaltenbasis zulässig, da $s = b \geq 0$ gilt. M muß so groß gewählt werden, daß jede zulässige Basislösung von (1.91) einen geringeren Zielfunktionswert aufweist als jede zulässige Basislösung von (1.92), für die ein $s_i > 0$ ist. Terminiert der Simplex-Algorithmus mit $s = 0$, so ist x eine optimale Lösung von (1.91); andernfalls wurde die Unzulässigkeit des LPs nachgewiesen. Ein konkreter Wert für M kann z.B. über die Kodierungslänge von (1.91) theoretisch bestimmt werden. Die Größe von M führt jedoch zu numerischen Schwierigkeiten, die dieses Verfahren für eine praktische Implementierungen ausschließen.

Bei praktischen Implementierungen werden deshalb Zwei-Phasen-Algorithmen eingesetzt. Dabei wird zunächst ein sog. Phase-1-LP aufgestellt, für das eine zulässige Basis

direkt angegeben werden kann. Anschließend wird aus dessen Lösung eine zulässige Basis für das Ausgangs-LP konstruiert.

In Lehrbüchern findet man meistens einfache Varianten des Phase-1-LPs, die zwar eine numerisch stabile Implementierung erlauben, jedoch eine hohe Iterationszahl bedingen. Der einfachste Fall eines Phase-1-Problem zu (1.91) ist das LP

$$\begin{array}{lrcrcl} \min & & & 1^T s & & \\ \text{s.t.} & Ax & + & Is & = & b \\ & x & & & \geq & 0 \\ & & & s & \geq & 0. \end{array} \tag{1.93}$$

Auch hier ist die Basis aus den Variablen s zulässig. Gilt für die optimale Basis-Lösung von (1.93) $s \neq 0$, so ist das Ausgangs-LP unzulässig. Andernfalls ist die optimale Basis von (1.93) eine zulässige Basis des zu (1.91) äquivalenten LPs

$$\begin{array}{lrcrcl} \min & c^T x & & & & \\ \text{s.t.} & Ax & + & Is & = & b \\ & x & & & \geq & 0 \\ & & & s & = & 0. \end{array} \tag{1.94}$$

Der Effizienznachteil dieses Verfahrens ist, daß einmal die Basis komplett ausgetauscht werden muß, damit alle "künstlichen" Variablen s aus der Basis entfernt werden.

Das *composite Simplex*-Verfahren arbeitet deshalb direkt auf den Variablen des Ausgangs-LPs und modifiziert das LP dynamisch zur jeweils aktuellen Basis [100]. Sei $S = (B, N)$ eine beliebige Basis von (1.91). Dann definiert man das zugehörige Phase-1-Problem als

$$\begin{array}{lrcrcl} \min & c(B)^T x_B & & & & \\ \text{s.t.} & A_B x_B & + & A_N x_N & = & b \\ & x_B & & & \geq & 0 \\ & & & x_N & \geq & 0, \end{array} \tag{1.95}$$

wobei $c(B)_i = -x_{B_i}$ für $x_{B_i} < 0$ und $c(B)_i = 0$ sonst ist. Dieses Verfahren hat sich als effizient erwiesen und kommt in kommerziellen Implementierungen zum Einsatz (z.B. CPLEX [16]). Trotzdem erscheint es noch verbesserungsbedürftig, denn dieses Phase-1-Problem ist ausschließlich mit Blick auf das Erreichen der Zulässigkeit konstruiert, berücksichtigt aber nicht die Zielfunktion.

Bei SoPlex wird daher einen andere Strategie verfolgt, die auch in der Phase-1 die Zielfunktion so gut wie möglich mitberücksichtigt. Außerdem werden wieder keine künstlichen Variablen dem LP hinzugefügt. Es sei $S = (B, N)$ eine beliebige Basis von (1.91). Als Phase-1-Problem wird das LP

$$\begin{array}{lrcrcl} \min & c_B^T x_B & + & c_N^T x_N & & \\ \text{s.t.} & A_B x_B & + & A_N x_N & = & b \\ & x_B & & & \geq & l \\ & & & x_N & \geq & 0, \end{array} \tag{1.96}$$

verwendet, wobei $l_i = x_{B_i}$ für $x_{B_i} < 0$ und $l_i = 0$ sonst ist. Dieses LP behält somit soviel von der Struktur des Ausgangs-LPs bei, wie es die Basis S zuläßt. Außerdem wird die Schranke l im Laufe der Phase-1 weiter hochgesetzt, sofern es die jeweilige Basis zuläßt. Dadurch kommt es nicht selten vor, daß ein LP bereits nach der Phase 1 gelöst ist.

Per Konstruktion ist S für das LP (1.96) zulässig, so daß der primale Simplex gestartet werden kann. Terminiert er mit einer Optimallösung von (1.96), so ist die optimale Basis für (1.91) immer noch dual zulässig, und es kann der duale Simplex gestartet werden. Andernfalls ist mit (1.96) auch das Ausgangs-LP unbeschränkt. Entsprechend kann auch ein Phase-1-Problem durch Manipulation der Zielfunktion konstruiert werden, für das die Basis S dual zulässig ist. In diesem Fall wird zunächst der duale Simplex verwendet.

Eine Implementierung für allgemeine Basen gemäß Abschnitt 1.2.4 gestaltet sich besonders einfach. Es muß lediglich eine andere Initialisierung der Schranken erfolgen und zwischen beiden Algorithmustypen umgeschaltet werden.

1.5 Kreiseln und dessen Vermeidung

Wenn der primale oder duale Algorithmus eine unendliche Folge degenerierter Pivot-Schritte durchführt, spricht man vom *Kreiseln*. Das ist die einzige Möglichkeit dafür, daß ein Simplex-Algorithmus nicht terminiert. Dies sieht man wie folgt: Es gibt maximal $\binom{n}{k}$ verschiedene Basen zum LP (1.1). Bei jedem nicht degenerierten Pivot-Schritt ändert sich der Zielfunktionswert des Basislösungsvektors streng monoton fallend für den primalen und streng monoton steigend für den dualen Algorithmus. Demnach kann die alte Basis nie wieder angenommen werden, so daß maximal $\binom{n}{k} - 1$ nicht degenerierte Pivot-Schritte möglich sind. Damit ist folgender Satz bewiesen:

SATZ 30

Wenn ein Simplex-Algorithmus nicht terminiert, kreiselt er.

Die Vermeidung des Kreiselns ist somit eine wichtige Anforderung für die Brauchbarkeit von Simplex-Algorithmen. In [26] wird das Kreiseln zwar als ein rares Phänomen beschrieben, das in den meisten Implementierungen aufgrund numerischer Fehler nicht auftrete und somit nicht berücksichtigt zu werden brauche. Heute werden Simplex-Algorithmen jedoch vielfach für kombinatorische Optimierungsprobleme eingesetzt, bei denen die Nebenbedingungsmatrix oft nur aus Werten 0, 1 und -1 besteht und das LP sowohl primal als auch dual degeneriert ist. Die dabei auftretenden Basismatrizen sind häufig so gut konditioniert, daß die Degeneriertheit auch numerisch erhalten bleibt. Deshalb kann das Kreiselproblem nicht mehr unberücksichtigt bleiben.

Es gibt eine Reihe theoretischer Ansätze [17, 99, 94, 103], die über spezielle Pricing- und Quotiententestverfahren ein Kreiseln ausschließen. In praktischen Implementierungen werden sie jedoch kaum eingesetzt. Ein Grund dafür ist, daß eine strikte Verwendung solcher

Pivot-Verfahren in der Regel zu wesentlich mehr Iterationen führt. Dies kann z.B. dadurch umgangen werden, daß, solange ein Fortschritt bei der Lösung erzielt wird, eine bessere Pivot-Strategie verwendet und erst, wenn mehrere degenerierte Schritte nacheinander ausgeführt worden sind, zeitweise auf eine Kreiselvermeidungsstrategie umgeschaltet wird [80].

Ein anderer Grund, theoretische Kreiselvermeidungsverfahren nicht in praktischen Implementierungen einzuseztzen, ist, daß sie bei numerisch stabilen Implementierung nicht mehr anwendbar sind. Sie basieren meist darauf, daß die Basislösung während des Kreiselns unverändert bleibt. Wie in Abschnitt 1.3 beschrieben wurde, trifft diese Voraussetzung bei numerisch stabilen Implementierungen auch im degenerierten Fall nicht immer zu, da eine leichte Unzulässigkeit toleriert wird. Dadurch ist es möglich, daß ein "numerisches Kreiseln" mit leicht variierenden Lösungsvektoren und schwankendem Zielfunktionswert entsteht. Z.B. kann bei Algorithmus 1 eine Verschlechterung des Zielfunktionswertes auftreten, wenn eine Variable $d_i - \delta < s_i < d_i$ die Basis verläßt. Dies kann durch sog. *Shifting* verhindert werden, bei dem die Schranke d_i z.B. auf s_i herabsetzt wird.

Das Shifting bildet die Grundlage für die Kreiselvermeidung von SoPlex, die nun anhand von Algorithmus 1 beschrieben werden soll. Sie ähnelt der in [12] vorgeschlagenen Methode, die bei MPSX zum Einsatz kam. Nach `maxcycle` Pivot-Schritten ohne Fortschritt wird beim nächsten Pivot-Schritt einen Fortschritt erzwungen, indem für alle i, mit $s_i < d_i + \delta$ und $\Delta s_i < 0$, die Schranke d_i auf $d_i \leftarrow s_i - \text{rand}(100\delta, 1000\delta)$ geshiftet wird. Dabei ist $\text{rand}(100\delta, 1000\delta)$ ein zufällig aus dem Intervall $(100\delta, 1000\delta)$ gewählter Wert. Für den Parameter `maxcycle` hat sich ein Wert in der Größenordung 150 bewährt (vgl. Abschnitt 4.2.8).

Der Zweck der zufälligen Wahl von d_i ist, die Degeneriertheit auch für zukünftige Pivot-Schritte aufzubrechen. Andere Verfahren perturbieren deshalb das gesamte LP, eventuell sogar gleich zu Beginn. Dies erscheint jedoch eher ungünstig, da so meist mehr Simplex-Iterationen notwendig werden als ohne Perturbation: Oft hat man nämlich Glück und der Simplex-Algorithmus verläßt eine degenerierte Ecke ohne weiteres Zutun. Beim gestörten LP ist eine solche Ecke in eine Schar von Ecken aufgebrochen, die der Simplex-Algorithmus nun einzeln traversieren muß. Entsprechend reicht es völlig aus, wenn der Simplex-Algorithmus *eine* optimale Basis findet — im gestörten Fall ist dies zwangsläufig auch die einzige und somit schwerer zu finden.

Handelt es sich um ein Phase-1-Problem, so ist das Shiften des LPs unproblematisch, handelt es sich doch ohnehin schon um ein LP mit veränderten Schranken. Andernfalls muß nach Terminierung mit dem jeweils dualen Algorithmus fortgefahren werden. Dies wird so lange iteriert, bis ein Algorithmus auf dem unveränderten LP terminiert.

Es gibt keinen Terminationsbeweis für eine solche Kreiselvermeidungsstrategie. In der Praxis sind jedoch noch keine Fälle aufgetreten, bei denen der Algorithmus ad infinitum zwischen den Phasen umschaltet, weil das LP immer wieder verändert wurde. Dies kann dadurch weiter erschwert werden, indem man die Anzahl `maxcycle` degenerierter Pivot-

Schritte vor dem Shiften bei jedem Umschalten hochsetzt.

Ein weiterer Vorteil von Kreiselvermeidungsstrategien mittels Shifting gegenüber dem Umschalten auf theoretische Strategien ist, daß die gewählte Pricing-Strategie beibehalten werden kann. Außerdem wird ein stärker modularisierter Softwareentwurf ermöglicht.

Neben den Arbeiten zur theoretischen Kreiselvermeidung gibt es kaum Literatur über praktikable Strategien. In [50] beschreiben Gill, Murray, Sauders und Wright eine Strategie, bei der der Toleranzparameter δ beim Harris-Quotiententest mit jedem Pivot-Schritt hochgesetzt wird. Wenn anschließend in der zweiten Phase des Quotiententests kein Fortschritt erzielt wird, shiften sie die Schranke um den aktuellen Wert δ, und da δ größer ist als in der vorigen Iteration, wird so ein Fortschritt erzwungen.

Zwei Nachteile sind bei diesem Verfahren gegenüber dem zuvor beschriebenen zu verzeichnen. Zum einen wächst der Toleranzparameter δ mit jeder Iteration. Dies entspricht aber einer gleichzeitigen Relaxierung *aller* Schranken, auch wenn dies nicht notwendig wäre. Insbesondere ist nach Terminierung die Lösung bis auf das gerade aktuelle δ zulässig, und in aller Regel muß eine weitere Phase angeschlossen werden. Zum anderen führen degenerierte Pivot-Schritte i.a. weitere Variablen an die um δ relaxierte Schranke heran. Dadurch schränkt man die Wahlfreiheit beim numerischen Quotiententest wieder unnötig ein. Insbesondere ist die Erhöhung von δ essentiell für eine numerisch stabile Implementierung. In [57] wird schließlich eine Gegenbeispiel angegeben, für das diese Kreiselvermeidung nicht funktioniert. Die Konstruktion eines solchen Gegenbeispiels scheitert jedoch für randomisierte Ansätze, wie sie für SoPlex zugrundegelegt wurden.

1.6 Pricing-Strategien

Bei der Aufstellung der Simplex-Algorithmen wurde beim jeweiligen Pricing-Schritt lediglich die Bedingung für wählbare Indizes genannt. Somit sind viele Varianten von Algorithmen möglich, die sich in der konkreten Auswahl, der sog. *Pricing-Strategie*, unterscheiden. Dabei ist gerade sie von entscheidender Bedeutung für die Anzahl von Iterationen und somit für die Effizienz von Simplex-Algorithmen. Es ist jedoch keine Strategie bekannt, die für alle LP-Probleme am effizientesten arbeitet. Deshalb werden nun eine Reihe von Pricing-Strategien vorgestellt.

Der Einfachheit halber gehen wir dazu auf das LP 1.1 und die Algorithmen 1 und 2 zurück. Die Erweiterung auf die Algorithmen mit allgemeiner Basisdarstellung ist problemlos möglich, erfordert jedoch eine kompliziertere Notation, die die zugrundeliegenden Ideen nur verschleiern würde.

1.6.1 Most-Violation Pricing

Dies ist das Verfahren, das ursprünglich von Dantzig vorgeschlagen wurde. Beim entfernenden Algorithmus wird

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

und beim einfügenden Algorithmus

$$q = \arg\min\{g_i - d_i : g_i < d_i\}$$

gewählt. Für einen dualen Algorithmus bedeutet dies, daß die am stärksten verletzte Ungleichung für den Pivot-Schritt herangezogen wird. Beim primalen hingegen wird der Index mit den niedrigsten reduzierten Kosten gewählt. Deshalb ist dieses Verfahren auch als reduced-cost Pricing bekannt.

1.6.2 Partial Pricing

Das partial Pricing stammt aus der Zeit, in der Computer nur über wenige Kilobyte an Hauptspeicher verfügten. Um dennoch größere Probleme lösen zu können, wurde ein sog. out-of-core-Ansatz verfolgt. Dabei wird jeweils nur ein Teil der Vektoren der Bedingungsmatrix im Hauptspeicher gehalten. Wenn dieses Teil-LP gelöst ist, werden die nichtbasischen Vektoren durch andere aus dem Massenspeicher ersetzt. Dies wird so lange iteriert, bis das gesamte LP „ausgepriced“ und das Gesamt-LP gelöst ist. Als Pricing-Strategie wird für die Teil-LPs meist das most-violated Pricing eingesetzt.

Auch ohne Hauptspeicherzwänge hat diese Pricing-Strategie ihre Bedeutung zur Reduktion des Rechenaufwands. Statt tatsächlich einen Teil des LPs auszulagern, beschränkt man das Pricing über eine gewisse Anzahl von Iterationen auf eine Teilmenge der Nichtbasisvektoren. Dementsprechend muß auch der Pricing-Vektor nur für diese Teilmenge von Indizes berechnet werden, wodurch der Rechenaufwand mitunter stark reduziert werden kann. Insbesonders für LPs mit $m \gg n$ führt dies zu erheblichen Beschleunigungen.

Das partial Pricing kann nur für einfügende Algorithmen eingesetzt werden, da für entfernende Algorithmen stets der gesamte Vektor Δg berechnet werden muß. Damit gewinnt die Wahlmöglichkeit zwischen einer zeilenweisen und einer spaltenweisen Darstellung der Basis zusätzlich an Bedeutung, denn sie ermöglicht es, auch den dualen als einfügenden Algorithmus zu verwenden und dadurch das partielle Pricing einzusetzen.

1.6.3 Multiple Pricing

Auch das multiple Pricing zielt auf eine Reduzierung des Rechnenaufwandes für die Matrix-Vektor-Multiplikation beim einfügenden Algorithmus. Es basiert auf der Beobachtung, daß ein beim Pricing wählbarer Index mit hoher Wahrscheinlichkeit auch in der nächsten Iteration wählbar bleibt.

Zunächst wird ein Satz von wählbaren Indizes, sog. Kandidaten, bestimmt. Von ihnen wir einer (meist nach der most-violated Pricing-Strategie) für den nächsten Pivot-Schritt benutzt. In den folgenden Pivot-Schritten wird das Pricing auf die zuvor gewählte Kandidatenmenge beschränkt. Dies reduziert wieder den Rechenaufwand zur Berechnung von g.

1.6.4 Partial multiple Pricing

Eine wichtige Pricing-Strategie ergibt sich aus der Kombination der beiden zuletzt genannten Methoden. Es ist das partial multiple Pricing, bei dem zur Auswahl der Kandidatenmenge nur ein Teil der Matrix benutzt wird. Es ist dabei besonders günstig, wenn man bei jedem Pricing-Schritt die Kandidatenmenge wie folgt neu aufbaut. Man nimmt alle noch wählbaren Indizes der alten Kandidatenmenge und durchsucht einen Teil der Restmatrix nach weiteren wählbaren Indizes. Diese werden der Kandidatenmenge zugefügt.

Mit diesem Verfahren wird schneller als beim multiple Pricing die gesamte Nebenbedingungsmatrix durchsucht, was in der Regel zu einer geringeren Iterationszahl führt. Der Rechenaufwand scheint etwas höher als beim partial oder multiple Pricing zu sein, was man jedoch durch Anpassung der Größe der Kandidatenmenge und der Teile kompensieren kann.

1.6.5 Steepest-edge Pricing

Das steepest-edge Pricing-Verfahren begründet sich am besten aus der geometrischen Interpretation des Pricings beim primalen Simplex (vgl. Abb. 1.7). Betrachte für das LP

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Dx & \geq \quad d, \end{array} \tag{1.1}$$

den primalen Algorithmus in Zeilendarstellung (Algorithmus 1). Sei $S = (P, Q)$ eine primal zulässige Basis. Beim Pricing soll aus mehreren Richtungen $\Delta x^{(i)} = D_{P\cdot}^{-1} e_i$, entlang denen eine Verbesserung möglich wäre, eine möglichst günstige gewählt werden. Die steepest-edge Pricing-Strategie wählt diejenige aus, die im Sinne des Zielfunktionsvektors am steilsten ist. Dazu werden die Werte $f_i^2 = \|c^T \Delta x^{(i)}\|$ durch die Norm

$$\rho_i = \|\Delta x^{(i)}\|^2 \tag{1.97}$$

geteilt, und unter diesen Zahlen wird das Maximum ausgewählt:

$$p = \arg\max \left\{ \frac{f_i^2}{\rho_i} : f_i < 0 \right\} \tag{1.98}$$

Die Werte $\frac{f_i^2}{\rho_i}$ sind proportional zum Quadrat des Cosinus des Winkels zwischen Zielfunktionsvektor und möglicher Fortschrittsrichtung $\Delta x^{(i)}$ und geben somit die Steilheit an.

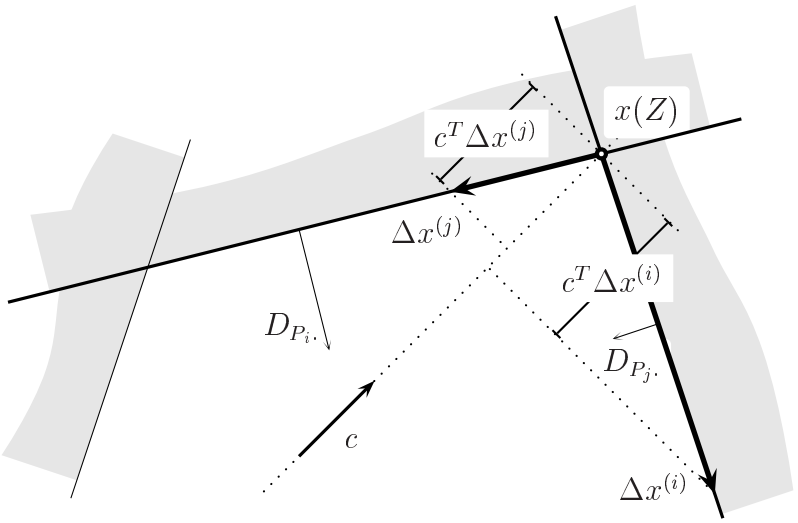


Abbildung 1.7: Das steepest-edge-Pricing verhindert, daß aufgrund unterschiedlicher Normierungen der Richtungsvektoren $\Delta x^{(i)}$ eine weniger steile Richtung gewählt wird. Diese Situation ist in der Abbildung dargestellt. Obwohl $\Delta x^{(j)}$ einen geringeren Winkel zum Zielfunktionsvektor $-c$ aufweist als $\Delta x^{(i)}$, sind die zugehörigen reduzierten Kosten höher.

Die Berechnung der Werte ρ_i nach (1.97) erforderte einen hohen Rechenaufwand. Für jeden Wert müßte ein lineares Gleichungssystem zur Bestimmung von $\Delta x^{(i)}$ gelöst und darüberhinaus ein Skalarprodukt berechnet werden. Aus diesem Grund galt das steepest-edge Pricing lange Zeit als zu aufwendig, obwohl bekannt war, daß es zu einer erheblichen Reduktion der Anzahl der Pivot-Schritte führen kann. Erst als Goldfarb und Reid Update-Formeln für die Größen ρ_i formulierten, wurde es zu einem praktikablen Verfahren [52, 47].

Es werden nun die Update-Formeln für ρ_i für den primalen Simplex in Zeilendarstellung aufgestellt. Diese Darstellung erlaubt ein besseres geometrisches Verständnis als der in [52] beschrittene Weg.

Sei $Z' = (P', Q')$ die Basis nach einem Pivot-Schritt mit Indizes p und q. Nach (1.12) ist die neue Basismatrix $D_{P'.} = VD_{P.}$, mit $V = I + e_p(D_{q.}D_{P.}^{-1} - e_p^T)$ und nach (1.89) gilt $V^{-1} = I - \frac{e_p}{(D_{q.}D_{P.}^{-1})_p}(D_{q.}D_{P.}^{-1} - e_p^T)$. Sei $\Delta x^{(i)} = D_{P.}^{-1}e_i$. Dann gilt mit $\Delta' x^{(i)} = D_{P'.}^{-1}e_i$ und $t^T = D_{q.}D_{P.}^{-1}$:

$$
\begin{aligned}
\Delta' x^{(i)} &= D_{P.}^{-1}V^{-1}e_i \\
&= D_{P.}^{-1}(e_i - \tfrac{e_p}{t_p}(t^T - e_p^T)e_i) \\
&= \begin{cases} D_{P.}^{-1}(e_i - \frac{t_i}{t_p}e_p) & = \Delta x^{(i)} - \frac{t_i}{t_p}\cdot\Delta x^{(p)} \quad , \text{für } i \neq p, \\ D_{P.}^{-1}(e_p - \frac{e_p}{t_p}(t^T - e_p^T)e_p) & = \frac{1}{t_p}\Delta x^{(p)} \quad , \text{sonst.} \end{cases}
\end{aligned}
$$

Daraus ergibt sich

$$\rho'_p = \frac{1}{t_p^2}\rho_p \tag{1.99}$$

und für $i \neq p$ mit $\Delta x = \Delta x^{(p)}$

$$\begin{aligned} \rho'_i &= (\Delta' x^{(i)})^T (\Delta' x^{(i)}) \\ &= \rho_i + 2\tfrac{t_i}{t_p} \Delta x^T D_{P.}^{-1} e_i + \left(\tfrac{t_i}{t_p}\right)^2 \rho_p. \end{aligned} \tag{1.100}$$

Also muß für das Aktualisieren der Multiplikatoren ρ_i bei einem Basistausch ein zusätzliches Gleichungssystem, nämlich $\xi = \Delta x^T D_{P.}^{-1}$, gelöst werden. Für den allgemeinen entfernenden Algorithmus wird Δx durch Δh und t durch Δf ersetzt.

Um die entsprechenden Formeln für den einfügenden Algorithmus aufzustellen, betrachten wir wieder den Spezialfall (1.29) und gehen zurück auf die Form (1.30) der Basismatrix. Damit kann $t = \Delta x$ berechnet werden als $\Delta x_N = e_p$ und $\Delta x_B = -A_{.B}^{-1} A_{.p} = -\Delta f$, und für ξ gilt $\xi_B^T = \Delta f^T A_{.B}^{-1}$ und $\xi_N^T = e_p^T - \xi_B^T A_{.N}$.

Für den einfügenden Algorithmus mit Spaltenbasis bezeichne $\bar{q}$ den eintretenden und $\bar{p}$ den aus der Basis austretenden Index. Diese entsprechen beim entfernenden Algorithmus mit Zeilenbasis $\bar{q} = P_p$ und $B_{\bar{p}} = q$. Deshalb ist $D_{q.} = e_{B_{\bar{p}}}$, woraus für t mit (1.30) $t_B^T = e_{\bar{p}}^T A_{.B}^{-1} = \Delta h^T$ und $t_N^T = -e_p^T A_{.B}^{-1} A_{.N} = \Delta g_N^T$ folgt.

Für $i \in N \setminus \{\bar{q}\}$ ist deshalb nach (1.100)

$$\begin{aligned} \rho'_i &= \rho_i + 2\tfrac{\Delta g_i}{\Delta g_{\bar{q}}} \xi_i + \left(\tfrac{\Delta g_i}{\Delta g_{\bar{q}}}\right)^2 \rho_{\bar{q}} \\ &= \rho_i - 2\tfrac{\Delta g_i}{\Delta g_{\bar{q}}} \xi_B A_{.i} + \left(\tfrac{\Delta g_i}{\Delta g_{\bar{q}}}\right)^2 \rho_{\bar{q}} \end{aligned} \tag{1.101}$$

und

$$\rho'_{\bar{q}} = \frac{\rho_{\bar{q}}}{\Delta h_{\bar{q}}^2}. \tag{1.102}$$

Der zusätzliche Rechenaufwand für steepest-edge Pricing beim einfügenden Simplex ist also die Lösung eines Gleichungssystems für ξ_B und die Berechnung der Skalarprodukte $\xi_B^T A_{.i}$, sofern $\Delta g_i \neq 0$ ist.

Obwohl die Update-Formeln wegen der besseren geometrischen Anschauung anhand der primalen Algorithmen entwickelt wurden, können sie ebenso für die dualen Algorithmen eingesetzt werden. Dies gilt auch für die Algorithmen mit allgemeiner Basis. Dabei müssen die Werte ρ_i jedoch nicht aktualisiert werden, wenn kein Basistausch erfolgt, weil eine Variable von ihrer einen auf ihre andere Schranke verschoben wurde.

1.6.6 Devex Pricing

Beim Devex Pricing handelt es sich um eine Approximation des steepest-edge Pricings, bei der ein möglichst geringer Rechenaufwand anfällt, indem die Multiplikatoren ρ_i nicht exakt bestimmt werden. Es wurde von P. Harris vorgeschlagen [60].

Für den einfügenden Algorithmus wird für $i \neq \bar{q}$ anstelle von (1.101) folgende Update-Formel verwendet[5]:

$$\rho_i' = \rho_i + \left(\frac{\Delta f_i}{\Delta f_{\bar{q}}}\right)^2 \rho_{\bar{q}}; \tag{1.103}$$

für $i = \bar{q}$ kann weiter (1.102) verwendet werden. Man erhält (1.103) durch einfaches Weglassen des Summanden in (1.101), der die Lösung des zusätzlichen Gleichungssystems und die Berechnung der Skalarprodukte erfordern würde. Entsprechend erhält man für den entfernenden Simplex für $i \neq p$ die Update-Formel

$$\rho_i' = \rho_i + \left(\frac{t_i}{t_p}\right)^2 \rho_p^2, \tag{1.104}$$

wobei für $i = p$ wieder (1.99) verwendet wird.

Die Update-Formeln des Devex Pricings führen dazu, daß die Gewichte ρ_i stetig anwachsen. Um dieses Wachstum zu begrenzen, werden sie auf 1 zurückgesetzt, sobald ein Gewicht einen Schwellwert überschreitet. Es hat sich ein Schwellwert von 10^6 bewährt.

1.6.7 Weighted Pricing

Wie das steepest-edge oder Devex Pricing, benutzt auch das weighted Pricing Präferenzparameter ρ_i für die Pivot-Auswahl. Im Gegensatz zu den erstgenannten werden diese statisch beim Start des Simplex-Algorithmus festgesetzt und nicht bei jeder Simplex-Iteration aktualisiert. Solch ein Verfahren eignet sich, wenn man über eine gute Heuristik verfügt, die Informationen dazu liefern kann, welche Bedingungen bei der optimalen Lösung mit Gleichheit erfüllt sein wird. Es können aber auch Gewichte für allgemeine LPs konstruiert werden, wie dies in Abschnitt 1.8.6 auch für die Konstruktion einer Startbasis beschrieben wird.

[5] Von Harris wurde ursprünglich

$$\rho_i' = \max\left\{\rho_i, \left(\frac{\Delta f_i}{\Delta f_{\bar{q}}}\right)^2 \rho_{\bar{q}}\right\}.$$

vorgeschlagen. Diese Approximation erscheint jedoch noch gröber als (1.103) und hat sich auch bei Tests mit SoPlex als die schlechtere erwiesen.

1.6.8 Hybrid Pricing

Beim hybriden Pricing versucht man, abhängig vom zu lösenden LP, der Basisdarstellung und des Simplex-Typs den jeweils besten Pricer automatisch einzusetzen. In einer vorläufigen Implementierung für SoPlex wird beim entfernenden Algorithmus das steepest-edge Pricing eingesetzt. Beim einfügenden Simplex wird das partial multiple Pricing verwendet, wenn die Dimension der Basismatrix klein gegen die Anzahl der sonstigen Vektoren ist, andernfalls kommt das Devex Pricing zum Einsatz. Besser wäre jedoch ein Verfahren, das die Strategieauswahl dynamisch anhand von Laufzeitdaten durchführt.

1.7 Lösung linearer Gleichungssysteme mit der Basismatrix

Dantzig formulierte den Simplex-Algorithmus mit sog. Tableaus. In dieser Formulierung bedarf es keiner expliziten Lösung von Gleichungssystemen mit der Basismatrix, denn ein Simplex-Tableau besteht i.w. aus dem Produkt aus der Inversen der Basismatrix und der *gesamten* Nebenbedingungsmatrix. Nach jedem Basistausch wird deshalb das gesamte Simplex-Tableau durch Multiplikation mit V^{-1} nach (1.89) aktualisiert.

Dieses Vorgehen hat zwei Nachteile gegenüber dem hier vorgestellten revidierten Simplex. Zum einen akkumulieren sich numerische Fehler, die auch dann nicht behoben werden, wenn sich die Kondition der Basismatrix wieder verbessert. Zum anderen benötigt der Simplex-Algorithmus in Tableauform wesentlich mehr Rechenoperationen und Speicherplatz.

Für den revidierten Simplex-Algorithmus stellt sich also das Problem, wie die Lösung von Gleichungssystemen am besten vorgenommen werden sollte. Basismatrizen beim Simplex-Algorithmus enthalten typischerweise einen großen Prozentsatz von Einheitsvektoren. Diese stammen bei einer spaltenweisen Darstellung von Schlupfvariablen und bei einer Zeilenbasis von den Schranken der Variablen. Die anderen Spalten oder Zeilen der Basismatrix sind Spalten oder Zeilen der Nebenbedingungsmatrix und enthalten damit i.a. auch nur wenige von Null verschiedene Elemente (NNEs).

Basismatrizen sind also i.a. nicht symmetrisch und *dünnbesetzt*, was so viel heißt wie, „es lohnt sich aus Geschwindigkeitsgründen, beim gegebenen numerischen Problem nur die NNEs der Matrix statt eines zweidimensionalen Feldes zu speichern“ (dichtbesetzt). Offenbar handelt es sich hierbei um keine strenge Definition. Vielmehr hängt die Dünnbesetztheit einer Matrix von der gewählten Rechnerarchitektur[6] und den gewählten Datentypen zur

[6]Ein Vektor- oder Parallelrechner kann linear abgespeicherte Daten wesentlich schneller berechnen als bei einem unstrukturierten Zugriff auf den Speicher. Dementsprechend kann für derartige Architekturen ein Algorithmus auf dichten Daten effizienter ausgeführt werden als ein Algorithmus, der auf dünnen Daten operiert und weniger Operationen ausführt.

Repräsentation der NNEs einer Matrix ab. Heute aufgestellte LPs sind fast immer dünn besetzt, unabhängig von der Rechnerarchitektur.

Es gibt zwei Ansätze zur numerischen Lösung von linearen Gleichungssystemen nämlich *direkte* und *iterative* Löser. Wie in Abschnitt 1.7.1 aufgezeigt wird, erscheinen letztere auf den ersten Blick vielversprechend, scheiden jedoch aus, da ihre Konvergenz nicht genügend abgesichert werden kann. Deshalb werden bei Implementierungen des Simplex-Algorithmus direkte Methoden eingesetzt. Sie werden in den Abschnitten 1.7.2 und 1.7.3 diskutiert. In letzterem wird auch die für SoPlex implementierte Version einer LU-Zerlegung der Basismatrix beschrieben. In Abschnitt 1.7.4 wird die Lösung von Gleichungssystemen bei gegebener LU-Zerlegung der Matrix beschrieben. Aus Effizienzgründen ist es angezeigt, eine LU-Zerlegung der Basismatrix für eine Folge von Pivot-Schritten zu nutzen. Zwei Methoden, dies zu tun, werden in Abschnitt 1.7.5 vorgestellt.

1.7.1 Iterative Löser

Iterative Löser für lineare Gleichungssysteme verbessern iterativ (daher der Name) eine Approximation des Lösungsvektors bis die gewünschte Genauigkeit erreicht wird. In jede Iteration geht meist nur die Matrix, der Vektor der rechten Seite und die aktuelle Approximation des Lösungsvektors ein, wodurch der Speicherbedarf im Gegensatz zu direkten Lösern für dünnbesetzte Matrizen beschränkt bleibt.

Für die Implementierung von Simplex-Algorithmen erscheint ein weiterer Punkt vorteilhaft. Gibt es etwa beim dualen Simplex noch „sehr stark“ verletzte Ungleichungen, so erkennt man diese auch, wenn die Genauigkeit des aktuellen Basislösungsvektors noch gering ist. Mit Fortschreiten der Lösung des LPs kann die Genauigkeit der Lösungsvektoren angepaßt werden. Dabei kann der Lösungsvektor der vorigen Simplex-Iteration als Startwert für die iterative Lösung des Gleichungssystems für den neuen Basislösungsvektor benutzt werden. Ferner bieten iterative Löser bessere Ansätze zur Parallelisierung als die Vorwärts- und Rückwärtssubstitution, da ihnen weniger Sequentialität inhärent ist [83, 98, 66]. Grund genug also, die Anwendbarkeit iterativer Gleichungssystemslöser für den Simplex-Algorithmus zu untersuchen. Dazu geben wir eine kurze Einführung in die Materie. Für einen umfassenderen Überblick sei z.B. auf [34, 6] verwiesen.

Es gibt zwei Ansätze für iterative Gleichungssystemlöser. Klassische Iterationsverfahren basieren auf einer Fixpunktiteration

$$x_{k+1} = Gx_k + g, \text{ mit } G = I - Q^{-1}B \text{ und } g = Q^{-1}b, \tag{1.105}$$

wobei Q^{-1} eine (reguläre) Approximation von B^{-1} ist. Offenbar ist nach Konvergenz $x = x_{k+1} = x_k$ die Lösung des Gleichungssystems, denn aus $x = Gx + g = x - Q^{-1}(Bx - b)$ folgt $Bx = b$. Verschiedene Iterationsverfahren unterscheiden sich in der Wahl von Q. Schon beim einfachen Fall symmetrischer positiv definiter Matrizen B konvergieren derartige Iterationsverfahren jedoch nur falls der Spektralradius $\rho(G) < 1$ ist. Somit scheiden sie

für die Anwendung in Simplex-Algorithmen aus, denn für allgemeine Basismatrizen kann keine Konvergenz garantiert werden.[7]

Modernere Iterationsverfahren sind Varianten des Verfahrens der konjugierten Gradienten. Ihr Vorteil ist, daß für beliebige symmetrisch positiv definite Matrizen die Konvergenz nachgewiesen werden kann. Die Konvergenzgeschwindigkeit hängt jedoch von der Kondition der Matrix B ab [34]:

$$(x - x_k)^T B(x - x_k) \leq 2 \left(\frac{\sqrt{\kappa(B)} - 1}{\sqrt{\kappa(B)} + 1} \right)^k (x - x_0)^T B(x - x_0). \tag{1.106}$$

Erweiterungen von Konjugierte-Gradienten-Methoden auf unsymmetrische Matrizen basieren auf der Symmetrisierung des Gleichungssystems zu

$$B^T B x = B^T b. \tag{1.107}$$

Dies verschlechtert jedoch die Konvergenzgeschwindigkeit, denn $\kappa(B^T B) = \kappa(B)^2$.

Eine Verbesserung der Konvergenzgeschwindigkeit von Konjugierten-Gradienten-Methoden besteht in der sog. Vorkonditionierung. Dazu wird die Matrix B von beiden Seiten mit einer Vorkonditionierungsmatrix C bzw. C^T multipliziert. Die Matrix C wird so gewählt, daß mit ihr die Gleichungssysteme direkt lösbar sind, und die Kondition von CBC^T klein ist. Für Gleichungssysteme, die aufgrund ihrer Herkunft (etwa der Lösung partieller Differentialgleichungen) eine vorhersagbare Struktur haben, können wirkungsvolle problemspezifische Vorkonditionierer entwickelt werden.

Für den Einsatz in einem allgemeinen Simplex-Algorithmus bedarf es jedoch eines verläßlichen allgemeingültigen Vorkonditionierers. Dieser entspräche aber einer Approximation eines direkten Lösers. Damit sollte, zumal in der Praxis bewährt, ein vollständiger direkter Löser den Vorzug erhalten.

Auch wenn iterative Löser für Simplex-Algorithmen ausscheiden, könnten sie dennoch für die Lineare Programmierung von Nutzen sein. Der Hauptaufwand innere-Punkte-Verfahren besteht in der Lösung von Gleichungssystemen mit der Matrix $A^T A$, wobei A die Nebenbedingungsmatrix bezeichnet. Diese ist offenbar symmetrisch, und es bleibt zu untersuchen, ob damit genügend Struktur vorhanden ist, um einen geeigneten Vorkonditionierer zu definieren. Ein möglicher Vorteil bei der Anwendung iterativer Löser läge in der besseren Parallelisierbarkeit und in der Möglichkeit, *inexakte Methoden* zu realisieren, d.h. die Genauigkeit der Lösung von Gleichungssystemen den aktuellen Erfordernissen innerhalb des inneren-Punkte-Verfahrens anzupassen.

[7] Es gibt sogar eine Publikation [37], die eine Implementierung des Simplex-Algorithmus mit dem Gauß-Seidel-Verfahrens (Q ist die obere Dreiecksmatrix von B) beschreibt. Allerdings ist diese Implementierung eher ein Kuriosum als ein zuverlässiger Algorithmus.

1.7.2 Direkte Methoden

Bei direkten Lösungsverfahren wird die Matrix B in ein Produkt von Matrizen transformiert, mit denen Gleichungssysteme direkt gelöst werden können. Beim Simplex-Algorithmen kommen dabei zwei Typen von Matrizen zum Einsatz, Rang-1-Update-Matrizen und Dreiecksmatrizen. Andere direkte Lösungsmethoden, wie z.B. mit Givens-Rotationen oder Householder-Reflexionen [34] werden trotz besserer numerischer Eigenschaften aufgrund ihres höheren Rechenaufwands und der schlechteren Ausnutzung der Besetzungstruktur nicht verwendet.

Rang-1-Update-Matrizen haben die Form

$$V = I + e_l(\eta^T - e_l^T) \tag{1.108}$$

und wurden bereits für den Basistausch (vgl. (1.12)) eingeführt. Die Inverse solcher Matrizen kann nach (1.89) direkt angegeben werden, so daß sie sich zur Konstruktion direkter Lösungsverfahren eigenen. Die linksseitige Multiplikation einer Matrix B mit V ergibt die Matrix $B' = VB = B + e_l(\eta^T B - B_{l.})$, die aus B durch Austausch der l-ten Zeile mit $\eta^T B$ resultiert.

Historisch wurde für den revidierten Simplex-Algorithmus zunächst die sog. Produktform der Inversen (PFI) verwendet [95, 63]. Bei der PFI wird die Einheitsmatrix durch das maximal n-fache Produkt von Rang-1-Update-Matrizen in die Matrix B transformiert:

$$B = V_n \cdot \cdots \cdot V_1 \cdot I.$$

Stimmen dabei einige Zeilen von B mit der Einheitsmatrix überein, so bedarf es entsprechend weniger Matrixmultiplikationen. Die Reihenfolge der ausgetauschten Zeilen ist dabei entscheidend für die numerische Stabilität und den Speicherverbrauch der PFI.

Beim Simplex-Algorithmus werden die Vektoren η ohnehin für die Updates berechnet, so daß die PFI als die für Simplex-Algorithmen natürliche Wahl erscheint. Im Laufe des Simplex-Algorithmus nimmt jedoch die numerische Stabilität der PFI ab und der benötigte Speicheraufwand zu. Deshalb muß die PFI von Zeit zu Zeit neu mit einer besseren Pivotreihenfolge berechnet werden.

Untersuchungen haben gezeigt, daß die PFI i.a. einen größeren Speicherbedarf aufweist als die im nächsten Abschnitt dargestellte LU-Zerlegung [11]. Da der Speicherbedarf etwa der Berechungszeit zur Lösung von Gleichungssystemen entspricht, verwenden heutige Implementierungen des Simplex-Algorithmus die LU-Zerlegung der Basismatrix. Dennoch bleibt auch die PFI weiterhin interessant, denn Rang-1-Update-Matrizen werden mitunter für Basistauschschritte verwendet (vgl. Abschnitt 1.7.5).

1.7.3 LU Zerlegung

Gegeben sei ein lineares Gleichungssystem der Form

$$Bx = b \tag{1.109}$$
$$\text{oder } x^T B = b^T, \tag{1.110}$$

wobei $B \in \mathbb{R}^{n \times n}$ und $x, b \in \mathbb{R}^n$. Wenn nichts weiter gesagt wird, seien im folgenden Matrizen und Vektoren entsprechender Dimensionen vorausgesetzt.

Bei der LU-Zerlegung (oder Faktorisierung) einer regulären Matrix B werden Permutationsmatrizen P und Q sowie reguläre Matrizen L und U bestimmt, derart daß

$$B = LU \tag{1.111}$$

gilt, wobei $\bar{L} = PLQ$ eine untere und $\bar{U} = PUQ$ eine obere Dreiecksmatrix ist. Anschließend können Gleichungssysteme der Form (1.109) mit folgenden Operationen gelöst werden:

1. Vorwärts-Substitution: Löse $Ly = b$
2. Rückwärts-Substitution: Löse $Ux = y$

Entsprechend löst man Gleichungssysteme der Form (1.110) mit:

1. Rückwärts-Substitution: Löse $y^T U = b^T$
2. Vorwärts-Substitution: Löse $x^T L = y$

Die Permutation von L und U wird direkt in den Substitutions-Algorithmus integriert.

Die LU-Zerlegung von B geschieht mit Gaußscher Elimination. Dabei wird die Matrix B einer Folge von Transformationen unterzogen, derart daß

$$B = B^1 \to B^2 \to \cdots \to B^n = U. \tag{1.112}$$

Eine Transformation wird *Pivot-Schritt* genannt[8]. Bei jedem Pivot-Schritt werden ein Spalte von L erzeugt und die Permutationsmatrizen aufgebaut. Die letzte Spalte von L ist ein Einheitsvektor.

Für jede Transformation $s = 1, \ldots, n-1$ wird zunächst ein NNE, etwa $B^s_{i^s j^s}$, der aktuellen Matrix, das sog. *Pivot-Element*, gewählt. Die Zeile i^s heißt die *Pivot-Zeile* und die Spalte j^s die *Pivot-Spalte* der s-ten Transformation. I^s respektive J^s bezeichnen die Menge der Zeilen- bzw. Spaltenindizes, aus denen bis zur s-ten Transformation noch kein Pivot-Element gewählt wurde. Somit gilt $I^1 = \{1, \ldots, n\} = J^1$. Schließlich werden die Permutationsmatrizen ausgehend von $P = Q = 0$ während der LU-Zerlegung aufgestellt.

[8] Es muß also zwischen Pivot-Schritten beim Simplex- und beim LU-Zerlegungs-Algorithmus unterschieden werden. Ursprünglich stammt die Bezeichnung Pivot-Schritt jedoch aus der LU-Zerlegung. Der Grund auch beim Simplex-Algorithmus von Pivot-Schritten zu sprechen, liegt bei seiner ursprünglichen Formulierung in Tableau-Form, wo die Iterationen eine größere Ähnlichkeit zu den Eliminations-Schritten bei der LU-Zerlegung aufweisen.

ALGORITHMUS 7 (LU-ZERLEGUNG)

Für $s = 1, \ldots, n$:

Schritt 1 (Pivot-Auswahl):
Wähle ein Pivot-Element $B^s_{i^s j^s} \neq 0$, mit $i^s \in I^s$ und $j^s \in J^s$.

Schritt 2 (Permutation):
Setze
$P_{i^s s} \leftarrow 1$ und
$Q_{s j^s} \leftarrow 1$

Schritt 3 (L-Loop):
Für $i \in I^s$:
Setze $L_{ij^s} \leftarrow B^s_{ij^s} / B^s_{i^s j^s}$

Schritt 4: Setze
$I^{s+1} \leftarrow I^s \setminus \{i^s\}$ und
$J^{s+1} \leftarrow J^s \setminus \{j^s\}$

Schritt 5 (Update-Loop):
Für $i \in I^{s+1}$, $j \in J^s$:
Setze $B^{s+1}_{ij} \leftarrow B^s_{ij} - L_{ij^s} \cdot B^s_{i^s j}$
Für $i \notin I^{s+1}$, $j \in \{1, \ldots, n\}$:
Setze $B^{s+1}_{ij} \leftarrow B^s_{ij}$

Bei Termination von Algorithmus 7 sind P und Q Permutationsmatrizen, und $\bar{U} = PB^nQ$ ist eine obere Dreiecksmatrix, während $\bar{L} = PLQ$ untere Dreiecksgestalt hat. Falls die Matrix singulär ist, schlägt bei einer Iteration Schritt 1 fehl.

Die Matrizen B^1 bis B^n werden nicht alle neu erzeugt. Stattdessen wird eine Arbeitsmatrix in jeder Iteration manipuliert; die jeweils vorige geht dabei verloren. Deshalb wird die zweite Schleife in Schritt 5, die lediglich alle Elemente, die nicht zur aktiven Submatrix gehören, in die nächste Matrix überträgt, nicht explizit durchgeführt. Dasselbe gilt auch für alle Matrixelemente, die z.B. wegen $L_{ij^s} = 0$ unverändert bleiben.

1.7.3.1 Numerische Aspekte bei der Pivot-Auswahl

Zur Untersuchung der numerischen Stabilität in Rückwärtsanalyse (vgl. Abschnitt 1.3) muß die Norm der Matrix

$$H = \tilde{L}\tilde{U} - B \tag{1.113}$$

abgeschätzt werden, wobei $\tilde{L}$ und $\tilde{U}$ die *numerisch* gewonnenen Faktoren bezeichnen. Für die Elemente von H kann folgende Abschätzung gewonnen werden

$$|H_{ij}| \leq 5.01 \varepsilon n \max_s \{|B^s_{ij}|\}, \tag{1.114}$$

wobei ε die Maschinengenauigkeit bezeichnet [39]. Somit muß das Wachstum der Matrixelemente während des Eliminationsprozesses kontrolliert werden.

Bei der Pivot-Auswahl bietet sich eine Möglichkeit, das Wachstum der Koeffizienten während des Eliminationsprozesses zu steuern. Offenbar führen kleine Werte $|B^s_{i^s j^s}|$ in Schritt 3 eher zu einem Anstieg der Koeffizienten als große. Verschieden stabile Varianten des LU-Zerlegungsalgorithmus unterscheiden sich daher darin, mit welchem Aufwand nach betragsgroßen Pivot-Elementen gesucht wird. Beste Stabilität bietet natürlich die sog. vollständige Pivot-Suche, bei der jeweils das betragsgrößte Element der aktiven Matrix $B^s_{I^s J^s}$ gewählt wird. Dessen Bestimmung bedeutet aber oft einen nicht akzeptablen Rechenaufwand, so daß meist weniger strenge Varianten zum Einsatz kommen. Oft wird das betragsgrößte Element einer Spalte oder Zeile gewählt (partielle Pivot-Suche). Wie im folgenden Abschnitt beschrieben, werden bei dünn besetzten Matrizen noch weniger strenge Anforderungen gestellt.

1.7.3.2 Pivot-Auswahl für dünnbesetzte Matrizen

Während eines Eliminationsschrittes kann in Schritt 5 ein Elemente $B^{s+1}_{ij} \neq 0$ werden, für das $B^s_{ij} = 0$ galt. Derartige Elemente heißen *Fillelemente*. Damit auch die Faktoren L und U dünnbesetzt bleiben, muß darauf geachtet werden, daß möglichst wenig Fillelemente entstehen.

Das folgende ist das Standardbeispiel für den entscheidenden Einfluß der Pivot-Auswahl auf die Anzahl von Fillelementen. Wählt man bei einer Matrix mit der Besetzungsstruktur

$$\begin{pmatrix} \bullet & \times & \dots & \times \\ \times & \times & & \\ \vdots & & \ddots & \\ \times & & & \times \end{pmatrix}$$

als Pivot-Element $\bullet$, so entsteht nach diesem Eliminationsschritt eine dicht besetzte Matrix. Wählt man hingegen ein anderes Diagonalelement, so entsteht kein weiteres Fillelement. Leider ist die Bestimmung der Folge von Pivot-Elementen, die zur geringsten Menge von Fillelementen führt, ein NP-schweres Problem [102]. Deshalb wurden verschiedene Heuristiken entwickelt, um den Fill gering zu halten.

Es gibt zwei grundlegende Ansätze zur Fillminimierung. Der eine besteht darin, die Matrix *vor* der Faktorisierung so zu permutieren, daß anschließend die Diagonalelemente als Pivot-Elemente verwendet werden können und möglichst wenig Fillelemente auftreten. Ein Beispiel hierfür sind sog. Skyline Solver, bei denen eine Permutation der Zeilen und Spalten der Matrix gesucht wird, so daß alle NNEs „nahe“ an der Diagonalen liegen. Dadurch ist bei der anschließenden Faktorisierung gewährleistet, daß sich der Fill auf einen engen Bereich um die Diagonale (die Skyline) beschränkt. Diese Methode eignet sich besonders für den symmetrischen Fall, wenn zusätzlich noch eine spezielle Struktur der Matrix ausgenutzt werden kann (etwa von Finite-Elemente-Gittern).

Der zweite Ansatz ist die *lokale* Fillminimierung. Dabei werden während des Eliminationsvorganges die Pivot-Elemente so gewählt, daß in jedem Schritt nur ein geringer Fill zu erwarten ist. Für Matrizen ohne bekannte Struktur haben sich lokale Methoden als günstiger erwiesen. Seien r_i^S und c_j^S die Anzahl der NNEs in der i-ten Zeile bzw. der j-ten Spalte der aktiven Matrix $B_{I^s J^s}^s$ im s-ten Eliminationsschritt. Dann ist $m_{ij}^s = (r_i^s - 1)(c_j^s - 1)$ wegen Schritt 4 eine ober Schranke für die Anzahl von Fillelementen, die bei der Wahl von B_{ij}^s als Pivot-Element im s-ten Schritt entstehen kann. Man nennt m_{ij}^s die *Markowitz-Zahl* der Elementes B_{ij}^s. Lokale Pivot-Strategien wählen als Pivot-Elemente solche mit geringer Markowitz-Zahl. Verschiedene Varianten unterscheiden sich darin, wieviel Aufwand für die Suche nach Elementen mit niedriger Markowitz-Zahl getrieben wird.

Bei der Fillminimierung muß jedoch auch die Stabilität der LU-Zerlegung berücksichtigt werden. Wie in 1.7.3.1 beschrieben, müssen dafür Pivot-Elemente mit möglichst großem Betrag gewählt werden. Es gilt also, einen Kompromiss zwischen Fillminimierung (kleinste Markowitz-Zahl) und Stabilität (betragsgrößtes NNE) zu schließen. In der Praxis hat sich dafür die *Schwellwertpivotsuche* bewährt. Für einen vorgegebenen Schwellwert $0 < u \leq 1$ werden als Pivot-Elemente alle NNEs erlaubt, für die

$$|B_{i^s j^s}^s| \geq u \cdot \max_{j \in J^s} |B_{i^s j}^s| \tag{1.115}$$

gilt. Für $u = 1$ entspricht dies der partiellen Pivot-Suche. Kleinere Werte von u erlauben die Wahl von Pivot-Elementen mit kleinerer Markowitz-Zahl, möglicherweise jedoch auf Kosten der Stabilität. Als günstig haben sich Schwellwerte $u \geq 0.01$ erwiesen.

1.7.3.3 Implementierung

In den vorigen beiden Abschnitten wurde der Rahmen für die Wahl des Lösungsverfahrens für lineare Gleichungssysteme im Simplex-Algorithmus abgesteckt. Nun soll die für SoPlex gewählte Variante und deren Implementierung vorgestellt werden. Dabei gilt es zwei Aspekte zu beschreiben, in denen sich Unterschiede verschiedener Implentierungen von LU-Zerlegungsalgorithmen für dünnbesetzte nicht symmetrische Matrizen manifestieren. Dies ist zum einen das zugrundeliegende Datenlayout zur Speicherung der dünnbesetzten Matrizen und zum anderen die Strategie bei der Suche nach günstigen Pivot-Elementen im Sinne der Schwellwertpivotsuche.

Es gibt eine Reihe unterschiedlicher Speicherschemata für dünnbesetze Matrizen [39, 89]. Dabei unterscheidet man zwischen der Speicherung der NNEs in Feldern und ihrer Speicherung in verketteten Listen. Von der Komplexitätsabschätzung her sind für LU-Zerlegungsalgorithmen, bei der Fillelemente entstehen und andere Elemente Null werden können, doppelt verkettete Listen besonders geeignet [89]. Auf heutigen Cachearchitekturen treten aber bei einem Listenansatz häufig sog. Cache-Misses[9] auf, da kein regulärer

[9] Ein Cache-Miss führt dazu, daß der Prozessor seine Arbeit einige Taktzyklen unterbrechen muß, bis die geforderten Daten vom Hauptspeicher nachgeladen wurden. Dadurch verlangsamt sich die Bearbeitungsgeschwindigkeit.

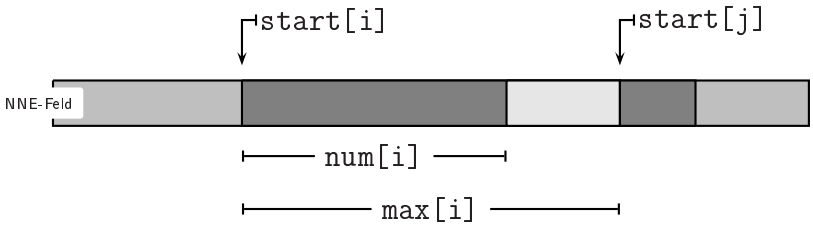


Abbildung 1.8: Speicherlayout für die NNEs der Zeilen der Arbeitsmatrix beim LU-Zerlegungsalgorithmus von SoPlex: Alle NNEs aller Zeilen werden in einem gemeinsamen NNE-Feld gehalten. Für die i-te Zeile wird jeweils ein Zeiger `start[i]` auf das erste NNE in dem Feld sowie die Anzahl `num[i]` der NNEs der i-ten Zeile gespeichert. Schließlich wird die Anzahl der NNEs im Feld bis zur nächsten Zeile in `max[i]` verwaltet.

Speicherzugriff erfolgt.

Deshalb wurde für SoPlex einer Speicherung in Feldern der Vorzug gegeben, ähnlich der in [92] beschriebenen. Die Arbeitsmatrix wird zeilenweise abgespeichert: Die NNEs aller Zeilen werden in einem Feld verwaltet. Für Zeile i wird in `start[i]` ein Zeiger auf das erste NNE in diesem Feld, in `num[i]` die Anzahl der NNEs des Zeilenvektors und in `max[i]` die Anzahl der im Feld verfügbaren NNE-Speicherplätze bis zur nächsten Zeile gespeichert (vgl. Abbildung 1.8).

Es werden also nicht nur die NNEs sondern auch die „Löcher“ dazwischen verwaltet: Die Differenz `max[i] - num[i]` beschreibt unbenutzten NNE-Speicher, in dem Fillelemente für die i-te Zeile erzeugt werden können. Entstehen hingegen Fillelemente wenn `max[i] == num[i]` gilt, so wird die betroffene Zeile an das Ende des benutzten Teils des NNE-Feldes kopiert, wo noch genügend Speicherplatz für die Fill-Elemente bereitsteht (andernfalls wird das Feld vergrößert). Dabei wird der von der verschobenen Zeile zuvor beanspruchte Platz der davor befindlichen Zeile als freier NNE-Speicher hinzugefügt. Um dies effizient durchführen zu können, werden alle Zeilen in einer doppelt verketteten Liste verwaltet, die nach der Anfangsaddresse im NNE-Feld sortiert ist.

Schließlich werden die Indizes der Matrix-NNEs auch spaltenweise abgespeichert, wobei dasselbe Layout benutzt wird. Dies ermöglicht auch einen spaltenweisen Zugriff zumindest auf die Indizes der NNEs der Matrix.

Das zweite Unterscheidungskriterium für verschiedene LU-Faktorisierungsalgorithmen ist die Pivot-Auswahl. Auch hier orientiert sich SoPlex an der in [92] beschriebenen Strategie, die wiederum geringfügig abgeändert wurde. Es werden doppelt verkettete Listen Z_i und S_i für $i = 1, \ldots, n$ geführt. Jeder Zeilenindex i und Spaltenindex j der aktiven Teilmatrix wird in der Liste $Z_{r_i^s}$ bzw. $S_{c_j^s}$ verwaltet. Wenn sich die Anzahlen der NNEs r_i^s oder c_j^s bei einem Pivot-Schritt ändern, werden die Indizes i bzw. j in die entsprechenden Listen verschoben.

Die Listen ermöglichen einen schnellen Zugriff auf Zeilen bzw. Spalten mit genau i NNEs. Insbesondere kann schnell von kleinen Werten aufwärts die Zeile oder Spalte mit der geringsten Anzahl von NNEs gesucht werden. Aus einigen (1-4) in diesem Sinne ersten Zeilen oder Spalten wird dasjenige NNE als Pivot-Element ausgewählt, daß die geringste Markowitzzahl aufweist und die Schwellwertbedingung erfüllt.

Zur Überprüfung der Schwellwertbedingung muß der größte Betrag in einer Zeile bestimmt werden. Um dies nicht für jedes untersuchte NNE erneut zu bestimmen, wird ein Feld `maxabs` verwaltet. Ein Wert `maxabs[i]` < 0 zeigt an, daß der größte Betrag in der i-ten Zeile nicht bekannt ist. Immer wenn er berechnet wird, wird er in `maxabs[i]` eingetragen. Ändert sich aber etwas an der i-ten Zeile, so wird wieder `maxabs[i] = -1` gesetzt. Dadurch wird das Maximum immer nur dann bestimmt, wenn es benötigt wird, zuvor aber noch nicht berechnet wurde.

Eine weitere wesentliche Eigenschaft der Implementierung für SoPlex ist, daß sie gleichzeitig ein Maß für die Stabilität der LU-Zerlegung bestimmt. Dazu wird das Maximum $\max_s\{|B^s_{ij}|\}$ mitgerechnet und auch bei den im folgenden Abschnitt zu beschreibenden Updates der LU-Zerlegung aktualisiert. Falls nach erfolgter Faktorisierung keine hinreichende Stabilität erreicht wurde, wird die LU-Zerlegung mit erhöhtem Parameter u wiederholt. Dies wird so lange iteriert, bis entweder eine stabile Zerlegung gefunden oder mit $u = 1$ eine partielle Pivot-Suche verwendet wurde. Letzteres tritt in der Praxis nur sehr selten ein.

1.7.4 Lösung von Gleichungssysteme mit Dreiecksmatrizen

Wie im vorigen Abschnitt beschrieben müssen zur Lösung von linearen Gleichungssystemen bei gegebener LU-Zerlegung der Matrix $B = LU$ Gleichungssysteme mit den permutierten Dreiecksmatrizen L und U gelöst werden. Für Gleichungssysteme der Form (1.109) sind dies

$$Ly = b \tag{1.116}$$

$$\text{und} \qquad Ux = y, \tag{1.117}$$

wofür nun die zugehörigen Algorithmen, die Vorwärts- und Rückwärtssubstitution, dargestellt werden. Die entsprechenden Algorithmen für Gleichungssysteme der Form (1.110) erhält man einfach durch Transposition und werden deshalb nicht näher beschrieben.

Betrachte zunächst die Lösung des Systems (1.116), wobei L eine untere Dreiecksmatrix mit Diagonalelementen 1 sei. Der Lösungsalgorithmus arbeiten in n Schritten, wobei n die Dimension von L ist. Ausgehend von $y = b$ wird y in jedem Schritt aktualisiert, wobei jeweils ein weiteres Element des Lösungsvektors von (1.116) berechnet wird. Dabei wird ausgenutzt, daß es wegen der Dreiecksgestalt von L jeweils eine Zeile mit nur noch einer Unbekannten gibt.

ALGORITHMUS 8 (VORWÄRTSSUBSTITUTION)
Sei $y = b \in \mathbb{R}^n$ und $L \in \mathbb{R}^{n \times n}$ eine untere Dreiecksmatrix.
Für $i = 1, \ldots, n$:
    Für $j = i + 1, \ldots, n$:
        Setze $y_j \leftarrow y_j - y_i \cdot L_{ji}$

Ersichtlich berechnet Algorithmus 8 y gemäß $y_j = b_j - \sum_{i=1}^{j-1} L_{ji} y_i$, wobei die Summe über mehrer Iterationen von i verteilt ausgewertet wird. Dies ermöglicht es, Dünnbesetztheit von b und y algorithmisch auszunutzen. Wann immer in einer Iteration $y_i = 0$ auftritt, kann die Schleife über j ausgelassen werden. Die Besetzungsstruktur von L wird bei der Schleife über j ausgenutzt, indem man sie auf die NNEs der i-ten Spalte von L einschränkt.

Der Algorithmus zur Lösung von (1.117) funktioniert analog zu Algorithmus 8:

ALGORITHMUS 9 (RÜCKWÄRTSSUBSTITUTION)
Sei $x = y \in \mathbb{R}^n$ und $U \in \mathbb{R}^{n \times n}$ eine obere Dreiecksmatrix.
Für $i = n, \ldots, 1$:
    $x_i \leftarrow x_i / U_{ii}$
    Für $j = i - 1, \ldots, 1$:
        Setze $x_j \leftarrow x_j - x_i \cdot U_{ji}$

Bei beiden Algorithmen 8 und 9 können die Permutationen P und Q direkt in die Verwaltung der Indizes i und j eingearbeitet werden.

Um die Besetzungsstruktur von y bzw. x wie oben beschrieben auszunutzen, müssen L und U spaltenweise abgespeichert werden. Entsprechend bedarf es für Gleichungssysteme der Form $x^T B = b^T$ bei L und U einer zeilenweisen Speicherung der Matrizen. Da bei Simplex-Algorithmen beide Formen von Gleichungssystemen gelöst werden müssen, werden die Faktoren L und U sowohl zeilen- als auch spaltenweise gespeichert.

Schließlich kann bei besonders dünnbesetzten Gleichungssystemen und Vektoren b eine weitere Geschwindigkeitssteigerung erzielt werden. Die Algorithmen 8 und 9 durchlaufen immer die gesamte Schleife für i von 1 bis n. Dabei muß dann jedoch nur für wenige Indizes i die Schleife über j durchgeführt werden, da meistens $y_i = 0$ bzw. $x_i = 0$ gilt. Um auch die Schleife über i zu verkürzen, kann man in einem d-heap die Indizes der NNE im Arbeitsvektor x oder y mitführen. Ein d-heap bietet einen direkten Zugriff auf den jeweils niedrigsten bzw. höchsten Index, so daß sofort auf das nächste NNE in der i-Schleife zugegriffen werden kann. Natürlich bedingt die Verwendung eines d-Heaps einen zusätzlichen Verwaltungsaufwand. Deshalb schaltet die für SoPlex vorgenommene Implementierung auf den heaplosen Algorithmus um, sobald der Heap „zu voll“ wird. Bezeichne h die Anzahl der Indizes im Heap. Es wird auf den originalen Algorithmus 9 zurückgeschaltet, sobald $d > \kappa i$, bzw. bei Algorithmus 8, wenn $d > \kappa(n - i)$ gilt. Für κ hat sich ein Wert von 0.05 bewährt. Damit wird der d-Heap nur in seltenen Fällen verwendet. Es gibt jedoch Beispiele, bei denen er eine deutliche Beschleunigung des Simplex-Algorithmus verursacht.

1.7.5 Basis-Updates

In jeder Iteration von Simplex-Algorithmen sind zwei lineare Gleichungssysteme mit der Basismatrix zu lösen, wobei sich diese von Iteration zu Iteration ändert. Jedesmal eine neue LU-Zerlegung zu berechnen wäre zu aufwendig, zumal sich die Basismatrix nur in einer Zeile oder Spalte gegenüber der aus der vorigen Iteration unterscheidet und es geeignete Verfahren gibt, um die alte LU-Zerlegung weiter zu verwenden. Zwei solcher sog. *LU-Updates*, nämlich die Produktform (PF) [29] und der Forest-Tomlin Update [46] in der Implementierung nach [93], werden in diesem Abschnitt am Beispiel des in der Literatur üblicheren Falles eines Spaltentausches beschrieben. Beide wurden für SoPlex implementiert, das zweite wegen seiner besonderen Effizienz [46] und PF wegen seiner Anwendung beim parallelen Simplex (vgl. Abschnitt 2.2.3). Nicht implementiert wurde das Verfahren von Bartels und Golub [10] mit seinen Weiterentwicklungen [84, 86] oder spezielle Updateverfahren für Vektorcomputer [42].

Betrachte die Matrix B, zu der die LU-Zerlegung $B = LU$ bekannt sei. Die Matrix $B' = BV$, mit $V = (I + (B^{-1}r - e_l)e_l^T)$ geht aus B durch Austausch der l-ten Spalte mit dem Vektor r hervor. Das Ziel ist es, ein Verfahren zu finden, mit dem Gleichungssysteme mit B' unter Verwendung der LU-Zerlegung von B gelöst werden können.

Das ursprüngliche Verfahren, das für Simplex-Algorithmen verwendet wurde, basiert auf (der transponierten Version von) Gleichung (1.89). Damit kann man Gleichungssysteme der Form $B'x = b$ lösen, indem man zunächst $Bx' = b$ mit der bekannten LU-Zerlegung von B löst und anschließend

$$x' = x + \frac{x_l}{(B^{-1}r)_l}(B^{-1}r - e_l) = V^{-1}x \tag{1.118}$$

bestimmt. Dabei wird der Vektor $B^{-1}r$ ohnehin als Δf im Simplex-Algorithmus berechnet. Bei mehreren Updates müssen entsprechend viele Korrekturschritte nach (1.118) durchgeführt werden.

Während das obige Verfahren die LU-Zerlegung von B aufrechterhält, wird diese durch das nun zu beschreibende Verfahren von Forest und Tomin manipuliert. Betrachte dazu die LU-Zerlegung von B in folgender von (1.112) abgeleiteten Form

$$L_n^{-1} \cdot \cdots \cdot L_1^{-1} \cdot B = L^{-1}B = U,$$

wobei PUQ eine obere Dreiecksmatrix ist. Da oft viele Spalten von L mit der Einheitsmatrix übereinstimmen, sind meist weniger als n Rang-1-Matrizen L_i nötig.

Betrachte $L^{-1}B' = UV$.

$$\begin{aligned} UV &= U(I + (B^{-1}r - e_l)e_l^T) \\ &= U + (UB^{-1}r - U_{.l})e_l^T \\ &= U + (L^{-1}r - U_{.l})e_l^T. \end{aligned}$$

Für die permutierte Matrix $PUVQ$ bezeichne k den permutierten Index l, d.h. $e_l^T Q = e_k^T$, und $x = PL^{-1}r$. Dann gilt

$$PUVQ = \begin{pmatrix} u_{11} & \dots & u_{1k-1} & x_1 & u_{1k+1} & \dots & u_{1n} \\ & \ddots & \vdots & \vdots & & & \\ & & u_{k-1k-1} & x_{k-1} & \vdots & & \\ & & & x_k & & & \vdots \\ & & & \vdots & u_{k+1k+1} & & \\ & & & x_j & & \ddots & \\ & & & 0 & & & u_{nn} \end{pmatrix},$$

wobei $j = \max\{i : x_i \neq 0\}$. Sofern B' nicht singulär ist, muß dabei $j \geq k$ gelten.

Im folgenden wird diese Matrix durch geeignete Permutation und Multiplikation mit Rang-1-Matrizen Λ_i in eine permutierte Dreiecksmatrix U' transformiert, so daß

$$\Lambda_j^{-1} \cdot \dots \cdot \Lambda_{k+1}^{-1} \cdot L^{-1} B' = U'$$

gilt, wobei $P'U'Q'$ eine obere Dreiecksmatrix, mit geeigneten Permutationsmatrizen P' und Q', ist. Definiere dazu zunächst die Permutationsmatrix.

$$\Pi = \begin{pmatrix} 1^1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1^{k-1} & & & & & & \\ & & & 0 & & 1^j & & & \\ & & & 1^k & \ddots & & & & \\ & & & & \ddots & \ddots & & & \\ & & & & & 1^{j-1} & 0 & & \\ & & & & & & 1^{j+1} & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1^n \end{pmatrix},$$

die die Indizes k bis j um eins durchrotiert. Damit ist

$$\Pi^{-1} PUVQ\Pi = \begin{pmatrix} u_{11} & \dots & u_{1k-1} & u_{1k+1} & \dots & u_{1j} & x_1 & u_{1j+1} & \dots & u_{1n} \\ & \ddots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ & & u_{k-1k-1} & u_{k-1k+1} & \dots & u_{k-1j} & x_{k-1} & u_{k-1j+1} & \dots & u_{k-1n} \\ & & & u_{k+1k+1} & \dots & u_{k+1j} & x_{k+1} & u_{k+1j+1} & \dots & u_{k+1n} \\ & & & & \ddots & \vdots & \vdots & \vdots & & \vdots \\ & & & & & u_{jj} & x_j & u_{jj+1} & \dots & u_{jn} \\ & & & u_{kk+1} & \dots & u_{kj} & x_k & u_{kj+1} & \dots & u_{kn} \\ & & & & & & & u_{j+1j+1} & & u_{j+1n} \\ & & & & & & & & \ddots & \vdots \\ & & & & & & & & & u_{nn} \end{pmatrix}.$$

Diese Matrix kann mit Gaußscher Elimination in eine obere Dreiecksmatrix $\bar{U}$ transformiert werden:

$$\bar{U} = \Lambda_{j-1} \cdot \cdots \cdot \Lambda_k \cdot \Pi^{-1} PUVQ\Pi = \Lambda \cdot \Pi^{-1} PUVQ\Pi \tag{1.119}$$

$$\text{mit} \qquad \Lambda_i = I + \lambda_i \cdot e_j e_i^T \qquad \text{und} \qquad \Lambda = \prod_{i=j-1}^{k} \Lambda_i = I + e_j \lambda^T,$$

wobei $\lambda^T = (0, \ldots, 0, \lambda_{k+1}, \ldots, \lambda_j, 0, \ldots, 0)$ ist. Setze nun

$$\begin{aligned} P' &= \Pi^{-1} P, \\ Q' &= Q\Pi, \\ L_{n+1}^{-1} &= (P')^{-1} \Lambda P', \\ U' &= L_{n+1}^{-1} UV \text{ und} \\ L' &= L \cdot L_{n+1}. \end{aligned}$$

Dann folgt

$$B' = BV = LUV = LL_{n+1}U' = L'U'$$

und

$$P'U'Q' = P'(P')^{-1} \Lambda P' UVQ\Pi = \Lambda \Pi^{-1} PUVQ\Pi = \bar{U}.$$

Diese Art von LU-Update führt somit zu neuen Permutationsmatrizen P' und Q', zu einer neuen permutierten oberen Dreiecksmatrix U' und zu einer zusätzlichen Rang-1-Matrix L_{n+1}. Da Gleichungssysteme mit L, L_{n+1} und U' direkt gelöst werden können, kann man damit auch das Gleichungssystem für B' lösen.

Der Vorteil dieser Update-Methode gegenüber der zuerst geschilderten ist, daß L_{n+1}^{-1} i.a. weniger NNEs hat als $B^{-1}r$, so daß die Lösung von Gleichungssystemen anschließend eines geringeren Rechenaufwandes bedarf. Der Update selbst bedingt jedoch einen höheren Aufwand, da der Vektor λ per Gaußscher Elimination berechnet werden muß. Der Vektor x fällt hingegen während des Simplex-Verfahrens an, denn $L^{-1}r$ wird bei der Berechnung von Δf benötigt, muß also lediglich zwischengespeichert werden. Bei geschickter Speicherverwaltung erfordert dies nicht einmal einen Kopiervorgang.

Der Forest-Tomlin Update ist eine Vereinfachung des Bartels-Goloub Updates. Bei letzterem werden bei der Gaußschen Elimination in (1.119) eventuell noch weitere Zeilenpermutationen eingebaut, um eine bessere Stabilität zu gewährleisten. Bei Simplex-Algorithmen bringt dieser zusätzliche Aufwand in der Regel keinen Vorteil. Ohnehin wird die Stabilität der (aktualisierten) LU-Zerlegung überwacht, und gegebenfalls erfolgt eine neue LU-Zerlegung (vgl. Abschnitt 1.8.5).

1.8 Tips und Tricks

Nachdem in den vorigen Abschnitten die Grundlagen für eine numerisch stabile Implementierung von basisdarstellungsunabhängigen Simplex-Algorithmen zusammengetragen wurden, sollen nun einige Kniffe vorgestellt werden, mit denen die Geschwindigkeit und Stabilität der Implementierung gesteigert werden kann. Von zentraler Bedeutung für die Effizienz ist in jedem Fall die sorgfältige Auswahl und Implementierung der zugrundegelegten Datentypen, die in Abschnitt 3.2 dargestellt werden.

1.8.1 Skalierung

Die Zahlenwerte in der Nebenbedingungsmatrix eines LPs können von unterschiedlicher Größenordung sein. Dies kann zu LU-Zerlegungen der Basismatrizen mit einer schlechten Stabilität führen, da die Schwellwertbedingung (1.115) kaum Wahlmöglichkeiten zuläßt. Durch sog. Skalierung kann die Stabilität der LU-Zerlegung verbessert werden. Dazu wird die Matrix von rechts und links mit Diagonalmatrizen so multipliziert, daß die NNEs der skalierten Matrix möglichst eng beieinander liegen. Dadurch erhöht sich die Wahlmöglichkeit bei der Schwellwert-Pivotsuche, was zu einer besseren Stabilität führt.

Die Berechnung solcher Skalierungsmatrizen für jede Basismatrix und ihre Multiplikation würde jedoch einen erhöhten Rechenaufwand bei der LU-Zerlegung der Basismatrix und der Lösung der linearen Gleichungssysteme bedeuten. Um diesen zu vermeiden, versucht man, das gesamte LP skalieren, so daß beliebige Basismatrizen bereits gut skaliert sind. Dazu multipliziert man je eine Diagonalmatrix R und C von rechts und links an die Nebenbedingungsmatrix des LPs A, damit Größenunterschiede der Zahlenwerte in RAC nivelliert werden:

$$\begin{array}{lll} \min & & c^T x \\ \text{s.t.} & l \leq & Ax \leq u \end{array} \quad \rightarrow \quad \begin{array}{lll} \min & & c^T C x \\ \text{s.t.} & Rl \leq & RACx \leq Ru \end{array} .$$

Es wurden verschiedene Ansätze zur Aufstellung der Diagonalwerte von C und R vorgeschlagen [96, 28, 48]. Einfache Skalierungsverfahren dividieren jede Zeile des LPs mit einem aus der Zeile gewonnen Wert, wie den Wert mit maximalem Absolutbetrag (Equilibrierung) oder dem geometrischen oder arithmetischen Mittel aller NNEs der Zeile. Anschließend wird jede Spalte entsprechend skaliert. Derartige Verfahren haben einen geringen Rechenaufwand, denn jedes NNE der Nebenbedinungsmatrix wird nur zweimal verwendet.

Neben solch einfachen Methoden gibt es aufwendigere Verfahren, die die „Gleichartigkeit“ der NNEs optimieren: Das Verfahren von Fulkerson-Wolfe [48] löst ein Optimierungsproblem zur Minimierung der Differenz zwischen maximalen und minimalen NNE. Dagegen bestimmt das Verfahren von Curtis und Reid [28] die Skalierungsfaktoren so, daß die Varianz der NNEs des LPs minimiert wird.

Aus verschiedenen Tests schließt Tomlin in [96], daß die Skalierungsverfahren nur einen geringen Einfluß auf die Stabilität des Simplex-Algorithmus haben, jedenfalls für „wohl

modellierte“ LPs. Um den Algorithmus vor schlechten LP-Modellen abzusichern, sollten hingegen einfache Skalierungsverfahren ausreichen. Deshalb wurde für SoPlex ähnlich wie für CPLEX [15] lediglich ein Equilibrierungsverfahren implementiert, das jedoch im Gegensatz zu CPLEX auch abgeschaltet werden kann. Darüberhinaus bietet SoPlex die Wahl, ob zuerst die Zeilen oder zuerst die Spalten skaliert werden sollen. Was günstiger ist, hängt von dem zu lösenden LP ab.

Über den Stabilitätsaspekt hinaus wird häufig als Argument für eine Skalierung des LPs noch eine mögliche Reduktion der Anzahl der Simplex-Iterationen genannt [96]. In der Tat kann die Skalierung einen Einfluß auf die Anzahl der Simplex-Iterationen haben. Dieser kann sich jedoch sowohl positiv als auch negativ auswirken (vgl. Abschnitt 4.2.6).

1.8.2 Quotiententest

Wie bereits in Abschnitt 1.3 bemerkt wurde, ist die stabilisierte Variante des „Textbook“-Quotiententests oft hinreichend stabil. Mit nur einer Phase ist sie effizienter als zweiphasige Varianten. Die Implementierung des stabilen SoPlex Quotiententest bricht deshalb direkt nach der ersten Phase ab, wenn das Θ_{max} bestimmende Element bereits zu einem Pivot-Schritt hinreichender Stabilität führt. Solch eine Wahl entspricht bis auf das Shifting dem stabilisierten „Textbook“-Quotiententest.

Darüberhinaus speichert der SoPlex Quotiententest in seiner ersten Phase die Indizes der NNEs von Δx. Dadurch kann in der zweiten Phase direkt auf die NNEs zugegriffen werden, und unnötige Zugriffe auf Nullelemente werden vermieden. Auch wenn keine zweite Phase benötigt wird, wurden die Indizes der NNEs von Δx nicht unnötig gespeichert; sie werden in jedem Fall beim Update von x benutzt.

Für die meisten LPs liefert der SoPlex Quotiententest mit festem Toleranzparameter δ eine hinreichende Stabilität. Dem Autor liegen jedoch LPs[10] vor, bei denen erst zwei weitere Techniken zur erfolgreichen Lösung verhelfen. Dies ist zum einen die *dynamische Toleranzanpassung* und zum anderen das *Zurückweisen* von Pivot-Elementen.

Für das Zurückweisen von Pivot-Elementen wird ein weiterer Parameter `minstab` eingeführt. Führt der Quotiententest zu einem Pivot-Schritt mit $|\Delta s_q| <$`minstab`, so wird kein Basisupdate durchgeführt, denn ein solcher würde nach Satz 29 zu einer zu schlecht konditionierten Basis führen. Stattdessen wird direkt zum Pricing-Schritt 1 übergegangen, wo ein neues Pivot-Element gewählt wird. Dabei werden Vorkehrungen getroffen, die eine erneute Wahl desselben Elementes vermeiden.

Ein zurückgewiesenes Pivot-Element zeigt somit an, daß sich der Simplex-Algorithmus in einem numerisch problematischen Bereich des zugrundeliegenen Polyeders bewegt. Deshalb wird der Toleranzparameter δ (dynamisch) heraufgesetzt, um so eine größere Wahlfrei-

[10] Diese besonders schwierigen LPs dürfen leider nicht weitergegeben werden und wurden deshalb auch nicht zur Bewertung der Implementierung in Kapitel 4 herangezogen.

heit beim Quotiententest und damit einen numerisch stabilieren Basisupdate zu ermöglichen. Gleichzeitig wird auch der Parameter `minstab` herabgesetzt, um nicht wegen überzogener Stabilitätsanforderungen in einer unendlichen Folge zurückgewiesener Pivot-Schritte zu enden. Beide Parameter δ und `minstab` werden wieder gestrafft, wenn die Größe der Werte $|\Delta s_q|$ anzeigt, daß der numerisch problematische Bereich des Polyeders verlassen wurde.

1.8.3 Pricing

Die Auswahl der für ein Problem „richtigen“ Pricing-Strategie ist essentiell für die Lösungsgeschwindigkeit von Simplex-Algorithmen. Dabei führt das steepest-edge Pricing meist zu signifikant weniger Iterationen als andere Verfahren, ist jedoch aufgrund des erhöhten Rechenaufwandes dennoch oft langsamer. Insbesondere bei einfügenden Algorithmen sollte oft das partial multiple Pricing oder das Devex Pricing vorgezogen werden. Welche Pricing-Strategie am schnellsten arbeitet, hängt überdies von der zugrundeliegenden Hardware ab. Bei Computern mit hoher Gleitkomma-Arithmetik-Leistung kann das steepest-edge häufiger eingesetzt werden als z.B. auf einem 486er PC.

Beim steepest-edge Pricing kann der Rechenaufwand dadurch in Grenzen gehalten werden, daß beim Start die Werte $\rho_i = 1$ gesetzt werden, anstatt sie korrekt durch Lösung der entsprechenden Gleichungssysteme zu bestimmen. Dies führt meist zu keiner nennenswerten Steigerung der Iterationszahl. Insbesondere beim entfernenden Algorithmus und einer Schlupfbasis ist diese Initialisierung sogar exakt.

Wiederum für das steepest-edge Pricing kann die Lösung des zusätzlichen Gleichungssystems beschleunigt werden. Es kann nämlich zu jedem beliebigen Zeitpunkt nach der Lösung des ersten Gleichungsssystems in Schritt 2 der Simplex-Algorithmen und vor der Aktualisierung der Basismatrix in Schritt 5 gelöst werden, insbesondere auch gleichzeitig mit dem zweiten System in Schritt 4. Dann kann man beide Gleichungssysteme mit nur einer Traversierung der LU-Zerlegung lösen. Dabei muß die Schleife über i in den Algorithmen 8 und 9 nur einmal durchlaufen werden. Gleiches gilt für die j-Schleife, wenn sie für beide Gleichungssysteme erforderlich ist. Insgesamt ergeben sich dadurch weniger Speicherzugriffe als bei zweifacher Ausführung der Algorithmen 8 und 9, und es gelingt eine bessere Nutzung des Caches.

Allgemein kann man einen Pricer oft dadurch etwas verbessern, daß man beim einfügenden Algorithmus Vektoren preferiert, die die Basis nicht wieder verlassen können, bzw. beim entfernenden Algorithmus solche, die nicht wieder in die Basis eintreten werden. Für eine Spaltenbasis würde man also zunächst fixierte Variablen aus der Basis entfernen. Dabei ist jedoch zu bemerken, daß — wie bei allen heuristischen Ansätzen — auch Problembeispiele existieren, bei denen diese Strategie verheerende Auswirkungen haben kann.

1.8.4 Zeilen- versus Spaltenbasis

Die Möglichkeit, sowohl eine Zeilen- als auch eine Spaltenbasis zu verwenden, erlaubt es, bei der Dimension der Basismatrix zwischen der Anzahl der Spalten oder der Anzahl der Zeilen der Nebenbedinungsmatrix des LPs zu wählen. Dabei sollte die kleinere Dimension vorgezogen werden, da so die Lösung der linearen Gleichungssysteme einen geringeren Rechenaufwand erfordert.

Darüberhinaus kann die Wahl der Basisdarstellung einen Einfluß auf die Wahl der Pricer haben. Einige Pricing-Strategien, insbesondere das partial multiple Pricing, sind nur für einfügende Algorithmen sinnvoll. Weiß man nun zusätzlich, daß ein LP günstiger mit einem dualen als mit einem primalen Algorithmus gelöst wird, so kann dieses Pricing-Verfahren dennoch eingesetzt werden, indem die Zeilendarstellung verwendet wird.

1.8.5 Refaktorisierung

Wie in Abschnitt 1.7.3 dargelegt, wird zur Lösung der bei Simplex-Algorithmen auftretenden Gleichungssysteme eine LU-Zerlegung der Basismatrix vorgenommen. Nach einem Basistausch wird i.a. nicht wieder eine völlig neue LU-Zerlegung berechnet, sondern die Bestehende in einer der in 1.7.5 beschriebenen Arten manipuliert. Nun stellt sich die Frage, wie oft solche Updates sinnvoll sind. Dabei ist zweierlei zu berücksichtigen:

1. die Stabilität der Faktorisierung und

2. die Gesamtgeschwindigkeit des Simplex-Algorithmus.

Dem ersten Punkt muß stets Vorrang gewährt werden. Deshalb wird wie in 1.7.3 beschrieben die Stabilität der Faktorisierung überwacht und bei Überschreiten eines Grenzwertes eine neue LU-Zerlegung der Basismatrix berechnet. In der Praxis hat sich ein Grenzwert in der Größenordnung $10^6 - 10^8$ bewährt.

Der zweite Punkt ist immer dann zu beachten, wenn die Stabilitätsanforderung ohnehin erfüllt ist, und dies ist bei über 90% der dem Autor vorliegenden LPs der Fall. Somit lohnt sich ein genaueres Studium des Geschwindigkeitsaspektes. In [26] wird dies unter statischen Voraussetzungen durchgeführt, was zu einer konstanten Faktorisierungsfrequenz führt. In [64] wird die „optimale“ konstante Faktorisierungsfrequenz anhand eines Testlaufes bestimmt. SoPlex verfolgt hingegen einen dynamischen Ansatz, bei dem der optimale Zeitpunkt der erneuten LU-Zerlegung der Basismatrix anhand des jeweils aktuellen Zustands bestimmt wird.

Da i.a. bei jedem Update der Faktorisierung zusätzliche Werte gespeichert werden, die bei anschließend gelösten Gleichungssystemen bearbeitet werden müssen, wächst mit jedem Update der Aufwand zur Lösung von Gleichungssystemen. Ziel ist es daher, den

Zeitpunkt der erneuten LU-Zerlegung so zu wählen, daß die Gesamtgeschwindigkeit des Simplex-Verfahrens optimiert wird.

Sei N die Anzahl der Nicht-Null-Elemente der Basismatrix und n_i die der i-fach aufdatierten LU-Zerlegung. $\phi = n_0/N$ gibt den Anteil der Füllelemente bei der LU-Zerlegung an. Die mittlere Zeit zur Lösung eines Gleichungssystems t_i ist proportional zu n_i. Ferner setzen wir die Zeit für die LU-Zerlegung T als proportional zur Anzahl der Nicht-Null-Elemente der Faktoren L und U an:

$$t_i = \theta n_i \tag{1.120}$$

$$T = \eta\theta n_0 = \eta\theta\phi N. \tag{1.121}$$

Die Werte $\theta, \eta \geq 0$ sind architekturabhängige Konstanten. Natürlich ist ϕ vor einer Faktorisierung nicht bekannt. Statt dessen kann man jedoch den Wert ϕ von der vorigen Faktorisierung als gute Approximation verwenden.

Optimierung der Geschwindigkeit bedeutet offenbar die Minimierung der Zeit pro Iteration. Sei f die Anzahl der LU-Updates seit der letzten Faktorisierung, dann ist dies

$$\bar{\varphi}(f) = \frac{T + \sum_{i=0}^{f} t_i}{f} = \theta\frac{\eta\phi N + \sum_{i=0}^{f} n_i}{f}. \tag{1.122}$$

Der Faktorisierungszeitpunkt f^* muß also so gewählt werden, daß $\varphi(f^*)$ minimal ist:

$$f^* = \arg\min_f \bar{\varphi}(f) = \arg\min_f \frac{\eta\phi N + \sum_{i=0}^{f} n_i}{f}. \tag{1.123}$$

Somit rechnet man während des Simplex-Algorithmus die Größe

$$\varphi(f) = \frac{\eta\phi N + \sum_{i=0}^{f} n_i}{f}$$

mit. Zunächst fällt sie mit wachsendem f. Sobald sie wieder zu steigen beginnt, wird die Refaktorisierung der Basismatrix vorgenommen. Der Rechenaufwand für diese Buchführung umfaßt lediglich eine Addition und eine Division pro Simplex-Iteration und kann somit vernachlässigt werden.

Der Parameter η repräsentiert das Verhältnis von dem Rechenaufwand für die Faktorisierung zu dem für die Lösung der Gleichungssysteme. Er hängt sowohl von der Architektur als auch von dem gewählten Pricer ab. Letzteres liegt daran, daß etwa bei steepest-edge Pricing ein zusätzliches Gleichungssystem pro Iteration gelöst werden muß, so daß in jeder Iteration dem Lösen von Gleichungssystemen mehr Aufwand zuzuordnen ist. Dementsprechend lohnt sich hier eine häufigere Refaktorisierung, was durch einen niedrigeren Wert für η modelliert wird. Für jede Klasse von LPs sollte der beste Wert für $\eta\phi$ über eine Reihe von Testläufen bestimmt werden (vgl. Abschnitt 4.2.4).

1.8.6 Die Startbasis

Für allgemeine LPs der Form (1.45) ist $Z = (P, Q)$, mit $P = \{1, \ldots, n\}$ und $Q = \{n+1, \ldots, n+m\}$ eine mögliche Startbasis zum zugehörigen Phase-1-LP, die *Schlupfbasis* genannt wird. Es kann jedoch zur Reduktion der gesamten Iterationszahl von Vorteil sein, auf heuristischem Weg eine bessere Startbasis zu konstruieren, die sich weniger von einer zulässigen und optimalen Basis unterscheidet. Solch ein Verfahren heißt eine *Crash-Prozedur*, die generierte Basis eine *Crash-Basis*.

Auch für SoPlex wurde eine Crash-Prozedur entwickelt, die sich stark an [15] orientiert und nun für eine Basis in Zeilendarstellung vorgestellt wird. Sie arbeitet in drei Phasen. In der ersten Phase, wird jeder Zeile von (1.45) ein Präferenzparameter ω_i ($i = 1, \ldots, n+m$) zugewiesen, der eine Präferenz für die Aufnahme der i-ten Zeile in die Basismatrix darstellt. Anschließend werden alle Zeilen nach abnehmendem Präferenzparameter sortiert und schließlich greedy-artig eine Basis konstruiert.

Die Festsetzung der Präferenzparameter erfolgt auf heuristischem Wege. Dazu überlegt man sich, welche Bedingungen $l \leq a^T x \leq u$ eher in der Basis sein sollten und welche eher nicht. So werden Bedingungen mit $l = -\infty$ und $u = +\infty$ eher selten in der optimalen Basis zu finden sein, wohl aber Gleichungen ($l = u$). Allgemein wird man vermuten, daß je größer der Wert $l - u$ ist, desto wahrscheinlicher wird die zugehörige Bedingung in einer zulässigen und optimalen Basis zu finden sein.

Dieses heuristische Argument berücksichtigt zwar die Schranken der Bedingungen, nicht aber die Zielfunktion. Abbildung 1.1 legt jedoch nahe, daß in einer optimalen Basis eher Nebenbedingungen zu finden sind, deren Normalenvektor einen geringen Winkel zum Zielfunktionsvektor aufweisen.

Schließlich sollten auch die numerischen Eigenschaften der zu konstruierenden Startbasis Berücksichtigung finden. Dabei sind (Vielfache von) Einheitsvektoren numerisch besonders gutartig und sollten deshalb Vektoren mit mehreren Nichtnullelementen vorgezogen werden.

Diese drei Aspekte werden durch die folgende Initialisierung der Präferenzparameter aufgegriffen. Sie verwendet einen Strafwert M, der z.B. auf einen Wert von $M = 10^{10}$ gesetzt wird. Für die i-te Nebenbedingung $l_i^Z \leq A_{i.}^Z x \leq u_i^Z$ setze zunächst

$$\omega_i' = \begin{cases} 4M & \text{, falls } l_i^Z = u_i^Z \\ 2M + l_i^Z - u_i^Z & \text{, falls } -\infty < l_i^Z \neq u_i^Z < \infty \\ l_i^Z & \text{, falls } -\infty = l_i^Z \neq u_i^Z < \infty \\ -u_i^Z & \text{, falls } -\infty < l_i^Z \neq u_i^Z = \infty \\ -2M & \text{, sonst} \end{cases}$$

und

$$\omega_i'' = \begin{cases} M & \text{, falls } A_{i.}^Z \text{ ein (skalierter) Einheitsvektor ist} \\ 0 & \text{, sonst.} \end{cases}$$

Damit setze als Präferenzparameter

$$\omega_i = \delta \cdot A^Z_{i\cdot} c + \omega'_i + \omega''_i. \tag{1.124}$$

Mit dem Parameter δ kann bestimmt werden, wie stark der Einfluß der Zielfunktion gewählt sein soll. Ein Wert $\delta = 10^{-3}$ hat sich dabei als günstig erwiesen.

Bei der greedy-artigen Konstruktion der Basis muß darauf geachtet werden, daß die Basismatrix gut konditioniert bleibt. Dazu wird ein Parameter $\epsilon > 0$ benutzt, der typischerweise auf einen Wert $\epsilon = 0.01$ gesetzt wird. Außerdem sollte der Rechenaufwand in vertretbaren Grenzen bleiben, denn die Präferenzen basieren ohnehin nur auf heuristischen Überlegungen.

ALGORITHMUS 10 (KONSTRUKTION EINER STARTBASIS)

Sei $x \in \mathbb{R}^{m+n}$ und π eine Permutation der Indizes $1, \ldots, m+n$, derart daß $\omega_{\pi_i} \geq \omega_{\pi_{i+1}}$ für alle i gilt.

Schritt 0 (Initialisierung):
Setze
$x \leftarrow 0$,
$i \leftarrow 0$ und
$P \leftarrow \emptyset$.

Schritt 1: $i \leftarrow i + 1$.
Falls $i = m + n$ terminiere.

Schritt 2: Sei $a^T = A^Z_{\pi_i \cdot}$, und $\mu = \max\{|a_j| : 1 \leq j \leq m+n\}$.
Falls für alle $1 \leq j \leq m+n$ gilt $x_j \neq 0$ oder $|a_j| < \epsilon\mu$:
gehe zu Schritt 1.
Sonst setze $k \leftarrow \arg\max\{|a_j| : x_j \neq 0\}$.

Schritt 3: Setze $x_k \leftarrow 1$

Schritt 4: Für alle j, mit $a_j \neq 0$ und $x_j = 0$:
Setze $x_j \leftarrow 1$
und $P \leftarrow P \cup \{j\}$.

Schritt 5: Setze
$P \leftarrow P \cup \{\pi_i\}$ und

Schritt 6: Gehe zu Schritt 1.

SATZ 31

Nach Ablauf von Algorithmus 10 ist $A^Z_{P\cdot}$ eine permutierte Dreiecksmatrix. Falls $|P| = n$ ist diese regulär.

Beweis:
Sei P_e der e-te in den Schritten 4 oder 5 zu P hinzugefügte Index. Wir zeigen

$$0 < l < e \;\Rightarrow\; A^Z_{P_e P_l} = 0, \tag{1.125}$$

d.h. A^Z ist eine permutierte Dreiecksmatrix. Dies gilt in jedem Fall für $e = 1$. Nach jedem Schritten gilt jeweils ($x_h = 0 \;\Rightarrow\; A^Z_{Ph} = 0$). Nun wird in Schritt 4 P um einen Index j mit $x_j = 0$ erweitert, d.h. (1.125) gilt auch nach Schritt 4. In Schritt 5 wird k zu P hinzugefügt, wobei in Schritt 2 noch (1.125) galt. Da in Schritt 4 $j \neq k$ sein muß, trifft dies auch noch in Schritt 5 zu. Da die Schritte 4 und 5 die einzigen Schritte sind, die P manipulieren, gilt (1.125) auch nach Termination von Algorithmus 10, d.h. $A^Z P.$ ist eine Dreiecksmatrix.

Eine Dreiecksmatrix ist genau dann regulär, wenn es in jeder Zeile und Spalte mindesten ein von Null verschiedenes Element (kurz Nicht-Null-Element, NNE) gibt. Falls $|P| = n$ trifft dies für die Zeilen trivialerweise zu. Wird in Schritt 4 P um einen Index j erweitert, so enthält anschließend die Spalte j ein NNE, da $A^Z_{j.} = e_j^T$. Da dabei zuvor $x_j = 0$ galt, gab es zuvor in dieser Spalte noch kein NNE. Wird in Schritt 5 π_i zu P hinzugefügt, so war nach Schritt 2 in der Spalte k noch kein NNE. Dies gilt auch noch nach Schritt 4. Da aber nach Schritt 2 $A^Z_{\pi_i k} \neq 0$ ist, gibt es nach Schritt 5 in der Spalte k ein NNE. Insgesamt gibt es also stets in $|P|$ Spalten mindestens ein NNE. Terminiert Algorithmus 10 mit $|P| = n$, so findet sich in jeder Zeile ein NNE, was den Beweis beendet.

■

Kapitel 2

Parallelisierung

Aufgrund der wirtschaftlichen Bedeutung der Linearen Programmierung ist es nicht verwunderlich, daß es bereits verschiedene Untersuchungen gegeben hat, wie LPs durch Nutzung von Parallelität schneller gelöst werden können. Einige Autoren beschreiben parallele Implementierungen des Simplex-Algorithmus, während andere spezielle Varianten (z.B. Netzwerk-Simplex [9]) oder völlig andere Algorithmen zugrundelegen [97, 105]. In [75] wird sogar ein spezieller Parallelrechner für LP-Probleme vorgeschlagen.

In der Literatur zur Parallelisierung des Simplex-Algorithmus gibt es Veröffentlichungen, die LPs mit spezieller Struktur betrachten, die für eine Parallelisierung nutzbar gemacht werden kann [51, 74]. Die Mehrzahl der Publikationen befaßt sich jedoch mit dem üblichen primalen Simplex-Algorithmus. Allerdings gibt es bis auf [13] keine Arbeit, die zu einer Implementierung führt, die mit state-of-the-art sequentiellen Implementierungen konkurrieren könnte. Dies liegt zum einen daran, daß entweder vom Simplex-Algorithmus in Tableau-Form ausgegangen wird [20, 90, 8] oder zum anderen die vorgestellten Implementierungen auf dichtbesetzten Vektoren und Matrizen operieren [91, 78, 43]. Mit dünnbesetzten Matrizen arbeiten hingegen die in [62, 58, 59] vorgestellten Implementierungen. Jedoch ist nicht ersichtlich, ob tatsächlich ein leistungsfähiger Code entwickelt wurde. Dies ist nur für [13] sichergestellt, wobei sich die Parallelisierung jedoch nur auf das Matrix-Vektor-Produkt beim dualen Simplex-Algorithmus mit Spaltenbasis beschränkt (vgl. Abschnitt 2.2.1).

In diesem Kapitel werden die SMoPlex und DoPlex zugrundegelegten Parallelisierungsansätze vorgestellt. Zunächst wird in Abschnitt 2.1 eine kurze Einführung in das Gebiet des Parallelen Rechnens gegeben, wobei den drei zentralen Aspekten, nämlich der Zerlegung von Algorithmen in nebenläufig bearbeitbare Teilprobleme, den wichtigsten Klassen von parallelen Hardware-Architekturen sowie den hier zum Einsatz kommenden parallelen Programmiermodellen je ein Teil-Abschnitt gewidmet wird. Schließlich werden in Abschnitt 2.1.4 die wichtigsten Grundbegriffe und Zusammenhänge für die Beurteilung paralleler Algorithmen vorgestellt.

Die nutzbare Nebenläufigkeit im Simplex-Algorithmus wird in Abschnitt 2.2 herausgearbeitet. In je einem Unterabschnitt werden dort die parallele Berechnung des Matrix-Vektor-Produktes, die Parallelisierung des Pricings und Quotiententest, das Block-Pivoting sowie die gleichzeitige Lösung zweier linearer Gleichungssysteme beim steepest-edge Pricing erklärt. Abschnitt 2.2.5 faßt die Parallelisierungskonzepte anhand eines Laufzeitdiagramms noch einmal zusammen. Schließlich behandelt Abschnitt 2.3 die parallele Lösung dünnbesetzter linearer Gleichungssysteme auf Basis eines parallelen LU-Zerlegungs-Algorithmus'.

2.1 Grundlagen der Parallelverarbeitung

Bis heute hält eine intensive Jagd nach immer schnelleren Rechnern an, um damit immer größere Probleme zu lösen. Als Faustregel gilt eine Verdopplung der Rechengeschwindigkeit alle 18 Monate. Doch wie lange noch ist diese Entwicklung fortsetzbar?

Vom theoretischen Standpunkt aus ist die Lichtgeschwindigkeit als absolute Grenze jeder Signalübertragung eine — wenn auch noch lange nicht erreichte — Schranke für die Rechengeschwindigkeit. Technisch wird es jedoch schon heute immer schwieriger, noch höhere Taktfrequenzen zu realisieren, da bei Frequenzen im Mikrowellenbereich die Leiterbahnen innerhalb einer integrierten Schaltung induktive und kapazitive Widerstände darstellen, die das logische Verhalten des Chips stören können. Mit immer höheren Taktfrequenzen allein können also schon heute derartige Beschleunigungen nicht mehr erzielt werden. Ein Ausweg ist die *Ausnutzung von Parallelität*, d.h. das gleichzeitige Zusammenarbeiten mehrerer Funktionseinheiten, um so mehr Resultate pro Zeit zu berechnen.

Vom normalen PC- oder Workstation-Nutzer unbemerkt, wird schon heute bei allen moderen Computern Parallelität ausgenutzt, nämlich innerhalb von RISC (reduced instruction set computer) Prozessoren. Vereinfacht sind zur Ausführung eines RISC-Befehls vier Teilschritte, die sog. RISC-Pipeline, erforderlich [68]:

1. fetch: Laden des Befehlsbytes
2. decode: Bestimmung des auszuführenden Befehls
3. execute: Ausführung der Operation
4. write back: Rückschreiben der Ergebnisdaten

Jeder dieser Schritte erfordert einen Taktzyklus, so daß ein einzelner Befehl erst nach 4 Zyklen tatsächlich abgearbeitet ist. Beim Pipelining werden die Teilschritte aufeinanderfolgender Prozessorbefehle so ineinander verschachtelt, daß nach den ersten 4 Taktzyklen mit jedem Takt ein neues Resultat produziert wird (vgl. Abb. 2.1).

Ähnliche Prinzipien werden bei modernen Prozessoren immer weiter ausgenutzt. Mehrere Pipelines arbeiten parallel (superpipelined architecture), es werden mehrer arithmetische Einheiten parallel eingesetzt (superscalar), bei Verzweigungsanweisungen wird jeder Zweig in einer Pipeline weiter bearbeitet, wobei nicht benötigte Ergebnisse dann verworfen werden (specular execution) und vieles mehr. Auf all dies soll nicht tiefer eingegangen werden,

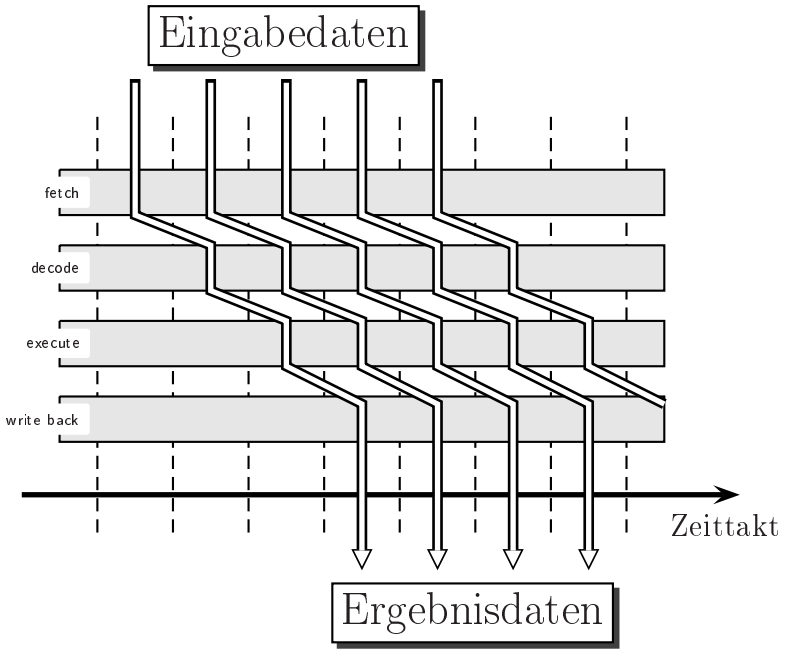


Abbildung 2.1: Das Prinzip des Pipelining moderner RISC-Architekturen. Jeder Befehl ist in vier Microbefehle unterteilt, die jeweils einen Zeittakt beanspruchen. Sie werden zeitversetzt abgearbeitet, so daß nach einer Startzeit in jedem Takt ein Ergebnis gewonnen wird.

---

da hier nur verdeutlicht werden soll, wieviel Parallelität derzeit schon bei der Ausführung "rein sequentieller" Programme intern ausgenutzt wird.

Auch auf einer "groben" Ebene ist die Nutzung von Parallelität zur unbemerkten Alltäglichkeit geworden. Bei jeder Multiprozessor-Workstation regelt das Betriebssystem die parallele Bearbeitung verschiedener Prozesse auf verschiedenen Prozessoren. Jeder einzelne Prozess wird jedoch weiterhin sequentiell ausgeführt.

Das parallele Rechnen beschäftigt sich mit der Nutzung von Parallelität, die zwischen den beiden o.g. Ebenen anzusiedeln ist: der *algorithmischen Parallelität*, die bei der Lösung mancher Probleme nutzbar gemacht werden kann. Während die Nutzung von Nebenläufigkeit auf Microcode- und Betriebssystemebene ohne Zutun des Benutzers gewährleistet wird, muß der Programmierer für die Nutzung algorithmischer Parallelität selbst sorgen.

Um algorithmische Parallelität nutzbar zu machen, müssen drei Voraussetzungen erfüllt sein. Es muß eine geeignete *Zerlegung* des Problems in unabhängige Teilprobleme erfolgen, es bedarf geeigneter *paralleler Programmiermodelle* zur Formulierung des parallelen Programms, und schließlich braucht man geeignete *Hardware*, auf der parallele Programme ausgeführt werden können. Diese drei Aspekte werden in den folgenden drei Abschnitten beleuchtet. Anschließend werden in Abschnitt 2.1.4 einige grundlegende Begriffe und Zusammenhänge für die Bewertung von parallelen Programmen vorgestellt.

2.1.1 Zerlegung in nebenläufige Teilprobleme

Um einen parallelen Algorithmus zu entwerfen, muß das Problem in voneinander unabhängig berechenbare Teilprobleme zerlegt werden. Man unterscheidet zwei grundlegende Zerlegungsprinzipien, zwischen denen jedoch eine Vielzahl von Mischformen möglich sind.

Funktionale Dekomposition
: Bei diesem Zerlegungsprinzip werden unabhängige Teilfunktionen identifiziert. Ein Beispiel dafür sind die Teilschritte beim Pipelining in RISC Prozessoren. Offenbar erlaubt dieses Prinzip nur die Nutzung von so viel Prozessor-Elementen (PEs), wie Teilfunktionen identifiziert werden.

Datenparallelismus
: Dieses Zerlegungsprinzip findet oft Anwendung, wenn gleichartige Funktionen auf einem großen Datensatz ausgeführt werden müssen, etwa bei der Berechnung einer Vektorsumme. Jedes PE bearbeitet dabei eine Teilmenge des Datensatzes, wobei alle PEs dieselbe Funktion (z.B. die Summenberechnung) ausführen. Genügend Daten vorausgesetzt, ermöglicht dieses Dekompositionsschema die effiziente Nutzung einer großen Anzahl von PEs.

Nur in einfachsten Fällen sind Compiler in der Lage, bestehende (sequentielle) Programme auf Daten- und Kontrollflußabhängigkeiten zu untersuchen und in entsprechende parallele Programme umzuwandeln. Oft ist es jedoch möglich und nötig, den Programmablauf zu modifizieren, um mehr Nebenläufigkeit zu erschließen. Derartiges wird jedoch nie von einem Compiler geleistet werden können, denn woher sollte dieser wissen, ob das modifizierte Programm das ursprüngliche Problem immer noch löst? Dies kann bereits an dem einfachen Problem der Maximumsbestimmung verdeutlicht werden. Bestimmt man das maximale Element in einem Feld mit einem sequentiellen Computer, so wird dies üblicherweise mit einer Schleife über alle Elemente implementiert. Gibt es nun in dem Feld mehrere Elemente von denen der maximale Wert angenommen wird, so findet der Algorithmus je nach Implementierung das *erste* oder *letzte* maximale Element. Ein parallelisierender Compiler müßte dies genau nachbilden, auch wenn für die Korrektheit des Programmes die Bestimmung eines beliebigen maximalen Elements hinreichend wäre. Ein paralleler Algorithmus, bei dem jedes PE ein maximales Element aus einer Teilmenge aller Elemente bestimmt, von denen anschließend eines ausgewählt wird, könnte z.B. das zeitlich zuerst oder das von dem PE mit der kleinsten Nummer gefundene auswählen. Dementsprechend kann auch der Ablauf des parallelen Programmes von dem des sequentiellen abweichen. In Kapitel 4 wird dies bei den Testläufen der parallelen Implementierungen des Simplex-Algorithmus erkennbar.

Unabhängig von dem gewählten Dekompositionsprinzip bedarf es zur Koordinierung der Arbeit geeigneter *Kommunikation* und *Synchronisation*. Beides führt i.a. zu einem zusätzlichen Zeitaufwand zum reinen Rechenaufwand. Außerdem können Wartezeiten entstehen, wenn einige Prozessoren auf das Eintreffen von Daten oder die Synchronisation mit

anderen PEs warten müssen. Deshalb ist die mittlere Rechenzeit zwischen aufeinanderfolgenden Kommunikationen bzw. Synchronisationen eine wichtige Kenngröße von parallelen Algorithmen, die man die *Granularität* eines parallelen Algorithmus nennt. Man unterscheidet zwischen feiner, mittlerer und grober Granularität, wobei jedoch keine definierte Zuordnung vorliegt.

2.1.2 Parallele Hardware

Erste Computer, die Parallelität nutzen, wurden in Form von Vektorrechnern entwickelt. Sie trugen der Tatsache Rechnung, daß häufig Operationen auf Vektoren hoher Dimension ausgeführt werden, was sich gut zum Pipelining eignet. Dafür wurden sog. Vektoroperationen in den Prozessor-Befehlssatz aufgenommen, die eine Pipeline-Bearbeitung initiieren. Dies verhalf Vektorrechnern zu einem beachtlichen Geschwindigkeitsvorteil gegenüber sequentiell arbeitenden Mikroprozessoren, der nun aber durch die Verbesserung von RISC-Prozessoren immer weiter dahinschwindet [22].

Die Idee lag also (nicht zuletzt auch aus Kostengründen) nahe, viele leistungsfähige RISC-Prozessoren zu einem Parallelrechner zusammenzuschließen. Damit erschließt man über vektorartige hinaus noch anders strukturierte Parallelität, zumal moderne Algorithmen oft kompliziertere Datenstrukturen statt dicht besetzter Vektoren verwenden, was eine Vektorisierung ausschließt.

Es gibt eine Vielzahl technischer Möglichkeiten, einen Zusammenschluß von Mikroprozessoren zu einem Parallelrechner auszuführen, die jeweils Vorzüge für gewisse Anwendungsgebiete aufweisen. Die wichtigsten Unterscheidungsmerkmale verschiedener Parallelrechnerarchitekturen bestehen beim *Kontrollfluß* und dem *Speicherlayout*.

2.1.2.1 Kontrollfluß

Flynn führte folgende grobe Klassifizierung des Kontrollflusses von (Parallel-)Rechner-Architekturen ein [72]:

SISD (*single instruction-stream single data*)
Hierbei handelt es sich um einen normalen sequentiellen Computer.

SIMD (*single instruction-stream multiple data*)
Alle PEs eines solchen Parallelrechners führen in jedem Takt dieselbe Instruktion aus (oder überspringen diese), allerdings jede auf eigenen Daten. Beispiele für solche Parallelrechner sind MASPAR und die Connection Machine CM-2 von Thinking Machines Inc.. Heute gibt es jedoch keine aktuellen Implementierungen dieses Modells mehr, da es sich als zu restriktiv erwiesen hat.

MIMD (*multiple instruction-streams multiple data*)
Bei solchen Parallelrechnern verfügt jedes PE über einen eigenen Kontrollfluß und bearbeitet eigene Daten. Insbesondere kann jedes Workstation-Cluster als MIMD-Rechner aufgefaßt werden, was Ende der 80-er Jahre dazu führte, spezielle MIMD-Rechner als unnötige Dinosaurier des Supercomputing abzutun. Es hat sich jedoch herausgestellt, daß nur solche Speziel-Rechner auch über genügend Kommunikationsleistung verfügen, um weite Applikationsfelder abzudecken.

In dieser Notation bleibend, hat man mit SPMD (*single program multiple data*) eine besondere "Betriebsart" von MIMD-Rechnern benannt, bei der alle PEs dasselbe Programm bearbeiten. Dies ist eine gute Modellierung eines SIMD-Rechners auf einer MIMD-Maschine. Sie bietet sowohl Vor- als auch Nachteile gegenüber einem echten SIMD-Rechner. Der Nachteil besteht darin, daß keine implizite Synchronisierung aller PEs vorliegt, so daß explizit synchronisiert werden muß. Dafür kann aber im SPMD-Fall bei einer Verzweigung, die bei verschiedenen PEs unterschiedlich ausfällt, jedes PE einen eigenen Zweig ausführen entsprechend dem lokalen Ergebnis der Testbedingung, während dies bei einer SIMD-Maschine zu einer gewissen Sequentialisierung führt. Die damit verbundenen Geschwindigkeitseinbußen haben die Abkehr von SIMD weiter befördert.

2.1.2.2 Speichermodelle

Neben dem Kontrollfluß ist das Speicherlayout ein wichtiges Unterscheidungskriterium für Parallelrechner. Man unterscheidet grob zwischen Parallelrechnern mit

- *gemeinsamem Speicher*, bei denen alle PEs auf den gesamten Speicher zugreifen können, und solchen mit
- *verteiltem Speicher*, bei denen jedes PE über eigenen Speicher verfügt, auf das es auch als einziges zugreifen kann.

Wichtig ist diese Unterscheidung aus zweierlei Hinsicht. Vom technischen Standpunkt ist es (in finanziell realistischen Grenzen) nicht möglich, Parallelrechner mit gemeinsamem Speicher für mehr als heute etwa 16 Prozessoren zu realisieren. Noch mehr Prozessoren blockieren sich gegenseitig beim Zugriff auf die Daten im Speicher, was die Skalierbarkeit (vgl. Abschnitt 2.1.4) beeinträchtigt.

Dagegen können Parallelrechner mit verteiltem Speicher aus mehreren tausend PEs zusammengesetzt werden, die mit einem Kommunikationsnetzwerk verbunden sind. Somit sind Parallerechner mit verteiltem Speicher besonders für solche Algorithmen geeignet, die gut skalieren.

Parallelrechner mit gemeinsamem Speicher gelten als die leichter zu programmierenden Computer. Dies gilt besonders für die Parallelisierung bestehender Programme, denn bei

gemeinsamem Speicher ist es möglich, ausgehend vom sequentiellen Programm, die Parallelisierung schrittweise zu vollziehen. Die gute Skalierbarkeit von Parallelrechnern mit verteiltem Speicher gegenüber der einfacheren Programmierbarkeit von Architekturen mit gemeinsamem Speicher hat zu vielerlei Entwicklungen geführt, physikalisch verteilten Speicher logisch als gemeinsamen Speicher addressierbar zu gestalten (virtual shared memory).

2.1.3 Parallele Programmiermodelle

Die Nutzung algorithmischer Parallelität erfordert neue Programme, bei der die Arbeit mehrerer Funktionseinheiten geeignet koordiniert wird. Zur Formulierung derartiger Programme, wurden verschiedene sog. *parallele Programmiermodelle* entwickelt. Darunter versteht man die Menge der bereitgestellten Möglichkeiten zur Formulierung von

- Nebenläufigkeit,
- Kommunikation und
- Synchronisation.

Prinzipiell sind die verschiedenen Programmiermodelle gleich mächtig. Sie unterscheiden sich jedoch erheblich in dem Komfort bei ihrer Anwendung für verschiedene Algorithmen und der effizienten Implementierbarkeit für verschiedene Parallelrechnerarchitekturen. Insbesondere können sie deshalb nicht losgelöst von der jeweils zugrundeliegenden Architektur betrachtet werden.

Bei verteiltem Speicher müssen Daten von einem PE zum anderen gelangen, sie müssen kommuniziert werden. *Message-Passing* ist ein Programmiermodell, das die Kommunikation explizit dem Programmierer überläßt. Die Synchronisation erfolgt dann implizit über die Kommunikation, indem ein PE auf die eintreffenden Daten wartet. Bei diesem Programmiermodell unterscheidet der Programmierer explizit zwischen schnellen lokalen Speicherzugriffen und dem langsameren Zugriff auf entfernte Daten. Dies fördert die Lokalität und damit die Effizienz solcher parallelen Programme auf Parallelrechnern mit verteiltem Speicher.

Dasselbe Programmiermodell könnte auch für einen Parallelrechner mit gemeinsamem Speicher verwendet werden. Allerdings bedeutet das Packen und Verschicken von Nachrichten zwischen den Prozessen einen unnötigen Aufwand gegenüber dem direkten Zugriff auf die Daten, der ja für alle PEs möglich ist. Solch ein direkter Zugriff muß allerdings aus Gründen der Datenkonsistenz synchronisiert werden. Das *Shared-Memory* Programmiermodell stellt daher explizite Synchronisationsmöglichkeiten zur Verfügung, während eine Kommunikation implizit über Synchronisation realisiert wird. Tabelle 2.1 faßt die Eigenschaften beider parallelen Programmiermodelle zusammen.

Programmiermodell	Synchronisation	Kommunikation
Message-Passing	implizit	explizit
Shared-Memory	explizit	implizit

Tabelle 2.1: Realisierung von Synchronisation und Kommunikation bei verschiedenen parallelen Programmiermodellen.

Darüberhinaus gibt es noch eine Vielzahl anderer paralleler Programmiermodelle, auf die nicht näher eingegangen wird, da sie für die Parallelisierung des Simplex-Algorithmus ungeeignet sind oder für die verfügbaren Parallelrechner nicht bereitstehen.

2.1.4 Grundbegriffe zur Bewertung paralleler Algorithmen

Es bezeichne T_s die Ausführungszeit eines Algorithmus auf einem Prozessor und T_p die Zeit die der Algorithmus auf p Prozessoren benötigt, dann nennt man

$$S = \frac{T_s}{T_p} \tag{2.1}$$

die *Beschleunigung* (speedup) des Algorithmus für p Prozessoren. Beschränkt man sich auf Algorithmen, bei denen alle parallelen Algorithmen dieselbe Arbeit verrichten[1], und vernachlässigt Caching-Effekte, so ist offenbar p die maximal erreichbare Beschleunigung. Doch wie nahe kommt man an diese Schranke? Das Amdahl'sche Gesetz gibt eine obere Schranke für die erreichbare Beschleunigung an [1].

Satz 32 (Amdahl'sches Gesetz)
Sei $T_s = T_0 + T_{||}$ die Zeit für die sequentielle Ausführung eines Algorithmus, wobei der Anteil T_0 im Gegensatz zu $T_{||}$ nicht parallelisierbar sei. Dann ist die maximal erzielbare Beschleunigung (unabhängig von der Anzahl der PEs)

$$S \leq \frac{T_s}{T_0}. \tag{2.2}$$

Beweis:
Für p Prozessoren gilt $T_p = T_0 + T_{||}/p$ woraus mit (2.1) für $p \to \infty$ die Behauptung folgt. ■

[1] Anders ist es z.B. bei manchen Suchalgorithmen wie Branch-and-Bound. Wenn im parallelen Fall das erste PE das gesuchte Element findet, so terminiert der Algorithmus evtl. schon, nachdem insgesamt weniger Elemente untersucht wurden als im sequentiellen Fall. Somit leisten der parallele und der sequentielle Algorithmus nicht dieselbe Arbeit — ein superlinearer Speedup, d.h. $S > p$, ist möglich.

Diese Schranke ist nur theoretisch zu verstehen. Im Normalfall wird sie nicht erreicht, weil durch zusätzlichen Aufwand bei der Parallelisierung (etwa zur Kommunikation oder Synchronisation) zusätzlich Zeit vergeht, in der kein Beitrag zur Lösung geleistet wird.

Die Schranke kann aber auch überschritten werden, ja sogar eine superlineare Beschleunigung ist möglich. Dies liegt am Cache heutiger Computerarchitekturen. Typischerweise verfügt jeder Prozessor eines Parallelrechners auch über eigenen Cache (dies gilt nicht für den T3D von CRAY). Dadurch können bei mehreren Prozessoren mehr Daten im Cache liegen und somit schneller bearbeitet werden, als es ein einzelner Prozessor (mit seinem beschränkten Cache) in der Lage wäre.

Dem etwas entmutigenden Amdahl'schen Gesetz, daß die Skalierbarkeit stark einschränkt, setzte Gustavson eine Beziehung entgegen, die die Nützlichkeit des parallelen Rechnens für die Berechnung größerer Probleme beschreibt (scale-up) [56]. Dabei geht er davon aus, daß sich beim Übergang zu größeren Probleminstanzen i.a. der parallele Anteil der Arbeit erhöht, sein sequentieller Anteil jedoch konstant bleibt. Wenn also auch die Beschleunigung bei der Lösung eines Problems beschränkt bleibt, so ermöglicht Parallelität die Lösung größerer Probleminstanzen.

Satz 33 (Gustavson's Gesetz)
Sei N ein Maß für die Größe einer Probleminstanz. Sei $T_s = T_0 + Np$ die Zeit die ein einzelner Prozessor zu dessen Lösung beansprucht, wobei $T_{\|} = Np$ den parallelisierbaren Anteil bezeichnet und der inherent sequentielle Anteil T_0 konstant sei. Dann ist die mit p Prozessoren erreichbare Beschleunigung

$$S \geq p + (1-p)\frac{T_0}{N}.$$

Beweis:
Für p Prozessoren gilt $T_p = T_0 + T_{\|}/p = T_0 + N$ woraus mit (2.1) $S = (T_0 + Np)/(T_0 + N) = p + (1-p)T_0/(T_0 + N) \geq p + (1-p)T_0/N$ wegen $p \geq 1$ folgt. ■

Ende der 80er Jahre versuchte Gustavson mit seiner Beziehung einen „mental block" gegen „massive Parallelität" zu bekämpfen [56]. Zu dieser Zeit erwartete man für die Zukunft Parallelrechner, die aus tausenden bis millionen Prozessoren zusammengesetzt sind. Heute hat man von diesen Vorstellungen weitgehend Abstand genommen und versteht unter „massiver Parallelität" eher einige Hundert PEs.

Man definiert ferner die *Effizienz* eines parallelen Algorithmus als

$$E = \frac{S}{p}. \tag{2.3}$$

Sie ist ein Maß dafür, in wie weit die PEs eines Parallelrechners tatsächlich genutzt werden. Der Begriff der *Skalierbarkeit* beschreibt, wie sich die Effizienz mit zunehmender Prozessoranzahl verhält: Ein paralleles Programm heißt skalierbar, wenn seine Effizienz bei Hinzunahme weiterer PEs nur "wenig abnimmt".

Es gibt noch eine Vielzahl weiterer Maße zur Bewertung von parallelen Algorithmen. Sie alle vorzustellen, ist nicht das Ziel dieser Einführung. Stattdessen haben wir uns auf die in dieser Arbeit relevanten Größen beschränkt.

2.2 Nebenläufigkeit in Simplex-Algorithmen

Das Amdahl'sche Gesetz (Satz 32) gibt eine obere Schranke für die maximal erreichbare Beschleunigung durch Nutzung von Nebenläufigkeit. Vor einer Parallelisierung des Simplex-Algorithmus sollte man deshalb untersuchen, wo der Algorithmus wieviel Zeit verbraucht und ob die für den Hauptanteil verantwortlichen Teile überhaupt parallelisiert werden können. Tabelle 2.2 zeigt die Aufteilung der Laufzeit für zwei verschiedene Test-LPs.

Problem	fit2d	stocfor3
Anzahl Zeilen	25	16675
Anzahl Spalten	10500	15695
Pricing	0.007%	30.6%
Quotiententest	23.3%	2.1%
Matrix-Vektor-Produkt	66.6%	0.5%
LU-Faktorisierung	0.002%	3.1%
Lineare Gleichungssystem	0.02%	58.5%

Tabelle 2.2: Zeitverbrauch des dualen Simplex-Algorithmus mit steepest-edge Pricing für zwei Beispielprobleme bei Verwendung einer Spaltenbasis.

Offenbar unterscheidet sich die Zeitverteilung für beide LPs erheblich, so daß zu erwarten ist, daß es bei der Parallelisierung unterschiedlicher Konzepte bedarf. Pricing, Quotiententest und Matrix-Vektor-Produkt eignen sich aufgrund ihrer mittleren Granularität zu einem datenparallelen Ansatz. Für das LP fit2d nehmen diese Teile insgesamt über 99% des gesamten Zeitverbrauchs in Anspruch. Damit kann für eine geeignete Hardware nach dem Amdahl'schen Gesetz eine gute Skalierbarkeit erwartet werden.

Dagegen wird man für das Problem stocfor3 bei dieser Art der Parallelisierung nach dem Amdahl'schen Gesetz die Lösungszeit auf maximal 70% reduzieren können. Um darüber hinaus zu kommen, muß eine Parallelisierung auch bei der Lösung linearer Gleichungssysteme ansetzen. Dazu werden zwei Wege unterschiedlicher Granularität verfolgt. Auf mittlerer bis grober Granularität wird der Simplex-Algorithmus so umstrukturiert, daß mehrere Gleichungssysteme gleichzeitig gelöst werden (vgl. Abschnitte 2.2.3 und 2.2.4). Als zweiter Weg wird ein paralleler Löser für lineare Gleichungssysteme entwickelt (vgl. Abschnitt 2.3). Dabei ist die Granularität für LP-Probleme jedoch so fein, daß keine Beschleunigung erzielt wird. Für Matrizen aus anderen Bereichen hingegen erreicht die Implementierung eine gute Beschleunigung.

In den folgenden Abschnitten wird die Nebenläufigkeit aufgezeigt, die sich zur Paralle-

lisierung der im ersten Kapitel beschriebenen Simplex-Algorithmen bietet. Dies geschieht exemplarisch für Basen in Spaltendarstellung, wobei jedoch alles auf die Verwendung einer Zeilenbasis übertragbar ist. Dabei bezeichnet p die Anzahl der zur Verfügung stehenden PEs, die von 0 bis $(p-1)$ durchnumeriert seien. Ferner seien n und m die Anzahl der Spalten bzw. Zeilen der Nebenbedingungsmatrix A des zu lösenden LPs.

2.2.1 Paralleles Matrix-Vektor-Produkt

In Schritt 2 oder 4 der Simplex-Algorithmen wird das Matrix-Vektor-Produkt

$$\Delta g^T = \Delta h^T A \tag{2.4}$$

berechnet. Diese Operation kann ohne Kommunikation oder Synchronisation nebenläufig durchgeführt werden, indem jedes PE das Produkt für eine Teilmenge der Spalten von A berechnet. Im einfachsten Fall bearbeitet PE i die Spalten $\lfloor (i-1) \cdot n/p \rfloor, ..., \lfloor i \cdot n/p \rfloor$. Dies führt i.a. jedoch nur für eine dichtbesetzte Matrix A zu einer effizienten Implementierung. Im dünnbesetzten Fall können die NNEs ungleichmäßig auf die Spalten von A verteilt sein, so daß einigen PEs mehr Arbeit zufällt als anderen. Dadurch warten letztere auf die, denen mehr Arbeit zugewiesen wurde.

Dieses Problem wird durch den Begriff des *Lastausgleichs* (load balancing) umschrieben. Die Aufgabe des Lastausgleichs ist es, allen PEs in etwa gleich viel Arbeit zuzuweisen, damit keine unnötigen Wartezeiten auftreten und so die Arbeit insgesamt möglichst schnell bewältigt wird. Man unterscheidet zwischen statischem und dynamischen Lastausgleich. Während im ersten Fall die Arbeit zu Beginn aufgeteilt wird und diese Aufteilung anschließend unverändert bleibt, wird beim dynamischen Lastausgleich anhand von Laufzeitdaten die Aufteilung dynamisch angepaßt.

Für das Matrix-Vektor-Produkt im Simplex-Algorithmus wird ein statisches Verteilungsschema gewählt, um keinen zusätzlichen Aufwand zu erzeugen. Die Zerlegung der Indexmenge ist jedoch für beide Rechnerarchitekturen unterschiedlich.

Bei der Parallelisierung für Multiprozessoren mit verteiltem Speicher, erfolgt die Verteilung durch Aufteilung der Nebenbedingungsmatrix auf die Prozessoren. Die dabei gewählte Aufteilung hat jedoch noch Implikationen für die Parallelisierung anderer Operationen im Simplex-Algorithmus als das Matrix-Vektor-Produkt, nämlich beim Update des Vektors g und beim Pricing- bzw. Quotiententest (vgl. Abschnitt 2.2.2), aber auch bei der Aktualisierung der Werte ρ_i beim steepest-edge Pricing. Daher kommt der Verteilung eine besondere Bedeutung zu.

Für DoPlex hat sich eine zyklische Verteilung als günstig erwiesen. Diese ist durch

$$pe(i) = i \mod p \tag{2.5}$$

bestimmt, wobei $pe(i)$ die Nummer des PEs angibt, das die i-te Spalte von A speichert. Durch die Verteilung der Daten auf die PEs wird implizit die Parallelisierung der Arbeit

vorgenommen: Jedes PE berechnet den Teil des Matrix-Vektor-Produktes, zu dem es die Vektoren von A speichert.

Das mit der Verteilung des LPs und damit der Arbeit gemäß (2.5) ein vernünftiger Lastausgleich erzielt wird, entspricht der Beobachtung, daß meist ähnliche Strukturen des LPs in der Nebenbedingungsmatrix nahe beieinander zu finden sind. Jede Struktur wird daher gleichmäßig auf alle PEs verteilt, so daß jedes PE von jeder Struktur einen Teil verwaltet. Dadurch wird insbesondere auch vermieden, daß einige PEs nicht an dem Lösungsprozess teilnehmen, weil sie zufällig an den relevanten Strukturen nicht partizipieren. Davon wird in Abschnitt 2.2.3 noch die Rede sein.

Die Version für Parallelrechner mit gemeinsamem Speicher, SMoPlex, arbeitet auf derselben Datenstruktur wie die sequentielle Implementierung SoPlex. Dadurch wird eine Verteilung der Arbeit von (2.4) gemäß (2.5) nicht mehr sinnvoll möglich. Sie würde es erforderlich machen, daß die Matrix spaltenweise traversiert wird, und jedes Δg_i als $\Delta g_i = \Delta h^T \cdot A_{.i}$ berechnet wird. Da aber Δh in der Regel dünnbesetzt ist, werden so mehr Rechenoperationen durchgeführt, als wenn man

$$\Delta g = \sum_{\Delta h_i \neq 0} \Delta h_i A_{i.}$$

berechnet.

Um solch eine zeilenweise Berechnung auch im parallelen Fall durchzuführen, ist es sinnvoller, A in Blöcke zu unterteilen, die jeweils zeilenweise bearbeitet werden. Die Blöcke werden dabei so erzeugt, daß in jedem Block etwa gleichviel NNEs sind. In der Regel entstehen so Blöcke mit unterschiedlich vielen Spalten von A.

2.2.2 Paralleles Pricing und Quotiententest

Beim Pricing und Quotiententest handelt es sich um einen einfachen Suchalgorithmus, bei dem aus einer Menge von Werten ein maximales (bzw. minimales) Element bestimmt wird. Die Parallelisierung solch eines Suchalgorithmus liegt auf der Hand. Man zerlegt die Menge in Teilmengen, aus denen jeweils ein PE das maximale (minimale) Element bestimmt. Anschließend wird von den p Elementen das beste ausgewählt. Diese Auswahl kann mit verschiedenen Algorithmen erfolgen, wobei der günstigste jeweils von der zugrundeliegenden Hardware abhängt.

Im verteilten Fall, ist die Aufteilung des Suchraumes bereits durch die Verteilung des LPs und damit einhergehend von den Vektoren g und Δg vorgegeben. Hat nun jedes PE das lokale maximale bzw. minimale Element bestimmt, gilt es, aus diesen p Kandidaten das global beste zu finden und allen PEs bekanntzugeben. Einen Algorithmus der solches leistet nennt man *Gossiping*. Eine optimaler Algorithmus für dieses Problem [73] ist als Algorithmus 11 für das i-te von $p = 2^{\bar{p}}$ PEs angegeben. Die anderen PEs führen gleichzeitig denselben Algorithmus aus, wobei i die jeweilige PE-Nummer ist. Die Operation $c = a$ xor

b berechnet bitweise das ausschließende Oder von den im Dualsystem dargestellten Zahlen a und b. Sie wird von heutigen Prozessoren direkt unterstützt.

ALGORITHMUS 11 (GOSSIPING)
Sei g das lokale Datum des i-ten PEs.
Für $j = 0, \ldots, \bar{p} - 1$:

Schritt 1: Setze
$k \leftarrow (i \text{ xor } 2^j)$

Schritt 2: Sende g zu PE k und
Empfange l von PE k

Schritt 3: Setze
$g \leftarrow l \circ g$

Nach Termination enhält g auf allen PEs das globale Ergebnis. Für den Fall der Maximumsbildung berechnet die Operation $l \circ g = \max\{l, g\}$. Allgemein kann für „$\circ$“ jedoch jede kommutative und assoziative Operation verwendet werden. Andere mögliche Operationen währen z.B. die Berechung einer Summe oder eines Produktes. Die Assoziativität der Gossiping-Operation „$\circ$“ ist wichtig, da die Operationsreihenfolge von der Kommunikationsstruktur und nicht von der Operation vorgegeben wird.

Die Arbeitsweise des Gossiping-Algorithmus ist in Abbildung 2.2 dargestellt. Er arbeitet in $\bar{p}$ Phasen, die jeweils aus einer zweiseitigen Kommunikation und einer Berechnung der Gossip-Operation „$\circ$“ besteht. In Schritt 1 wählt PE i einen Partner k für die Kommunikation. Dies geschieht so, daß PE k gerade PE i als Partner wählt. Beide PEs tauschen in Schritt 2 ihre lokalen Daten g aus und führen mit den eigenen und den vom Partner empfangenen Daten die Gossip-Operation aus. Da diese kommutativ ist, ist das Ergebnis für beide PEs dasselbe. In der nächsten Iteration wählen sich i und k neue Partner so, daß die jeweiligen PEs gerade in der vorigen Iteration ihre Daten abgeglichen haben. Dadurch haben anschließend schon vier PEs dasselbe Ergebnis bestimmt. Dies wird fortgesetzt, bis die Daten aller $2^{\bar{p}}$ PEs abgeglichen sind.

Die Struktur des Gossiping-Algorithmus besteht aus p ineinander verschachtelten Fan-in- und Fan-out-Bäumen. Von jedem PE geht ein binärer Fan-out-Baum los, der nach $\bar{p}$ Schritten alle PEs erreicht hat. Gleichzeitig ist jedes PE der Endpunkt eines binären Fan-in-Baumes, der sich von allen PEs nach $\bar{p}$ Schritten auf das eine zusammenzieht. Somit erreicht die Information jedes PEs jedes andere.

Für Parallelrechner mit gemeinsamem Speicher könnte im Prinzip ein Äquivalent zum Gossiping-Algorithmus implementiert werden, wobei an die Stelle der Kommunikation die Synchronisation tritt. Da solche Rechner nur aus wenigen PEs bestehen (vgl. Abschnitt 2.1.2.2) muß das beste Element nur unter wenigen Kandidaten ausgewählt werden. Diese Aufgabe wird aber schneller von einem einzigen PE mit einer sequentiellen Schleife ausgeführt, in der keine (zeitaufwendige) Synchronisation notwendig ist. Deshalb

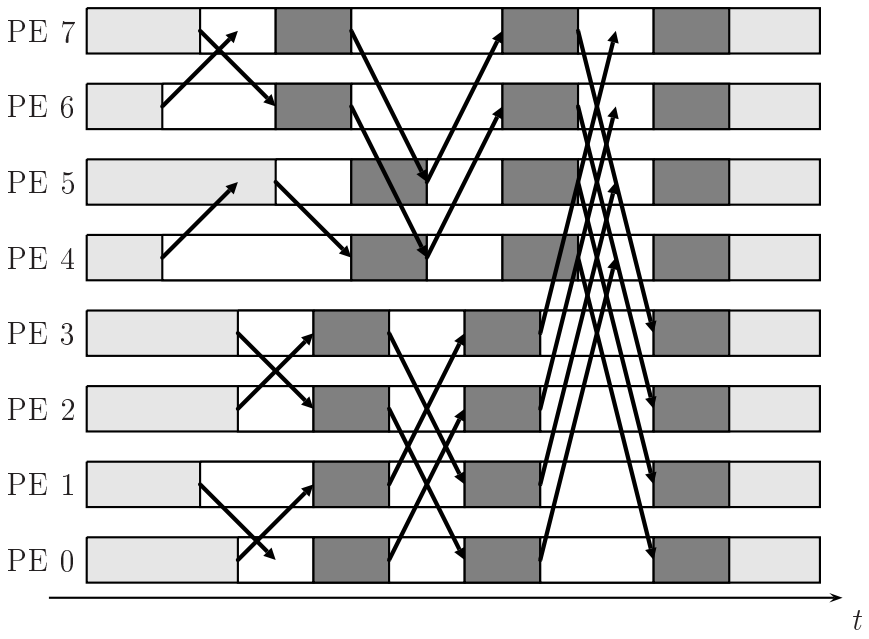


Abbildung 2.2: Kommunikations- und Bearbeitungsstruktur von Gossiping-Algorithmen für $p = 8$ PEs. Jede Säule repräsentiert den Zeitverlauf eines PEs. Graue Bereiche darin symbolisieren die Berechnung der Gossip-Operation „$\circ$“, während weiße Bereiche die durch Kommunikation bedingten Wartezeiten darstellen. Die Nachrichten sind durch Pfeile dargestellt. Die Kommunikationsstruktur wird am besten als p ineinander verflochtene binäre Bäume beschrieben. Bei jeder Kommunikation werden zwei PEs synchronisiert, so daß am Ende alle PEs synchronisiert sind.

---

schreibt jedes PE sein Maximum in eine Variable und wartet an einem sog. Barrier-Synchronisationspunkt, bis alle diesen erreichen. Dann wählt ein PE das globale Maximum aus.

2.2.3 Block-Pivoting

Das nun vorzustellende Block-Pivoting unterscheidet sich grundsätzlich von den in den vorigen Abschnitten beschriebenen Parallelisierungsansätzen, die einen datenparallelen Ansatz verfolgen. Beim Block-Pivoting handelt es sich hingegen um eine Umformulierung des Simplex-Algorithmus derart, daß auf funktionaler Ebene mehr Nebenläufigkeit entsteht. Eine solche Parallelisierung wird nie von einem Compiler erbracht werden können.

Das Block-Pivoting begegnet dem Problem, daß mit den Parallelisierungen aus den vorigen Abschnitten zwar gute Beschleunigungen der Lösung von LPs mit ähnlicher Struktur wie „fit2d“ zu erwarten sind, jedoch nicht, wenn die Lösung von linearen Gleichungssystemen einen großen Anteil des Rechenaufwandes beansprucht. Für diesen Fall erschließt es

zusätzliche Nebenläufigkeit, bei der mehrere Gleichungssysteme parallel gelöst werden.

Grundlage ist eine Beobachtung, die bereits für das multiple Pricing ausgenutzt wurde, daß nämlich beim Pricing wählbare Indizes nach einer Simplex-Iteration oft wählbar bleiben. Nach jedem Pricing-Schritt wird zu dem gewählten Index ein Gleichungssystem mit der aktuellen Basismatrix gelöst. Wählt man statt eines einzigen Index eine Menge von p Indizes, so können alle Prozessoren je ein lineares Gleichungssystem für jeden Index lösen. Anschließend wird zunächst nur ein Index für die Simplex-Iteration verwendet. Ist beim nächsten Pricing einer der Indizes der vorigen Iteration weiterhin wählbar, so wurde das zugehörige Gleichungssystem bereits (fast) gelöst, wenn auch mit der Basismatrix aus der vorigen Iteration. Die Lösung muß also lediglich auf die neue Basismatrix aktualisiert werden, was i.a. einen wesentlich geringeren Arbeitsaufwand erfordert. Die Aktualisierung wird gleich für alle noch wählbaren Indizes durchgeführt, damit weniger Wartezeiten anfallen wenn in den folgenden Iterationen andere Indizes gewählt und für diese die Lösungen aktualiesiert werden. Dies wird so lange fortgesetzt, bis kein wählbarer Index mehr verfügbar ist. In diesem Fall wird eine neue Indexmenge bestimmt.

Der folgende Satz stellt die Gleichungen für die Aktualisierung der Lösungsvektoren bei einer Zeilenbasis auf. Die entsprechenden Resulate für eine Spaltenbasis folgen einfach durch Transposition. Da für den einfügenden und den entfernenden Simplex unterschiedliche Typen von Gleichungssystemen gelöst werden müssen, werden beide Typen betrachtet.

Satz 34
Sei $D \in \mathbb{R}^{n\times n}$ regulär, $\Delta f^T = r^T D^{-1}$ und $\Delta h = D^{-1}e_p$. Dann gilt für die Matrix $D' = D + e_p(r^T - D_{p.})$, die aus D durch Austausch der p-ten Zeile mit r^T hervorgeht, und $j \neq p$

$$(D')^{-1}e_j = D^{-1}e_j + \frac{\Delta f_j}{\Delta f_p}\Delta h \tag{2.6}$$

$$d^T(D')^{-1} = d^T D^{-1} + \frac{-(d^T D^{-1})_p}{\Delta f_p}(\Delta f - e_p^T) \tag{2.7}$$

Beweis:
Gleichung (2.7) enspricht (1.10) aus Satz 5, wurde also bereits bewiesen. Dort wurde auch gezeigt, daß $D' = VD$ für $V = I + e_p(r^T D^{-1} - e_p^T)$ gilt. Nach (1.89) ist demnach

$$\begin{aligned}(D')^{-1}e_j &= D^{-1}V^{-1}e_j \\ &= D^{-1}\left(e_j + \tfrac{\Delta f_j}{\Delta f_p}e_p\right) \\ &= D^{-1}e_j + \tfrac{\Delta f_j}{\Delta f_p}\Delta h,\end{aligned}$$

was Gleichung (2.6) beweist.

■

Die Vektoren Δf und Δg werden bei dem Pivot-Schritt ohnehin berechnet. Damit beschränkt sich der Rechenaufwand auf die Addition des Vielfachen eines Vektors und ist damit wesentlich niedriger als der für die Lösung eines Gleichungssystems. Gleichung (2.7) wird für den einfügenden Simplex benötigt, während (2.6) beim entfernenden Algorithmus angewendet wird.

Um überhaupt genügend wählbare Pivot-Indizes zu finden, muß die Suche nach den einzelnen Indizes geeignet aufgeteilt werden. Hier hat sich die Verteilung gemäß (2.5) bewährt. Dagegen wurde bei einer blockweisen Verteilung oft nur ein wählbarer Index gefunden, denn alle wählbaren Indizes liegen häufig nah beieinander und somit in einem Block. Deshalb wird auch bei der Implementierung für Parallelrechner mit gemeinsamem Speicher die Arbeit nach (2.5) verteilt und somit anders als für die Berechnung des Matrix-Vektor-Produktes.

Damit im verteilten Fall die PEs überhaupt die Gleichungssysteme lösen können, ist es notwendig, daß jedes PE auch die Basis-Matrix samt ihrer LU-Zerlegung lokal zur Verfügung hat. Dazu muß der jeweils in die Basis eintretende Vektor allen PEs zugängig gemacht werden. Dies verursacht einen zusätzlichen Kommunikationsaufwand.

Man beachte, daß Block-Pivoting den Ablauf des Algorithmus beeinflußt. Während im sequentiellen Fall das Pricing in jeder Iteration den (im Sinne der jeweiligen Pricing-Strategie) besten Index auswählt, werden beim Block-Pivoting zunächst die p besten Indizes gewählt. In der $(p-1)$-ten Iteration danach, wird somit der pt-beste Index einer frühreren Iteration als Pivot-Index verwendet. Der sequentielle Pricer würde wahrscheinlich einen anderen Index präferieren. Inwieweit sich dies auf die Anzahl der Iterationen auswirkt, hängt von dem zu lösenden LP ab. Im allgemeinen wird man eine Erhöhung erwarten, aber auch Verbesserungen werden beobachtet.

Einen nicht zu vernachlässigen Nachteil hat das Block-Pivoting in Bezug auf die LU-Updates der Basismatrix. Da beim einfügenden Simplex nicht mehr vor dem Einfügen eines Vektors r in die Basis das Gleichungssystem $B^{-1}r$ gelöst wird, steht auch der Vektor $L^{-1}r$ nicht zur Verfügung, der für den Forest-Tomlin Update notwendig ist. Diesen Vektor explizit neu zu berechnen, impliziert einen nicht vertretbaren Rechenaufwand, so daß beim Block-Pivoting für den einfügenden Simplex-Algorithmus der PF Update benutzt werden muß.

M. Padberg gibt in [80] eine andere Interpretation für das Block-Pivoting. Dabei steht die Frage im Vordergrund, wie der einfügende Simplex-Algorithmus umzuformulieren wäre, wenn pro Iteration statt nur einem Pivot-Index eine Menge C von Pivot-Kandidaten gewählt würde. Am Ende der Iteration erwartet man eine optimale und zulässige Basis für das LP, das aus der Anfangsbasis und der Kandidatenmenge besteht. Für $|C| = 1$ entspricht dies gerade den üblichen Gegebenheiten, während für $|C| = n$ das gesamte LP mit nur einem Block-Pivot gelöst wäre. Die gesuchte Basis erhält man, indem man das aus der aktuellen Basis und C bestehende LP löst, für das die aktuelle Basis als Startbasis verwendet werden kann. Zur Lösung dieses Sub-LPs schlägt Padberg das Simplex-Verfahren

in Tableau-Form vor, zu dessen Aufstellung lineare Gleichungssysteme für alle Vektoren aus C gelöst und in jeder Iteration aktualisiert werden müssen. Dies entspricht genau dem oben beschriebenen, das jedoch auch für entfernende Algorithmen erweitert wurde.

2.2.4 Paralleles Lösen verschiedener linearer Gleichungssysteme

In jeder Simplex-Iteration sind zwei lineare Gleichungssysteme zu lösen. Beim Block-Pivoting wird das jeweils erste von mehreren Iterationen zusammengefaßt und parallel gelöst. Dadurch ergibt sich eine Parallelisierung bei der Lösung des ersten Gleichungssystems. Der Rechte-Seite-Vektor des zweiten zu lösenden Gleichungssystems wird jedoch erst von dem Quotiententest festgelegt, so daß ein entsprechender Parallelisierungsansatz nicht durchführbar ist.

Für den wichtigen Spezialfall des steepest-edge Pricings muß jedoch noch ein drittes Gleichungssystem pro Iteration gelöst werden. Hier setzt nun eine funktionale Dekomposition an, bei der das zweite und dritte Gleichungssystem nebenläufig gelöst werden. Im verteilten Fall löst dazu je eine Hälfte der PEs das zweite und die andere Hälfte das dritte Gleichungssystem, und anschließend werden die Ergebnisvektoren paarweise ausgetauscht. Dagegen lösen im Fall von gemeinsamem Speicher nur zwei PEs nebenläufig die beiden Gleichungssysteme, während alle anderen PEs warten, bis nach Lösung der Gleichungssysteme eine Barrier-Synchronisation erfolgt.

Wie bei einer funktionalen Dekomposition von Algorithmen üblich, führt dieser Ansatz zu keinem skalierbaren Programm; es können lediglich 2 Prozessoren mit sinnvollen Aufgaben beschäftigt werden. Allerdings findet man heute immer öfter Workstations mit gerade 2 Prozessoren, wofür dieser Ansatz vom Prinzip her besonders gut geeignet ist.

2.2.5 Zusammenfassung

Abbildung 2.3 zeigt die Wirkung der in den vorigen Abschnitten dargestellten Parallelisierungsansätze mit Hilfe von (nachempfundenen) Laufzeitdiagrammen für den entfernenden Simplex-Algorithmus. Für den parallelen Algorithmus werden zwei volle Simplex-Iterationen dargestellt, während der entsprechende sequentielle Algorithmus in derselben Zeit lediglich 1,5 Iterationen bewerkstelligt. Die Teilaufgaben sind mit den Buchstaben a bis e bezeichnet, wobei der Index die Iterationen numeriert. Die Diagramme zeigen zwar den verteilten Fall, jedoch ergeben sich auch für Parallelrechner mit gemeinsamem Speicher kaum Unterschiede. An die Stelle des Gossiping tritt dann jeweils eine Synchronisation mit entsprechendem Aufwand.

Betrachten wir zunächst den sequentiellen Algorithmus. Er beginnt mit dem Pricing-Schritt (a_1), innerhalb dessen das lineare Gleichungssystem aus Schritt 2 von Algorithmus 6 gelöst wird. Dies wird zum Pricing dazugerechnet, um später den Effekt des Block-Pivoting im parallelen Fall deutlicher zu machen. Es folgt die Berechnung des Matrix-

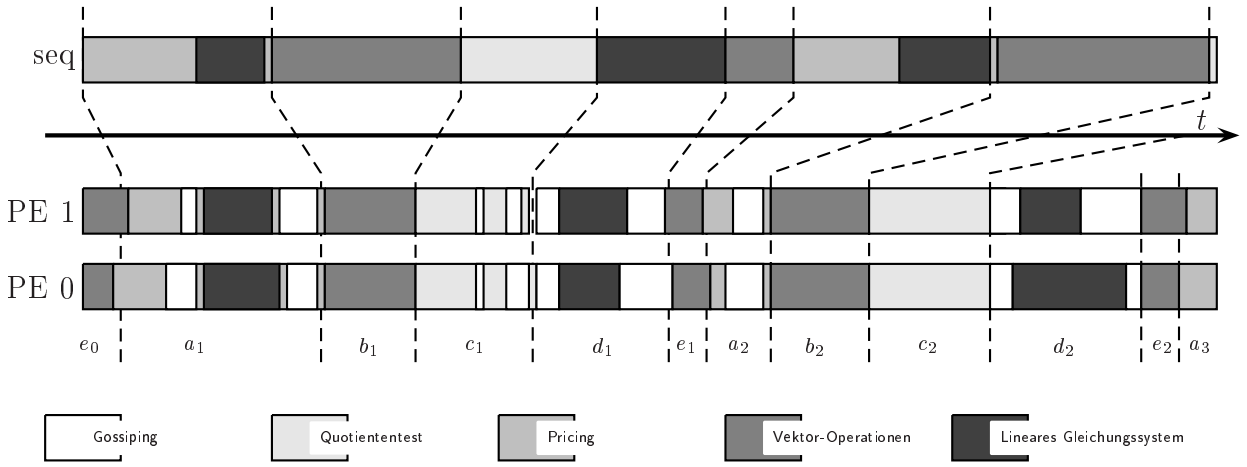


Abbildung 2.3: Laufzeitdiagramme für den entfernenden Simplex-Algorithmus mit steepest-edge Pricing für ein bzw. zwei PEs.

Vektor-Produktes $\Delta g^T = \Delta h^T A$ (b_1) und der Quotiententest (c_1). Anschließend werden die beiden linearen Gleichungssysteme für Δf und für die Aktualisierung der Normen ρ_i gelöst (d_1). Schließlich werden alle relevanten Vektoren aktualisiert (e_1) und die nächste Iteration beginnt (a_2).

Wir betrachten nun den parallelen Fall. Die Index-Auswahl im Pricing-Schritt (a_1) beansprucht etwas weniger Zeit als im sequentiellen Fall, da jedes PE nur einen Teil der Indexmenge durchsucht. Jedoch muß am Schluß mit einem Gossip der tatsächlich zu benutzende Index ausgewählt werden. Beide PEs lösen nun mit ihrem Kandidaten ein linears Gleichungssystem, wobei anschließend der Lösungsvektor zum Pivot-Index dem anderen PE übermittelt wird. Dies geschieht mit dem Gossip am Ende des Pricing-Schrittes. Man erkennt, daß der Kommunikationsaufwand den Geschwindigkeitsgewinn durch die Parallelisierung der Index-Auswahl wieder zunichte machen kann. Die genauen Verhältnisse sind problem- und architekturabhängig.

Nun kann jedes PE seinen Teil des Matrix-Vektor-Produktes berechnen (b_1). Hierfür ist keinerlei Kommunikation erforderlich, so daß die Laufzeit bei einem gutem Lastausgleich gegenüber dem sequentiellen Algorithmus halbiert werden kann.

Beim Quotiententest ergibt sich ein ähnliches Bild wie beim Pricing. Zwar wird die Zeit für die Index-Auswahl auf beide PEs verteilt, jedoch kann die notwendige Kommunikation zur Bestimmung des wirklich in die Basis eintretenden Vektors den Zeitgewinn wieder relativieren. Dabei sind zwei Gossips nötig, je einer für jede Phase des stabilen SoPlex-Quotiententests.

Im Anschluß an den Quotiententest werden die beiden weiteren linearen Gleichungssy-

steme gelöst (c_1). Zuvor wird jedoch mit einem weiteren Gossip der in die Basis eintretende Vektor allen PEs zugeschickt. Nun löst je eines PE eines der beiden Gleichungssysteme und schickt den Lösungsvektor dem jeweils andern PE. Bei mehr als zwei PEs werden die PEs dazu zu Paaren gruppiert. Auch in diesem Schritt ist für die Nutzung der Parallelität ein zusätzlicher Kommunikationsaufwand nötig.

Nach erfolgreicher Lösung der Gleichungssysteme können alle involvierten Vektoren aktualisiert werden (e_1). Diese Operation benötigt keine weitere Kommunikation oder Synchronisation, so daß wieder eine gute Effizienz erwartet werden kann.

In dem Pricing-Schritt der folgenden Iteration wirkt sich das Block-Pivoting aus. Kein PE muß nun noch einen Index auswählen, da jedes PE ja bereits in der vorigen Iteration einen Kandidaten bestimmt hatte. Einer davon wurde für den vorigen Pivot-Schritt benutzt. Sind die anderen (im Fall von 2 PEs der andere) immernoch wählbar, und dieser Fall wird in Abbildung 2.3 dargestellt, so muß der in der vorigen Iteration bestimmte Lösungsvektor lediglich aktualisiert und zu den anderen PEs übertragen werden. So wird der Pricing-Schritt über mehrere Iterationen hinweg beschleunigt.

Die folgenden Schritte unterscheiden sich nicht weiter von der vorigen Iteration. Lediglich bei Schritt (c_2) wurde angedeutet, daß häufig beide Gleichungssysteme unterschiedlich viel Zeit für die Lösung beanspruchen, was die Effizienz weiter reduziert. Insgesamt bleibt festzustellen, daß der Kommunikations- (oder Synchronisations-) Aufwand sowie Lastausgleichsprobleme die Effizienz des parallelen Algorithmus beeinträchtigen können. Je größer die zu lösenden Problem, desto gröber wird die Granularität, und damit die erreichbare Effizienz.

2.3 Parallele Lösung dünnbesetzter linearer Gleichungssysteme

Die im vorigen Abschnitt vorgestellten Ansätze zur Beschleunigung der Lösung von Gleichungssystemen im Simplex-Algorithmus befriedigen nicht vollständig, da sie zu keinem skalierbaren Algorithmus führen. Deshalb wurde auch ein datenparalleler Löser für dünnbesetzte Gleichungsssyteme entwickelt.

Um es gleich vorweg zu sagen: Wegen der besonders dünnbesetzten Struktur von Basis-Matrizen und Rechte-Seite-Vektoren, die bei derzeit behandelten LPs auftreten, konnte damit keine weitere Beschleunigung des Simplex-Verfahrens erzielt werden. Aufgrund der zentralen Bedeutung der Lösung dünnbesetzter linearer Gleichungssysteme weit über den Rahmen der Linearen Programmierung hinaus, wird in den folgenden Abschnitten dennoch der parallele LU-Zerlegungs-Algorithmus für dünnbesetzte Matrizen und der darauf basierende verteilte Löser vorgestellt. Für Matrizen aus anderen Bereichen arbeitet er performanter als eine vergleichbare Implementierung (vgl. Abschnitt 4.5).

Die Implementierung wurde für den T3D von Cray Research Inc. vorgenommen,

ein Parallelrechner mit verteiltem Speicher. Da die Synchronisationszeiten derzeitiger Multiprozessor-Workstations die Kommunikationszeit des Cray T3D übertreffen, wurde auf eine Implementierung für Architekturen mit gemeinsamen Speicher verzichtet. Im folgenden wird deshalb direkt der Algorithmus für Architekturen mit verteiltem Speicher vorgestellt.

2.3.1 Parallele LU-Zerlegung

Das Verfahren der LU-Zerlegung wurde bereits mit Algorithmus 7 beschrieben. Dabei liegt der hauptsächliche Rechenaufwand im Update-Loop, aber auch die Pivot-Auswahl und der L-Loop benötigen eine gewisse Zeit. Die nutzbare Parallelität in diesen drei Schritten hängt von der Verteilung der Daten auf die PEs ab. Sie wird in Abschnitt 2.3.1.1 vorgestellt. Ähnlich dem Block-Pivoting beim Simplex-Algorithmus können auch bei der LU-Zerlegung dünnbesetzter Matrizen bei gewissen Gegebenheiten mehrere Iterationen zu einer „Block-Iteration“ zusammengefaßt werden, die zusätzliche Nebenläufigkeit bietet. Die dazu nötige *Kompatibilität* von Pivot-Elementen und deren Ausnutzung wird in Abschnitt 2.3.1.2 erläutert. Schließlich wird in Abschnitt 2.3.1.4 ein neues Verfahren zum dynamischen Lastausgleich vorgestellt, daß ohne zusätzliche Kommunikation oder Synchronisation auskommt.

2.3.1.1 Datenverteilung

Die Verteilung der Daten auf die PEs bestimmt die nutzbare Nebenläufigkeit und sollte so erfolgen, daß alle PEs in etwa dieselbe Last erhalten. Es gibt zwei gundsätzliche Ansätze für die Datenverteilung. Bei einem wird die Beseztungsstruktur der Matrix analysiert und auf dieser Grundlage eine günstige Verteilung ermittelt, die ein Maximum an Parallelität und in Minimum an entfernten Datenzugriffen gewährleistet [61]. Dieser Ansatz ähnelt den Verfahren der Fillminimierung bei der LU-Zerlegung, bei denen die Permutationsmatrizen vor der Faktorisierung ermittelt werden. Deshalb eignet er sich auch besonders für solche Verfahren.

Beim zweiten Ansatz wird eine feste Verteilung der Daten auf die PEs zugrundegelegt [49, 89]. Dies erfordert ggfs. einen dynamischen Lastausgleich. Da beides ohne zusätzlichen Rechen- und Kommunikations- bzw. Synchronisationsaufwand erfolgen (vgl. Abschnitt 2.3.1.4) kann, wurde dieser Ansatz verfolgt. In [85] wurden verschiedene (feste) Verteilungsschemata von Zeilen und Spalten auf PEs untersucht, wobei eine gitterartige Verteilung zu einem besonders geringen Kommunikationsaufwand führt. Deshalb wurde sie wie in [89] auch für die hier vorgestellte Implementierung verwendet.

Sei $p = r \cdot c$ die Anzahl der verfügbaren PEs. Die PEs werden logisch als ein $r \times c$ Gitter angeordnet (vgl. auch Abb. 3.16), und deshalb mit (k, l) für $0 \leq k < r$ und $0 \leq l < c$ bezeichnet. Jedem Matrixelement B_{ij} wird einem PE zugeordnet, das es speichert, sofern

es von Null verschieden ist. Die Zuordnung von Matrixelementen zu PEs erfolgt gemäß

$$pe(i,j) = (i \mod r, j \mod c) \tag{2.8}$$

d.h. das Matrixelement B_{ij} wird auf dem PE $(i \mod r, j \mod c)$ gespeichert. Jedes PE verwaltet also eine (dünnbesetzte) Submatrix, wobei dieselben Datenstrukturen wie bei der sequentiellen Implementierung verwendet werden. Die Verteilung (2.8) erschließt Parallelität beim Update-Loop, beim L-Loop, bei der Pivot-Auswahl und bei der Berechnung der maximalen Absolutbeträge pro Zeile.

In [101] wurde eine Implementierung vorgestellt, bei der lediglich die Zeilen auf die PEs verteilt wurden, jedes PE aber alle Spalten verwaltet. Diese Verteilung benötigt weniger Kommunikationen als (2.8) da u.a. die Maxima in jeder Zeile von jedem PE ohne Kommunikation mit anderen bestimmt werden können. Deshalb eignet sich diese Implementierung besonders für Parallelrechner, die eine hohe Latenzzeit bei der Kommunikation aufweisen. Die Reduzierung der Anzahl von Kommunikationen erfolgt jedoch auf Kosten der nutzbaren Nebenläufigkeit, was die Skalierbarkeit dieser Implementierung einschränkt.

2.3.1.2 Kompatible Pivot-Elemente

Bei der LU-Zerlegung dichtbesetzter Matrizen ist die nutzbare Parallelität vollständig durch die Verteilung vorgegeben. Es ist nicht möglich, mehrer Pivot-Schritte nebenläufig zu bearbeiten, da sich die aktive Submatrix mit jedem Schritt ändert.

Für dünnbesetzte Matrizen gilt diese Aussage nicht mehr. Sofern *kompatible* Pivot-Elemente benutzt werden, stören sich die Änderungen der aktiven Submatrix nicht gegenseitig, so daß solche Pivot-Elemente parallel eliminiert werden können. Ähnlich dem Block-Pivoting beim Simplex-Algorithmus werden so mehrer Iterationen zu einer parallelen Iteration zusammengefaßt.

Definition 12

Die Elemente $B_{i_1 j_1}, \ldots, B_{i_c j_c}$ einer Matrix $B \in \mathbb{R}^{n \times n}$ mit $c \leq n$ heißen kompatibel, *falls für $1 \leq k \neq l \leq c$ gilt*

$$B_{i_l j_k} = 0. \tag{2.9}$$

Die Submatrix aus den Zeilen und Spalten von kompatiblen Elementen bilden also eine Diagonalmatrix. Die Bedeutung von kompatiblen Elementen für die parallele LU-Zerlegung dünnbesetzter Matrizen wird von dem folgenden Satz beschrieben.

Satz 35

Seien $B_{i_1 j_1} \neq 0$ und $B_{i_2 j_2} \neq 0$ kompatible Elemente einer $n \times n$ Matrix $B = B^1$. Nach Elimination beider Elemente gilt

$$B^3 = B - \frac{B_{.j_1} B_{i_1 .}}{B_{i_1 j_1}} - \frac{B_{.j_2} B_{i_2 .}}{B_{i_2 j_2}} \tag{2.10}$$

unabhängig von der Eliminationsreihenfolge von $B_{i_1 j_1}$ und $B_{i_2 j_2}$.

Beweis:
Wir zeigen (2.10) für die Eliminationsreihenfolge $B_{i_2 j_2}$, $B_{i_1 j_1}$. Nach Schritt 3 und 5 von Algorithmus 7 gilt:

$$\begin{aligned} B^3 &= B^2 - \frac{B^2_{\cdot j_1} B^2_{i_1 \cdot}}{B^2_{i_1 j_1}} \\ &= \left(B^1 - \frac{B^1_{\cdot j_2} B^1_{i_2 \cdot}}{B^1_{i_2 j_2}}\right) - \frac{\left(B^1_{\cdot j_1} - \frac{B^1_{\cdot j_2} B^1_{i_2 j_2}}{B^1_{i_2 j_2}}\right)\left(B^1_{i_1 \cdot} - \frac{B^1_{i_1 j_2} B^1_{i_2 \cdot}}{B^1_{i_2 j_2}}\right)}{B^1_{i_1 j_1} - \frac{B^1_{i_1 j_2} B^1_{i_2 j_1}}{b^1_{i_2 j_2}}} \\ &= B - \frac{B_{\cdot j_2} B_{i_2 \cdot}}{B_{i_2 j_2}} - \frac{B_{\cdot j_1} B_{i_1 \cdot}}{B_{i_1 j_1}}, \end{aligned}$$

wobei die letzte Gleichung aus der Kompatibilitätsbeziehung $B_{i_1 j_2} = 0 = B_{i_2 j_1}$ folgt. Gleichung (2.10) zeigt man analog für die umgekehrte Pivotreihenfolge, wobei die Indizes 1 und 2 jeweils vertauscht werden müssen.

■

Nach Satz 35 können kompatible Elemente in einer beliebigen Reihenfolge eliminiert werden, ohne daß dies zu einer Änderung der aktiven Submatrix führt. Insbesondere können verschiedene PEs eine unterschiedliche Reihenfolgen bei der Elimination wählen. Dies kann man sich für einen parallelen LU-Zerlegungs-Algorithmus zunutze machen, der weniger stark synchronisiert und somit von gröberer Granularität ist.

Das Konzept der Kompatibilität wurde zunächst in [33] für die Implementierung eines parallelen LU-Zerlegungs-Algorithmus auf Multiprozessoren mit gemeinsamen Speicher eingeführt. In [89] wurde es für Transputer-Systeme eingesetzt. Für symmetrische Matrizen hat die Kompatibilität eine einfache Darstellung als sog. *elimination Trees*, die bereits früher die Grundlage bei der parallelen Choleski-Zerlegung symmetrischer dünnbesetzter Matrizen bildete [41, 61]. Sie wurden durch Übergang zur symmetrisierten Matrix $B + B^T$ oder $B^T B$ auch auf den unsymmetrischen Fall erweitert [38]. In [33] wird jedoch gezeigt, daß diese Matrizen meist dichter besetzt sind als B, so daß mehr Arbeit bei der LU-Zerlegung anfällt und die durch Dünnbesetztheit gebotene Parallelität reduziert wird. Aus diesem Grund wird hier ähnlich zu [89] die Kompatibilität direkt auf der unsymmetrischen Besetzungsstruktur der zu faktorisierenden Matrix ausgenutzt.

ALGORITHMUS 12 (PARALLELE LU-ZERLEGUNG)

Setzte $s \leftarrow 0$.
Solange $s < n$:

Schritt 1 (Pivot-Auswahl):
Wähle eine Menge von e kompatiblen Pivot-Elementen.

Schritt 2: Berechne L-Loop für lokale Teile der Pivot-Spalten.

Schritt 3: Schicke lokalen Teil der L-Vektoren den anderen PEs derselben Prozessorzeile, und
schicke lokalen Teil der Pivot-Zeilen den anderen PEs derselben Prozessorspalte.

Schritt 4 (a): Für alle lokalen Pivot-Elemente:
Berechne lokalen Update-Loop

Schritt 4 (b): Für alle weiteren Pivot-Elemente:
Berechne lokalen Update-Loop

Schritt 5: Zeilen-Gossip:
Aktualisiere die Werte `maxabs` und die Anzahl der NNEs pro Zeile.

Schritt 6: Spalten-Gossip: Aktualisiere Anzahlen der NNEs pro Spalte.

Schritt 7: $s \leftarrow s + e$

Der Update-Loop für die kompatiblen Pivot-Elemente wurde in zwei Teile zerlegt: In Schritt 4.a werden die lokalen Pivot-Elemente eliminiert und in Schritt 4.b die von den anderen PE erhaltenen. Dadurch kann Schritt 4.a gleichzeitig mit Schritt 3 erfolgen. Gängige Parallelrechner mit verteiltem Speicher verfügen über geeignete Hardware, die Kommunikation ohne Prozessorbeteiligung abwickeln kann. Der Prozessor initiiert lediglich die Kommunikation, die anschließend asynchron zur weiteren Ausführung des Programms erfolgt. Damit gelingt es in Grenzen, den Kommunikationsaufwand zu verbergen.

In den Schritten 5 und 6 wird jeweils ein Gossiping-Algorithmus auf Teilmengen der Prozessoren ausgeführt, nämlich jeweils die Menge der Prozessoren einer Zeile bzw. Spalte des $r \times c$ Prozessorgitters. Sie sind nötig, damit jede PE für die Pivot-Auswahl der folgenden Iteration die korrekte Information der globalen Matrix hat. Die Anzahl der NNEs pro Zeile und Spalte wird für die Berechnung der Markowitzzahlen benötigt. Da Fill-Elemente auf andern PEs entstehen können, müssen diese Zahlen nach jeder paralleln Iteration aktualisiert werden. Um das Kommunikationsvolumen gering zu halten, werden nur die Änderungen kommuniziert.

Auch die Werte des betragsgrösten NNEs pro Zeile in `maxabs` müssen aktualisiert werden. Im Gegensatz zum sequentiellen Algorithmus können diese Werte nicht erst bei Bedarf berechnet werden, da zu ihrer Bestimmung jeweils alle PEs einer Prozessorzeile kooperieren müssen. Wenn ein PE einen neuen Wert benötig, wäre ein aufwendiges Protokoll nötig,

damit die anderen involvierten PEs davon erfahren und bei der Bestimmung mitarbeiten. Statt dessen werden die Maxima zu Anfang für alle Zeilen bestimmt und nach jeder parallelen Iteration aktualisiert. Dabei werden wieder nur die Änderungen kommuniziert.

Algorithmus 12 benötigt eine Synchronisation aller PEs lediglich am Anfang einer parallelen Iteration bei der Auswahl kompatibler Pivot-Elemente (s.u.) und am Ende, wenn die Zeilen- und Spalteninformation aktualisiert wird. Also erfolgt nur alle e Eliminationsschritte eine Synchronisation und damit wesentlich seltener, als wenn ohne Nutzung von Kompatibilität nach jedem Eliminationsschritt synchronisiert werden müßte. Da bei jeder Synchronisation alle PEs auf den letzten warten müssen, erwartet man durch die selteneren Synchronisationsoperationen insgesamt kürzere Wartezeiten.

2.3.1.3 Auswahl kompatibler Pivot-Elemente

Zu den beiden Zielen bei der Pivot-Auswahl für die sequentielle LU-Zerlegung, nämlich der numerischen Stabilität und der Fill-Minimierung, muß für den parallelen Fall ein weiteres Kriterium hinzugezogen werden, nämlich das Auffinden einer möglichst großen Menge kompatibler Pivot-Elemente. Glücklicherweise ist letzteres gut vereinbar mit der Fill-Minimierung, denn Pivot-Elemente mit niedriger Markowitz-Zahl haben wenige NNEs in derselben Zeile oder Spalte und sind somit eher kompatibel zu anderen Elementen als solche mit einer höheren Markowitz-Zahl. Aus diesem Grund kann die Auswahl der Pivot-Elemente weitgehend von der sequentiellen Version übernommen werden.

Der Algorithmus 13 für die Auswahl kompatibler Pivot-Elemente erfolgt in drei Phasen. In der ersten Phase wählt jedes PE aus seinen lokalen NNEs eine Menge von Pivot-Kandidaten aus, die zu einer globalen Kandidatenmenge zusammengefaßt wird. Zu jedem Paar von Pivot-Kandidaten wird in der zweiten Phase die Kompatibilität überprüft und gespeichert. Schließlich wird in der letzten Phase greedy-artig eine Teilmenge kompatibler Pivot-Elemente aus der Kandidatenmenge extrahiert.

Algorithmus 13 (Bestimmung kompatibler Pivot-Elemente)

Phase 1 (Kandidaten-Auswahl):
Bilde lokale sortierte Liste L von Pivot-Kandidaten

Gossip : Bilde globale sortierte Liste G von Pivot-Kandidaten

Phase 2 (Inkompatibilitätsbestimmung):
Bilde lokale Liste von Paaren inkompatibler Pivot-Kandidaten

Gossip : Bilde globale Liste von Paaren inkompatibler Pivot-Kandidaten

Phase 3 (Kompatiblen-Auswahl):
Setze $C = \emptyset$.
Solange $|G|$ nicht leer:

Schritt 1: Sei G_1 das erste Element in G.
Setze $C \leftarrow C \cup \{G_1\}$

Schritt 2: Entferne G_1 und alle zu G_1 inkompatiblen Elemente aus G

C ist die Menge kompatibler Pivot-Elemente

In der ersten Phase wählt zunächst jedes PE eine lokale Menge von Pivot-Kandidaten aus, wobei jeder Kandidat nach derselben Strategie bestimmt wird wie die Pivot-Elemente im sequentiellen Fall. Diese Kandidaten werden lokal nach aufsteigender Markowitz-Zahl sortiert. Anschließend wird mit einem Gossiping-Algorithmus aus den lokalen sortierten Kandidatenlisten eine globale sortierte Liste erzeugt. Die Gossiping-Operation ist dabei die Vereinigung zweier sortierter Listen in eine sortierte Liste und arbeitet in linearer Zeit.

In der zweiten Phase bestimmt jedes PE Paare inkompatibler Pivot-Elemente, indem es lokal die relevanten Matrix-Elemente überprüft. Typischerweise treten aufgrund der niedrigen Markowitz-Zahl der Kandidaten nur wenige Inkompatibilitäten auf. Mit einem weiteren Gossiping-Algorithmus wird die Menge alle inkompatibler Pivot-Kandidatenpaare aufgestellt.

In der dritten Phase wird schließlich die Menge der kompatiblen Pivot-Elemente aus der sortierten Kandidatenliste mit einem Greedy-Algorithmus extrahiert. Das erste Element der Kandidatenliste wird herausgenommen und der Menge der Pivot-Elemente zugefügt. Anschließend werden alle zu diesem Element inkompatiblen Kandidaten aus der Liste entfernt. Mit dem nun ersten Element in der Kandidatenliste wird ebenso verfahren. Dies wird iteriert, bis die Kandidatenliste leer ist.

Die Bestimmung einer maximalen Teilmenge kompatibler Pivot-Elemente ist ein Stabile-Mengen-Problem, zu dessen Lösung aufwendigere Algorithmen existieren, die bessere Lösungen versprechen. Sie werden jedoch aus zwei Gründen nicht benutzt. Zum einen wurde die Kandidatenmenge selbst ohne weitere Berücksichtigung von Kompatiblität aufgestellt, so daß eine etwaige „Verbesserung“ bereits dort ansetzen müßte. Zum anderen bevorzugt der Greedy-Algorithmen Kandidaten am Anfang der Liste, also solche mit geringer Markowitz-Zahl. Diese sind aber auch wegen ihrer Fill-Minimierungseigenschaft vorzuziehen.

Es gibt zwei Spezialfälle, für den sich eine spezielle Implementierung lohnt, nämlich für Zeilen oder Spalten mit nur einem NNE. Solche werden *Singletons* genannt. Offenbar führt jede Menge von Zeilensingletons mit paarweise verschiedenen Zeilenindizes zu kompatiblen Pivot-Elementen; gleiches gilt für Spaltensingletons. Für solche Mengen bedarf es also keiner weiteren Überprüfung der Kompatibilität. Ferner entfällt der Update-Loop und für Spaltensingletons auch der L-Loop.

Singletons treten bei den Basis-Matrizen im Simplex-Algorithmus besonders häufig auf, da diese meist auch Schlupfvariablen (bzw. Variablenschranken bei einer Zeilenbasis) enthalten. Aus diesem Grund werden Algorithmus 12 zwei Phasen vorgeschaltet. In der ersten werden alle Zeilen-Singletons und in der zweiten alle Spalten-Singletons eliminiert. Die

verbleibende aktive Submatrix heißt der *Nukleus* der Matrix und enthält keine weiteren Singletons. Im Nukleus können zwar nach einigen Pivot-Eliminationen neue Singletons entstehen, jedoch geschieht dies i.a. in so niedriger Zahl, daß eine separate Behandlung nicht mehr lohnt.

2.3.1.4 Lazy Loadbalancing

Da die zu verrichtende Arbeit mit Abnahme der Dimension der aktiven Submatrix dynamisch von Iteration zu Iteration variiert, und zwar wohl möglich unterschiedlich auf verschiedenen PEs, muß ein dynamischer Lastausgleich erfolgen. Dazu bedarf es eines Maßes für die Last eines PEs im Vergleich zur Gesamtlast.

Eine Möglichkeit für solch ein Maß wäre das Verhältnis der Anzahl der NNEs der aktiven Submatrix eines PEs zur Gesamtzahl der NNEs. Letztere zu bestimmen würde jedoch einen weiteren Kommunikationsaufwand erfordern, der vermieden werden sollte. Statt dessen wird ein approximatives Maß verwendet, die von einer gleichmäßigen Verteilung der NNEs auf die Submatrix ausgeht. Dies ist durch die Verteilung (2.8) relativ gut gewährleistet. Für jedes PE (i,j) bezeichne $r^s(i,j)$ die Anzahl der Zeilen und $c^s(i,j)$ die Anzahl der Spalten von der s-ten aktiven Submatrix, die von dem PE verwaltet werden. Dann wird die Last des i-ten PEs als

$$l^s(i,j) = c^s(i,j) \cdot r^s(i,j) \tag{2.11}$$

definiert. Wegen $\sum r^s(i,j) = n - s = \sum c^s(i,j)$ ist die Gesamtlast

$$L^s = (n-s) \cdot (n-s). \tag{2.12}$$

Die Aufgabe des Lastausgleichs ist es, $l^s(i) = L^S/p$ möglichst gut zu erreichen. Aktive Verfahren spüren Abweichungen davon auf und transferieren Last von überlasteten PEs zu unterforderten. Bei der LU-Zerlegung würde dies durch die Übergabe von Zeilen oder Spalten erfolgen, was zu einem zusätzlichen Kommunikationsaufwand führte.

Dieser kann mit einem passiven Lastausgleichsverfahren vermieden werden, der in [101] zuerst vorgestellt wurde. Da in jeder parallelen Iteration Last abgebaut wird, kann die Lastverteilung ausgeglichen werden, indem PEs mit einer höheren Last versuchen in der folgenden Iteration mehr Last abzubauen als solche mit einer geringern Last. Dies wird über die Anzahl der Pivot-Kandidaten $k^s(i,j)$ erreicht, die jedes PE (i,j) in der s-ten Iteration auswählt:

$$k^s(i,j) = K \cdot \left\lceil \frac{l^s(i,j)}{L^s} \right\rceil . \tag{2.13}$$

Damit werden insgesamt ca. K Kandidaten ausgewählt, wozu jedes PE anteilig zu seiner Last viele beiträgt. Abbildung 2.4 zeigt, daß damit in der Tat eine bessere Lastverteilung erreicht wird.

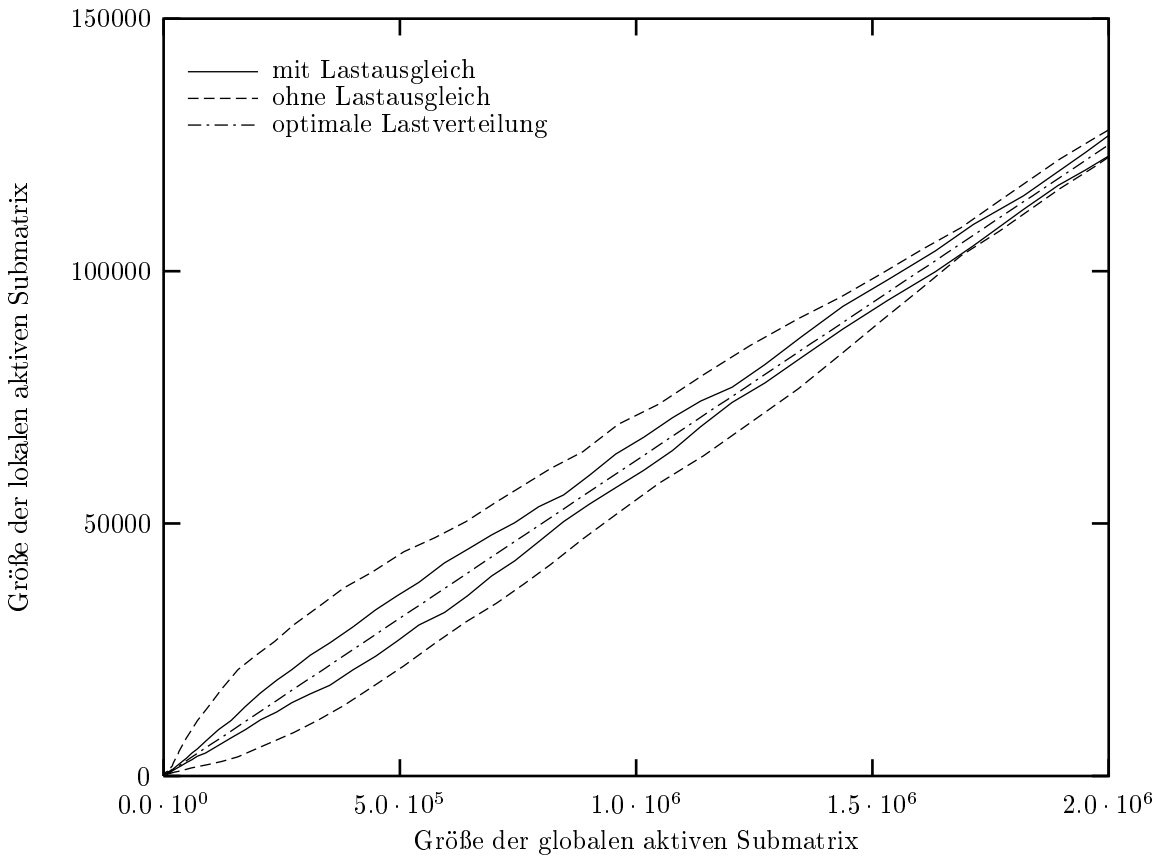


Abbildung 2.4: Effekt des lazy load-balancing. Es ist die maximale und minimale Last $\max\{l^s(i,j)\}$, bzw. $\min\{l^s(i,j)\}$ über der Gesamtlast aufgetragen. Durch das lazy load-balancing wird die Differenz beider Werte kleiner gehalten als ohne.

2.3.2 Parallele Vor- und Rückwärtssubstitution

Bei der Behandlung der parallelen Lösung gestaffelter linearer Gleichungssysteme beschränken wir uns exemplarisch auf den Fall

$$Ly = b, \tag{1.116}$$

wobei L eine untere Dreiecksmatrix mit Diagonalen 1 sei. Die Lösung von Gleichungssystemen mit einer oberen Dreiecksmatrix gestaltet sich analog. Der Lösungsalgorithmus zu (1.116) wurde bereits in Abschnitt 1.7.4 vorgestellt und sei hier wiederholt:

Für $i = 1, \ldots, n$:
    Für $j = i + 1, \ldots, n$:
        Setze $y_j \leftarrow y_j + (-y_i \cdot L_{ji})$

Die Berechnungen in der Schleife über j sind für verschiedene Indizes vollkommen unabhängig. Deshalb kann diese Schleife parallelisiert werden, indem jedes PE eine Teilmenge der Indizes bearbeitet. Die Schleife über i bleibt hingegen inherent sequentiell.

Wie schon bei der LU-Zerlegung bietet die dünnbesetzte Struktur von L weitere Nebenläufigkeit. L hat nämlich nach der parallelen LU-Zerlegung eine besondere Struktur, bei der sich entlang der Diagonalen jeweils diagonale Submatrizen befinden. Bezeichne mit $B_1, B_2, \ldots, B_k$ die zugehörigen Indexmengen, d.h. die Matrizen $L_{B_iB_i}$ sind für $i = 1, \ldots, k$ Einheitsmatrizen. Dabei gelte $b_i < b_j$ für alle $b_i \in B_i$ und $b_j \in B_j$ mit $i < j$ (vgl. Abb. 2.5).

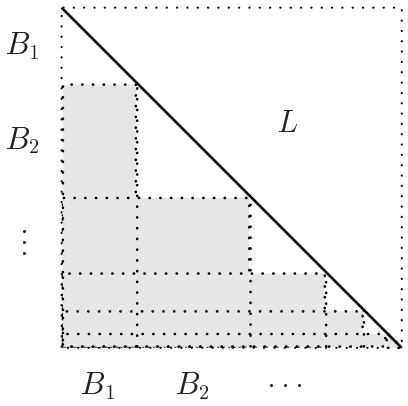


Abbildung 2.5: Struktur der Dreiecksmatrix L nach paralleler LU-Zerlegung mit kompatiblen Pivot-Elementen.

Wegen der Struktur von L kann für $i \in B_l$ die Schleife über j auf $j \in B_{l+1} \cup \cdots \cup B_k$ beschränkt werden. Insbesondere fällt keine Arbeit für $j \in B_l$ an. Da dies für alle $i \in B_l$ zutrifft und die Operation in der Schleife über j assoziativ und kommutativ ist, bietet sich eine zusätzliche Nebenläufigkeit innerhalb der Schleife über i.

ALGORITHMUS 14

Für $l = 1, \ldots, k$:
  Für $j \in B_{l+1} \cup \cdots \cup B_k$:
    Für $i \in B_l$:
      Setze $y_j \leftarrow y_j - y_i \cdot L_{ji}$

In Algorithmus 14 kann wieder die Schleife über j parallel ausgeführt werden. Dies gilt nicht für die Schleife über i, da die Schreib- und Lesezugriffe auf y_j Synchronisiert werden müssen. Es ist jedoch (bis auf numerische Fehler) unerheblich, in welcher Reihenfolge die Elemente von B_l traversiert werden.

Algorithmus 14 impliziert eine zeilenweise Verteilung der Matrix L. Dabei legt die Schleife über i eine zeilenweise Speicherung von L nahe. Wie in Abschnitt 1.7.4 ist dies jedoch für dünnbesetzte Vektoren b ungünstig, da bei der Schleife über i häufig unnötige Operationen mit $y_i = 0$ ausgeführt werden. Aus diesem Grund wurde eine spaltenweise Speicherung beibehalten und L gleichzeitig zeilenweise auf die PEs verteilt: Sind $p = k$ PEs vorhanden, so verwaltet PE i die Submatrix $L_{B_i,}$ mit in einer spaltenweisen Speicherung. Falls nur $p < k$ PEs verfügbar sind, erhält das PE i alle Submatrizen L'_j, mit $j \mod p = i$. Dadurch ergibt sich bei $p \ll k$ gleichzeitig eine Lastverteilung.

PE q führt folgenden Algorithmus aus:

ALGORITHMUS 15 (PARALLELE VORWÄRTSSUBSTITUTION)

Schritt 1 (Bearbeitung):
Für $l = 1, \ldots, q$:
  Warte auf PE l
  Für $i \in B_l$:
    Für $j \in B_q$:
      Setze $y_j \leftarrow y_j - y_i \cdot L_{ji}$

Schritt 2 (Synchronisation):
Signalisiere Termination

Dabei wartet jedes PE in Schritt 1 darauf, daß PE l Schritt 2 erreicht. Diese Synchronisation erzwingt die sequentielle Bearbeitung der globalen Schleife über l in Algorithmus 14. Die Schleife über j wird wieder nur dann ausgeführt, wenn $y_i \neq 0$ gilt. Dadurch kann die Dünnbesetztheit von b ausgenutzt werden.

Bei einem Parallelrechner mit verteiltem Speicher erfolgt die Synchronisation bei der Übertragung des Teil-Vektors y_{B_l}, der für die Berechnungen in Schritt 1 erforderlich ist. Für eine Shared-Memory-Architektur würde hingegen eine explizite Synchronisation benutzt.

Kapitel 3

Implementierung in C++

Dieses Kapitel beschreibt den objektorientierten Software-Entwurf für die drei Implementierungen SoPlex, SMoPlex und DoPlex. Als solcher gliedert er sich in eine Menge von Klassen. Die Klassen für die sequentielle Implementierung werden in Abschnitt 3.2 vorgestellt. In Abschnittt 3.3 werden die zusätzlichen Klassen beschrieben, die für die Parallelisierung für Parallelrechner mit gemeinsamen und verteiltem Speicher entwickelt wurden. Zuvor wird jedoch in Abschnitt 3.1 eine kurze Einführung in die Konzepte der objektorientierten Programmierung gegeben.

3.1 Grundlagen objektorientierter Programmierung

Seit in den 50er Jahren erste elektronische Datenverarbeitungsanlagen auf dem Markt erschienen sind, ist die Hardware einer rasanten Weiterentwicklung unterworfen. Mit dieser Entwicklung einher geht die Möglichkeit, immer schwierigere Probleme zu lösen. Allerdings bedarf es dazu auch immer komplexerer Software. Dabei traten die Grenzen der jeweils bis dato entwickelten Entwurfs- und Implementierungstechniken zum Vorschein, was wiederum Anlaß zu Fortentwicklungen auf den Gebieten des Softwar-Entwurfs, der Programmiersprachen und der formalen Spezifikation war. Diesen Entwicklungen gemein ist die Zielsetzung, folgende allgemeine Qualitätsanforderungen an Softare zu unterstützen.

1. Die Anforderung der *Korrektheit* verlangt, daß ein Programm seiner Spezifikation genügt. Sie ist natürlich die wichtigste Anforderung, die an Software zu stellen ist. Dazu wurden Methoden der formalen Verifikation entwickelt, mit denen die Korrektheit eines Programms gegenüber seiner Spezifikation formal nachgewiesen werden kann. Der Aufwand für solch einen Beweis ist jedoch so erheblich, daß diese Techniken in der Regel nicht zum Einsatz kommen. Vielmehr wird statt dessen durch umfangreiches und systematisches Testen die Korrektheit von Software überprüft, womit man jedoch nur in der Lage ist, das evtl. Vorhandensein von Fehlern nachzu-

weisen, nicht aber deren Abwesenheit. Mit dem Aufkommen von Nichtdeterminismus etwa bei parallelen Algorithmen werden die Grenzen des Testens immer deutlicher, so daß formalen Methoden eine zunehmende Bedeutung beigemessen wird.

2. *Erweiterbarkeit* beschreibt die Möglichkeit, die Software an (zukünftige) Änderungen und Erweiterungen der Spezifikation anzupassen.

3. Unter *Wiederverwendbarkeit* versteht man die Möglichkeit, die Software oder Teile davon bei der Programmierung anderer Anwendungen zu benutzen. Dabei ist *Kompatibilität* von zentraler Bedeutung.

4. *Effizienz* beschreibt die „gute“ Nutzung vorhandener Ressourcen. In wissenschaftlichen Applikationen wird der Begriff meist mit “Laufzeiteffizienz“, also der möglichst schnellen Berechnung des gewünschten Ergebnisses, gleichgesetzt. Für nicht wissenschaftlichen Anwendungen umfaßt der Effizienz jedoch auch den Einsatz anderer Ressourcen wie Speicherplatz, Ein- und Ausgabe oder Kommunikationsnetzwerke.

Darüberhinaus stellen sich in der Praxis oft weitere wichtige Anforderungen etwa hinsichtlich der Benutzerfreundlichkeit, der Fehlertoleranz u.v.m.

3.1.1 Programmierparadigmen

Die Entwicklungen auf den Gebieten der Software-Technik und Programmiersprachen ist durch verschiedene sog. Programmierparadigmen (oder -modelle) gekennzeichnet. Unter einem Programmierparadigma versteht man die bereitgestellten abstrakten Ausdrucksmittel zur Formulierung von Programmen. Auf grober Ebene unterscheidet man zwischen imperativer, funktionaler, logischer und objektorientierter Programmierung.

Im folgenden werden kurz imperative Programmierparadigmen sowie Grundzüge der objektorientierten Programmierung skizziert und auf ihre Leistungsfähigkeit in bezug auf die Anforderungen 1-4 verglichen. Dabei wird kein Anspruch auf Vollständigkeit erhoben sondern lediglich ein Überblick verschafft, der eine Einordung der Programmiersprache C++ ermöglicht, da die in dieser Arbeit beschriebene Software in C++ entworfen und implementiert wurde.

3.1.1.1 Imperative Programmierung

Die imperative Programmierung ist die älteste Art der Programmierung und wurde einer Reihe von Strukturierungen unterzogen. Zunächst bestand sie in der Beschreibung von Algorithmen als Folge *elementarer Operationen auf elementaren Daten*. Zur graphischen Darstellung solcher Programme eignen sich Flußdiagramme.

Die Anforderungen 1-4 wurden bei dieser Art der Programmierung kaum unterstützt. So kann etwa kein Programmteil frei in einem anderen Zusammenhang verwendet werden, da

dort Variablen möglicherweise anderweitig benutzt werden. Dies widerspricht Anforderung 3. Ebenso kann prinzipiell jeder Punkt in einem Programm von einem beliebigen anderem angesprungen werden. Dadurch ist eine Änderung stets mit kaum kalkulierbaren Risiken verbunden, was wiederum in Widerspruch zu Anforderung 2 steht [35].

Dieser Problematik wurde durch Bereitstellung weiterer Abstraktionsmittel auf Kontrollfluß- und Datenebene begegnet, die zur *strukturierten Programmierung* führte [36]. Sie hat den früheren imperativen Programmierstil vollständig abgelöst. Hierbei wird ein Programm als Folge strukturierter Operationen (Ausdrücke, Schleifen, Prozeduren, Blöcke) auf strukturierten Daten (lokale Namensräume, strukturierte Datentypen wie Verbunde und Reihungen) formuliert. Durch die Strukturierung des Namensraum der Variablen kann sichergestellt werden, daß Variablen nicht in verschiedenen Zusammenhängen inkompatibel benutzt werden, und durch strukturierte Operationen (Schleifen etc.) werden Sprünge mit den damit verbundenen Risiken vermieden. Zur Darstellung strukturierter Programme werden meist Nassi-Schneiderman Struktogramme verwendet [79]. Sie ermöglichen eine übersichtliche Darstellung strukturierter Operationen und sehen keine Möglichkeit zur Darstellung von Sprüngen vor.

Zunächst erscheint die Restriktion bei der Benutzung von Sprüngen oder beim Zugriff auf Daten als hinderlich. Es zeigt sich jedoch, daß z.B. Sprünge generell durch äquivalente strukturierte Ausdrucksmittel (wie Schleifen) ersetzt werden können. Dafür gewinnt man große Vorteile hinsichtlich der Anforderungen 1-4: (Seiteneffektfreie) Prozeduren etwa können in einem beliebigen anderen Programm eingesetzt werden (Anforderung 3), was zur Entwicklung umfangreicher Funktionsbibliotheken führte.

Dennoch unterstützt auch die strukturierte Programmierung nicht alle o.g. Anforderungen. Insbesondere zeigten sich Mängel bei der Wiederverwendbarkeit. Man denke etwa an einen Satz von Funktionen, die Operationen auf einer komplexen Datenstruktur, z.B. einer dünnbesetzten Matrix, ausführen. Oft wird diese Datenstruktur global zur Verfügung gestellt, damit alle Funktionen darauf zugreifen können. Dies kann aber zu Kompatibilitätsproblemen mit anderen Software-Komponenten führen, die evtl. dieselben globalen Daten anders verwenden, was zu Inkonsistenzen führen kann.

Diese Problematik hat zur Entwicklung der *modularen Programmierung* geführt. Sie stellt zusätzliche Ausdrucksmittel zur Trennung von Schnittstelle und der zugehörigen Implementierung zur Verfügung. Die Implementierung eines Moduls umfaßt alle benötigten Datenstrukturen und internen Funktionen zu deren Manipulation. Auf diese kann aber „von außerhalb“ des Moduls nicht direkt zugegriffen werden. Vielmehr kann nur die Schnittstelle benutzt werden. Die Modulschnittstelle ist eine Auflistung aller Deklarationen (Funktionen, Variablen, etc.), die von außerhalb des Moduls angesprochen werden können. Dementsprechend stellt sich ein modulorientiertes Programm als eine Menge von Modulen dar, die nur über ihre Schnittstellen interagieren. Die Darstellung eines solchen Programms geschieht mithilfe von Modulgraphen, in denen verschiedene Beziehungen zwischen den Modulen wie Benutztrelationen oder Kontrollfluß dargestellt werden.

Die Einschränkung hinsichtlich des Zugriffes erscheint zunächst als hinderlich. Wiederum zeigt sich jedoch, daß bei sorgfältiger Wahl der Schnittstellen keine Nachteile erwachsen. Dafür gewinnt man aber große Vorteile bei der Unterstützung der Anforderungen des Software-Entwurfs:

1. Die Anforderung nach *Korrektheit* wird von der modularen Programmierung in erheblichem Maße unterstützt. Kapselt man etwa die Datenstrukturen, auf denen ein Algorithmus arbeitet, in Module ein, so kann bei der Implementierung eines diese Module nutzenden Algorithmus die Konzentration allein auf dessen Funktionalität gerichtet werden, anstatt stets dabei Sorge tragen zu müssen, die Integrität der verwendeten Datenstrukturen aufrechtzuerhalten. Diese wird ja von den entsprechenden Modulen selbst gewährleistet.

2. Die *Erweiterbarkeit* wird durch eine modulare Strukturierierung der Software insoweit unterstützt, als zum einen die Erweiterung der Schnittstelle eines Moduls, also das Hinzufügen weiterer Vereinbarungen, keinerlei Rückwirkungen auf Code außerhalb des Moduls hat. Dies trifft selbst dann zu, wenn die Erweiterung eine vollkommene Umstrukturierung der Implementierung der internen Datenstrukturen des Moduls bedingt. Zum ander wird die Erweiterung von Software um zusätzliche Module über die Kompatibilität unterstützt.

3. Die *Kompatibilität* verschiedener Module zueinander wird ebenfalls durch die Trennung von Schnittstelle und Implementierung gewährleistet; kein Modul kann die Integrität eines anderen beeinflußen, da interne Daten nicht angesprochen werden können. Damit ist eine gute *Wiederverwendbarkeit* gesichert.

4. Wie die (Laufzeit-)*Effizienz* von der modularen Programmierung berührt wird, hängt im wesentlichen von der Ausgestaltung der Schnittstellen ab. Gleichzeitig ermöglicht die geringere Komplexität der Software durch die modulare Programmierung ein besseres Verständnis, und bietet somit bessere Möglichkeiten zur Optimierung, z.B. indem man die Datenstrukturen hinter einer Modulschnittstelle durch effizienter austauscht.

Module eignen sich besonders zur Implementierung abstrakter Datentypen. Das sind benutzerdefinierte Datentypen, die ausschließlich über die Schnittstelle manipuliert werden können. Daher kommt der Schnittstelle (und ihrer Semantik) eine besondere Bedeutung zu, denn sie definiert den Datentyp. Die Schnittstelle wird deshalb als Teil des Datentyps angesehen. Zu einem abstrakten Datentyp können wie bei elementaren Datentypen Variablen erzeugt werden. Diese werden dann Instanzen oder auch Objekte genannt.

3.1.1.2 Objektorientierte Programmierung

Zur Motivation der objektorierten Programmierung zeigen wir zunächst am Beispiel des Simplex-Algorithmus, daß dieser Programmierstil die Wiederverwendbarkeit besser un-

terstützt als die imperative Programmierung. Wie in Abschnitt 1.6 dargestellt, gibt es eine große Vielfalt unterschiedlicher Pricing-Strategien, die jeweils zu einem korrekten Simplex-Algorithmus führen (eine geeignetes Kreiselvermeidungsverfahren vorausgesetzt). Um diese Vielfalt programmtechnisch mit einem imperativen Programmierstil zu erfassen, wird in der Simplex-Schleife eine Verzweigung eingesetzt, die abhängig von einem Parameter die eine oder die andere Pricing-Funktion aufruft. Will man nun eine weitere Pricing-Strategie hinzufügen, so ist man dazu nur in der Lage, wenn man über den Quellcode verfügt und die Verzweigung entsprechend erweitert. Dies widerspricht den Anforderungen nach Erweiterbarkeit und Wiederverwendbarkeit.

Der Grund für die eingeschränkte Wiederverwendbarkeit liegt in Art, wie bei der imperativen Programmierung Funktionsaufrufe aufgelöst werden. Dies geschieht bereits zur Übersetzungszeit, so daß die Integration einer weiteren Pricing-Funktion eine Änderung des Aufrufes und eine erneute Übersetzung erfordert.

Um eine generische (also allgemein verwendbare) Simplex-Schleife zu programmieren, muß die Auflösung des Funktionsaufrufs für das Pricing erst zur Laufzeit erfolgen. Dies wird von objektorientierten Programmiersprachen besonders unterstützt[1]. Damit kann die Simplex-Schleife generisch formuliert werden, so daß sie für alle zukünftigen Pricing-Verfahren ohne Änderung geeignet ist. Verschiedene Pricing-Strategien werden in verschiedenen abstrakten Datentypen implementiert, die jedoch untereinander große Ähnlichkeiten aufweisen: Die Schnittstelle jedes dieser Datentypen enthält z.B. eine Funktion `selectEnter()`, um den Index der in die Basis eintretenden Variable auszuwählen. Jeder Datentyp implementiert jedoch bei dieser Funktion eine andere Pricing-Stategie. Zur Lösung eines LPs wird dem Simplex-Algorithmus eine Instanz eines solchen Pricing-Datentyps übergeben, das wir das Pricing-Objekt nennen. In der Simplex-Schleife wird nun die zu dem übergebenen Objekt gehörende Funktion `selectEnter()` aufgerufen. Dabei hat der aufrufende Code (also die Simplex-Schleife) keine Kontrolle darüber, welche Funktion und damit welche Pricing-Strategie tatsächlich ausgeführt wird. Dies ist allein durch das Pricing-Objekt bestimmt; ihm wird somit eine eigene Autonomie zugesprochen.

Die zentralen Begriffe der objektorientierten Programmierung sind *Klasse*, *Objekt* und *Methode*. Eine Klasse ist ein (abstrakter) Datentyp. Ihre Instanzen heißen Objekte und die Funktionen ihrer Schnittstelle heißen Methoden. Zwei zugehörige Konzepte unterscheiden Klassen von ihren imperativen Gegenstücken:

Dynamic Typing: Darunter versteht man die Möglichkeit, beliebige Objekte als Parameter von Methoden oder Funktionen zu übergeben ohne Rücksicht auf ihre Klasse (oder ihren Typ). Dies steht im Gegensatz zum static Typing, bei dem der Typ von Parametern zur Übersetzungszeit festgelegt ist. Dies ist der Standard bei gängigen imperativen Programmiersprachen wie C, FORTRAN oder PASCAL. Für das obige Beispiel bedeutet dynamic Typing, daß der

[1] Es ist auch möglich per Funktionszeiger eine generische Simplex-Schleife zu realisieren. Dies kann jedoch als eine objektorientierte Strukturierung per Hand angesehen werden.

Simplex-Löser jedes beliebige Objekt akzeptieren muß. Dies gilt insbesondere auch für Objekte von Pricing-Klassen, die erst später entwickelt wurden. Sofern die Methode `selectEnter()` korrekt implementiert wurde, erhält man stets einen funktionierenden Simplex-Algorithmus, ohne daß eine Änderung des Programmtextes notwendig wäre.

Late Binding: Als late Binding wird die Art des Methodenaufrufs bei der objektorientierten Programmierung bezeichnet, daß nämlich erst zur Laufzeit anhand des benutzten Objektes und der gewählten Methode festgelegt wird, welche Funktion tatsächlich ausgeführt wird. Im imperativen Fall steht dagegen die auszuführende Funktion bereits zur Übersetzungszeit fest.

Die Folge dieser Konzepte ist der sog. Paradigmenwechsel beim Übergang von der imperativen zur objektorientierten Programmierung. Bei der imperativen Programmierung befiehlt (imperare = befehlen) stets der aufrufende Code, welche Funktion ausgeführt wird. Dagegen werden bei der objektorientierten Programmierung Objekte als autonome Instanzen angesehen, die eigenständig zur Laufzeit „entscheiden“, welche Aktion sie bei einem Methodenaufruf ausführen.

Zur Beschreibung eines weiteren Konzeptes der objektorientierten Programmierung, nämlich der Ableitung, kehren wir zum Beispiel der Pricing-Klassen zurück. Offenbar ähneln sich alle Pricing-Klassen insofern, als daß sie (jedenfalls in Teilen) dieselbe Schnittstelle aufweisen. Um Ähnlichkeiten zwischen Klassen programmtechnisch auszudrücken und dabei eventuell redundante Implementierungen zu vermeiden, bietet die objektorientierte Programmierung den Mechanismus der *Ableitung* von Klassen. Sie ist über folgenden Zusammenhang definiert[2]: Seien A und B Klassen, wobei B von A *abgeleitet* sei. Dann kann jede Methode der Klasse A auch von Objekten der Klasse B ausgeführt werden. Dies hat zur Folge, daß Objekte von B überall dort eingesetzt werden können, wo Objekte von A erwartet werden. Man nennt A *Basisklasse* von B.

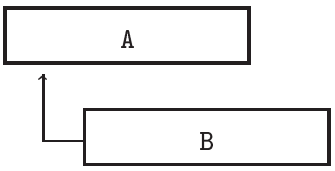


Abbildung 3.1: Ableitungsgraph

Die Ableitung von Klassen kann für zweierlei Konstruktionen verwendet werden, nämlich zur Erweiterung und zur Änderung von Klassen. Benötigt man eine Klasse, die zusätzlich zur Schnittstelle einer bereits bestehenden Klasse einige weitere Methoden anbietet, so kann dies durch Ableitung realisiert werden, bei der lediglich die zusätzlichen Methoden implementiert werden müssen. Wird hingegen eine Klasse gebraucht, die bei gleicher Schnittstelle eine andere Implementierung oder Semantik einiger Methoden aufweist, so kann dies wiederum per Ableitung geschehen, wobei lediglich die zu ändernden Methoden implementiert werden müssen. Natürlich sind auch Mischformen beider Konstruktionen möglich.

Zur Darstellung von Ableitungsbeziehungen zwischen Klassen werden sog. Ableitungs-

[2]Genaugenommen erklärt dies den wichtigen Fall der „public“ Ableitung.

graphen eingesetzt (vgl. Abb. 3.1). Jede Klasse wird durch einen Kasten repräsentiert, und eine Ableitungsbeziehung durch einen Pfeil von der abgeleiteten Klasse zur Basisklasse[3].

3.1.2 Einordnung von C++

Die Programmiersprache C++ erfreut sich derzeit weiter Verbreitung, ist sie doch aufgrund ihrer Ähnlichkeit zu C schnell zu erlernen, und bietet darüber hinaus ein genügendes Maß an Unterstützung für objektorientierte Software-Entwicklung. Dabei wurde auf eine vollkommene Unterstützung aller Leistungsanforderungen der objektorientierten Programmierung zugunsten einer Übersetzbarkeit in ebenso effizienten Code wie C verzichtet. Aus diesem Grund wurde C++ auch für die hier vorgestellten Implementierungen verwendet.

Ableitung von Klassen wird von C++ mittels *Vererbung* realisiert. Dies bedeutet, daß der Compiler für die abgeleitete Klasse eine Kopie der Basisklasse erzeugt und dieser die zusätzlichen Methoden und internen Daten hinzufügt[4].

C++ ist wie C eine statisch getypte Sprache und bietet somit kein echtes dynamic Typing. Es gibt jedoch zwei Ansätze eine Vereinfachung davon zu unterstützen. Zum einen sind dies parametrisierte (`template`) Klassen oder Funktionen, bei der ein Typ Parameter sein kann. Damit läßt sich z.B. ein Stapel für Objekte einer beliebigen aber festen Klasse realisieren. Zum anderen geschieht eine automatische Typanpassung, wenn ein Objekt einer abgeleiteten Klasse als Parameter übergeben wird, wo eigentlich ein Objekt seiner Basisklasse erwartet wird. Dies ist möglich, da die abgeleitete Klasse stets eine Kopie der Basisklasse beinhaltet. Um vom abgeleiteten Typ wieder zum Ursprungstyp eines Objektes zu gelangen, sieht der C++ Standard das sog. RTTI (runtime type information) vor. Dies wird jedoch noch kaum von Compilern unterstützt.

Im Normalfall werden nach einer automatischen Typanpassung die Methoden der Basisklasse benutzt, selbst wenn diese von der abgeleiteten Klasse überlagert wurden. Der Grund dafür ist, daß so eine Codeoptimierung mit Method-Inlining[5] möglich ist. Ist es für die Korrektheit des Programms hingegen wichtig, daß ein late Binding geschieht, d.h. die überlagerten Methoden der abgeleiteten Klasse benutzt werden, so kann dies für jede Methode gesondert spezifiziert werden (`virtual` Methoden). Für solche Methoden ist jedoch oft ein Inlining nicht möglich, so daß ein Methodenaufruf etwa den Aufwand eines Funktionsaufrufes in C bedingt.

C++ weist somit Einschränkungen sowohl hinsichlich des late Bindings wie auch beim dynamic Typing auf. Da beides nicht streng unterstützt wird, handelt es sich bei C++ auch

[3]Es herrscht ein erbitterter Streit unter den Informatikern, ob die Pfeile in der oben beschriebenen oder in der umgekehrten Richtung zu zeichnen seien. Sowohl für als auch gegen beide Varianten sprechen verschiedene Auffassungen. Die hier verwendete Pfeilrichtung entspricht der Bedeutung „erbt von“ oder „ist ein“ [45].

[4]Dies gilt nicht für den Fall von `virtual`-Vererbung.

[5]Beim function-inlining wird anstelle eines Unterprogrammaufrufes der Funktionscode direkt in den aufrufenden Code kopiert. Dadurch entfällt der Bearbeitungsaufwand für die Parameterübergabe.

nicht um eine „echte“ objektorientierte Programmiersprache. Dafür werden aber Approximationen bereitgestellt, die im überwiegenden Teil von Anwendungen, insbesondere aus dem mathematisch-naturwissenschaftlichen Bereich, den Anforderungen nach Objektorientierung genügen. Sie wurden so ausgelegt, daß der Rechenaufwand eines Methodenaufrufes auf den Aufwand eines Funktionsaufrufes in einem gewöhnlichen C-Programm beschränkt bleibt. Damit eignet sich C++ wie C für die effiziente Implementierung mathematischer Algorithmen.

3.2 Klassen und ihre Beziehungen

In diesem Abschnitt wird der SoPlex zugrundeliegende objektorientierte Software-Entwurf beschrieben; die beiden parallelen Erweiterungen DoPlex und SMoPlex werden im folgenden Abschnitt 3.3 diskutiert.

Einem objektorientierten Entwurf folgend, gliedert sich SoPlex in eine Vielzahl miteinander in Beziehung stehender Klassen. Es werden keine globalen Datenstrukturen verwendet, da solche stets das Problem der Inkompatibilität zu Datenstrukturen anderer Komponenten eines komplexen Programms in sich bergen. Dies ist besonders wichtig für einen LP-Löser, der in einer Vielzahl von Anwendungen eingesetzt werden soll.

Es werden drei Kategorien von Klassen unterschieden. Ein Satz von elementaren Klassen implementiert grundlegende Datentypen, die auch in anderen Projekten Anwendung finden können. Etwas mehr auf die Implementierung von SoPlex ausgerichtet sind die Vektor-Klassen. Sie implementieren verschiedene Datentypen zur Repräsentation von Vektoren bis hin zu ganzen LPs. Auch wenn sie über umfangreiche Schnittstellen zur Manipulation oder Operation (etwa der Vektoraddition) verfügen, ist ihr primärer Charakter doch der der Einkapselung von Daten. Im Gegensatz dazu dienen die algorithmischen Klassen der Kapselung von (Teil-)Algorithmen.

3.2.1 Elementare Klassen

Es wurden eine Reihe von Klassen implementiert, die grundlegende Datentypen wie dynamische Felder oder verkettete Listen bereitstellen. Entsprechende Klassen finden sich wohl in jedem Projekt, weshalb inzwischen eine standardisierte Klassenbibliothek für C++ entwickelt wurde. Sie stand jedoch zu Beginn des Projektes noch nicht zur Verfügung.

An dieser Stelle soll exemplarisch die `template`-Klasse `DataArray` vorgestellt werden, um daran den Nutzen auch solch einfacher Klassen bei der Programmentwicklung aufzuzeigen. Die Klasse `DataArray<T>` dient zur Bereitstellung von Feldern über einem festen Typ `T`. Somit entspricht ein `DataArray<double>` lediglich einem Feld von Gleitkommazahlen. Es bestehen jedoch mehrere Unterschiede in der Nutzbarkeit gegenüber normalen C-Feldern. Während für ein Feld explizit der Speicherplatz angefordert und schießlich

wieder freigegeben werden muß, kann man ein `DataArray<double>` wie eine automatische Variable instantiieren. Dabei wird der verlangte Speicherplatz automatisch alloziiert, und sobald der Geltungsbereich der Variablen endet, wird der Speicherplatz ohne Zutun des Programmierers wieder freigegeben. Dies zu vergessen, ist eine häufige Fehlerquelle bei der Speicherverwaltung „von Hand“. Ferner kann zur Fehlersuche der Zugriff auf die Feldelemente mit einer Schrankenüberprüfung versehen werden, die man für optimierte Versionen abschaltet. Die Klasse `DataArray` bietet somit eine sichere Speicherverwaltung für Felder eines beliebigen aber festen Typs. Redundanz, die bei Reimplementierung der Speicherverwaltung für jedes benutzte Feld aufkommt, wird durch die Lokalisierung in der Klasse `DataArray` vermieden. Dies zeigt eine grundlegende Vorgehensweise bei der objektorientierten Programmierung, nämlich mehrfach verwendbare Konzepte in Klassen zu implementieren.

Folgende weitere Grundklassen wurden für SoPlex entwickelt:

`IsList`	einfach verkettete Liste
`IdList`	doppelt verkettete Liste
`IdRing`	zu einem Ring geschlossene, doppelt verkettete Liste
`DataHashTable`	Hash-Tabelle
`Sorter`	Sortierklasse
`CmdLine`	Parser für die Argumentenliste von Programmen
`Random`	Zufallszahlengenerator
`Timer`	eine „Stoppuhr“

Bis auf die letzten drei Klassen handelt es sich wieder um `template`-Klassen, die in unterschiedlichsten Zusammenhängen eingesetzt werden können. Die Klassen `CmdLine`, `Random` und `Timer` entsprechen hingegen einem rein modulorientierten Programmierstil.

3.2.2 Vektor-Klassen

Die Vektor-Klassen implementieren abstrakte Datentypen zur Linearen Algebra, und zwar vom Einheitsvektoren bis hin zum vollständigen LP. Sie können wie Variablen benutzt werden, also als solche instantiiert und in Ausdrücken verwendet werden. Dabei beschränken sich jedoch die Schnittstellen auf Zugriffs- und Manipulationsmethoden sowie einfache Operationen, deren algorithmische Umsetzung kaum Alternativen zuläßt. Wo dies der Fall ist, etwa bei der Lösung linearer Gleichungssysteme oder ganzer LPs, kommen hingegen algorithmische Klassen zum Einsatz (vgl. Abschnitt 3.2.3).

Im folgenden werden die wichtigsten Ableitungs-Hierarchien und Benutzt-Relationen zwischen den Vektor-Klassen skizziert. Dies stellt keine Dokumentation der Klassen dar, sondern soll lediglich die Struktur des Entwurfes aufzeigen. Dennoch wird in manchen Fällen auch die interne Verwaltung der Daten aufgezeigt, soweit es dem Verständnis der Implementierung dienlich erscheint.

3.2.2.1 Dünnbesetzte Vektoren

Wie in Abschnitt 1.7 beschrieben, bedeutet die Dünnbesetztheit eines Vektors, daß es sich lohnt, nur die NNEs zu verwalten. Es gibt eine Vielzahl von Vorschlägen für geeignete Darstellungen, die sich in der Effizienz für verschiedene Operationen unterscheiden [39]. In Abschnitt 1.7.3.3 wurden verschiedene Speicherschemata für dünnbesetzte Matrizen vorgestellt und bewertet. Dieselben Überlegungen treffen auch für dünnbesetzte Vektoren zu. Deshalb verwaltet die Klasse `SVector` (Sparse Vector) die NNEs in einem Feld von Wert-Index-Paaren (Klasse `SVector::Element`). Eine andere, wegen früherer Implementierungen in FORTRAN oft eingesetzte Lösung, ist die getrennte Verwaltung der Indizes und Werte in je einem Feld. Sie wurde aus zwei Gründen nicht gewählt. Zum einen benötigte das Verschicken eines `SVector`-Objektes in diesem Fall zwei Kommunikationen, während bei der verwendeten Darstellung eine Kommunikation ausreicht. Zum anderen kann es bei zwei Feldern leichter zu Cache-Misses kommen, falls Indizes und Werte auf dieselben Cachelines abgebildet werden.

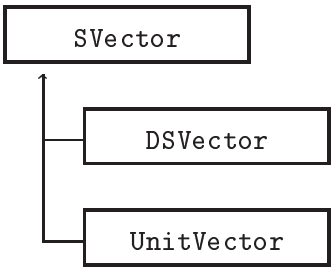


Abbildung 3.2: Klassen für dünnbesetzte Vektoren

Das NNE-Feld selbst wird nicht von `SVector` alloziiert. Stattdessen muß der zu verwendende Speicherplatz bei der Instantiierung eines `SVector`-Objektes übergeben werden. Die Größe des Feldes bestimmt, wieviele NNEs der `SVector` maximal aufnehmen kann. Benötigt man einzelne dünnbesetzte Vektoren, so ist diese Instantiierung und Restriktion unnötig kompliziert. Deshalb wird mit `DSVector` eine weitere Klasse bereitgestellt, die per Ableitung die Funktionalität von `SVector`en um eine dynamische Speicherverwaltung erweitert. `DSVector`en stellen den NNE-Speicher intern zur Verfügung und verwalten ihn dynamisch je nach Anforderung. Somit braucht von außen kein Speicher bereitgestellt zu werden, und es können ohne Beschränkungen NNEs hinzugefügt werden.

Benötigt man jedoch eine Vielzahl von dünnbesetzten Vektoren, so ist die Instantiierung vieler `DSVector`en nicht geeignet, da dabei viele Speicherblöcke alloziiert würden, was eine hohen Ressourcenverbrauch bedingt. Stattdessen kann eine Menge von `SVector`en einer gemeinsamen Speicherverwaltung unterworfen werden, die von der Klasse `SVSet` implementiert wird. Die Details der interne Speicherverwaltung wurden bereits in Abschnitt 1.7.3.3 vorgestellt.

Als zweite von `SVector` abgeleitete Klasse implementiert `UnitVector` Einheitsvektoren. Sie treten in SoPlex für die Darstellung von Schlupf-Variablen oder einfache Schrankenungleichungen auf. An dieser Stelle wird der Ableitungsmechanismus statt zur Erweiterung einer Klasse zu ihrer *Spezialisierung* eingesetzt: Ein `UnitVector` ist ein `SVector` mit nur einem NNE, das immer den Wert 1 hat. Alle mathematischen Operationen von dünnbesetzten Vektoren sind aber auch für Einheitsvektoren sinnvoll, was durch die Ableitungsbeziehung software-technisch ausgedrückt wird.

3.2.2.2 Dichtbesetzte Vektoren

Abbildung 3.3 zeigt die Klassenhierarchie für dichtbesetzte Vektoren. Wie schon bei den Klassen für dünnbesetzte Vektoren implementiert `Vector` die Schnittstelle für den Zugriff auf Elemente sowie für mathematische Operationen auf Vektoren, während `DVector` die Speicherverwaltung übernimmt.

Von `DVector` sind zwei weitere Klassen abgeleitet, die speziell für die Implementierung des Simplex-Algorithmus konzipiert wurden. Die Klasse `UpdateVector` wird für die Simplex-Vektoren f, g und h verwendet. Diesen werden in jeder Iteration das Vielfache eines Vektors Δf, Δh bzw. Δh hinzuaddiert. Deshalb erweitert `UpdateVector` die Klasse `DVector` um Methoden zur Verwaltung des Faktors und des zusätzlichen Vektors sowie eine Methode zur Ausführung des Updates.

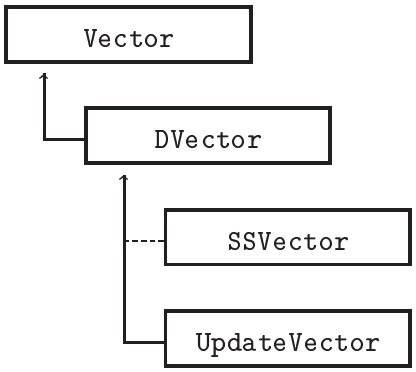


Abbildung 3.3: Klassen für dichtbesetzte Vektoren

Aus Effizienzgründen ist es oft notwendig, sowohl einen direkten Zugriff sowohl auf das i-te Element als auch auf die NNEs eines dünnbesetzten Vektors zu haben. Die erste Anforderung erfüllt die Klasse `Vector`, während der Zugriff auf NNEs von `SVector`en unterstützt wird. Die Klasse `SSVector` (Semi Sparse Vector) erlaubt beides, indem sie der Funktionalität ihrer Basisklasse `DVector` die Verwaltung der Indizes der Nicht-Null-Elemente hinzufügt. `SSVector`en werden in SoPlex für die Vektoren Δf, Δg bzw. Δh der `UpdateVector`en eingesetzt.

3.2.2.3 Von Vektormengen zum LP

Zur Repräsentation eines allgemeinen LPs der Form (1.45) müssen folgende Daten gespeichert werden:

- Optimierungsrichtung
- Zielfunktionsvektor
- rechte und linke Seite
- obere und untere Schranken
- (dünnbesetzte) Nebenbedingungsmatrix

Statt alles in nur eine Klasse zu implementieren, lohnt es sich, dies mit einer Ableitungshierarchie zu realisieren, da die auftretenden Klassen auch in anderen Zusammenhängen verwendet werden können, insbesondere als Parameter bei den Manipulationsmethoden für das LP. Die Ableitungshierarchie wird von Abbildung 3.4 gezeigt.

Die Klasse `SVSet` (Sparse Vector Set) wurde bereits oben erwähnt. Sie implementiert die

Speicherverwaltung für eine Menge von `SVector`en und bietet Methoden zum Hinzufügen und Entfernen ganzer `SVector`en sowie solche zum Hinzufügen und Entfernen von NNEs für einzelne `SVector`en in einem `SVSet`. Intern speichert ein `SVSet` die NNEs seiner `SVector`en in einem Feld, das wie in Abschnitt 1.7.3.3 beschrieben verwaltet wird.

Die Klasse `SVSet` benutzt (durch die gepunktete Linie symbolisiert) die Klasse `DataSet`, die sog. `Key`s für die in einem `DataSet` gespeicherte Einträge verwaltet. Fügt man einen Eintrag einem `DataSet` hinzu, so vergibt dieses ein eindeutiges `Key`, über das der Eintrag jederzeit in dem `DataSet` referenziert werden kann. Dieses `Key` bleibt solange gültig, wie sich der Eintrag im `DataSet` befindet, insbesondere auch dann, wenn andere Daten hinzugefügt oder entfernt werden. Dagegen kann sich die Numerierung der Einträge bei solchen Operationen ändern. Diese Funktionalität wird bis hin zur Verwaltung der Zeilen und Spalten des LPs durchgereicht. Dort eignet sie sich z.B. für die Implementierung von Schnittebenen-Verfahren, bei denen laufend Ungleichungen von einem LP entfernt bzw. ihm hinzugefügt werden. Dabei ändert sich zwar stets die Numerierung der Ungleichungen, ihre `Key`s (die dann allerdings `SPxRowId` bzw. `SPxColId` heißen) bleiben jedoch unverändert.

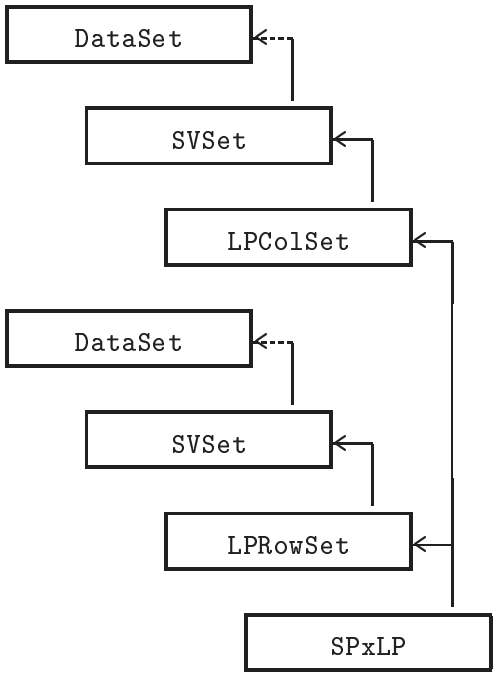


Abbildung 3.4: Klassengraph für SPxLP

Die Klasse `SPxLP` dient der Repräsentation von LPs. Sie ist sowohl von `LPColSet` als auch von `LPRowSet` abgeleitet. Letztere dient der Verwaltung von Bereichsungleichungen (Klasse `LPRow`), und bietet dafür umfangreiche Zugriffs- und Manipulationsmethoden. Entsprechend stellt die Klasse `LPColSet` Methoden für den Zugriff auf LP-Spalten (Klasse `LPCol`), bestehend aus Zielfunktionswert, oberer und unterer Schranke sowie (dünnbesetztem) Spaltenvektor, und zu deren Manipulation bereit. Sowohl `LPRowSet` als auch `LPColSet` sind von `SVSet` abgeleitet, das die Verwaltung der Zeilen- bzw. Spaltenvektoren übernimmt. Somit ist in `SPxLP` die Nebenbedingungsmatrix zweifach gespeichert, und zwar zeilen- und spaltenweise. Dies erlaubt eine besonders einfache Implementierung des transparenten Umschaltens zwischen beiden Basisdarstellungen und hat darüber hinaus Vorteile bei der effizienten Implementierung des Matrix-Vektor-Produktes.

3.2.2.4 Die LP-Basis

Die Klasse `SPxBasis` verwaltet eine allgemeine LP-Basis (vgl. Definition 6). Dies umfaßt folgende Aufgaben:

- Verwaltung der Basis-Indexvektoren,
- Verwaltung der Basismatrix und
- mathematische Operationen mit der Basismatrix.

Die Basismatrix wird als `DataArray` von Zeigern auf ihre Vektoren gespeichert. Einfache Operationen, wie die Multiplikation mit einem Vektor werden direkt von `SPxBasis` implementiert. Dagegen werden komplexe Algorithmen zur Lösung von Gleichungssystemen einem Objekt einer Gleichungssystem-Löser-Klasse (eine Implementierungsklasse von `SLinSolver`, vgl. Abschnitt 3.2.3.4) übertragen.

Jeder Index befindet sich zu einem Zeitpunkt in genau einem der Basisindexvektoren $B_x, \ldots, N_f$ (Basisbedingung 2). Deshalb wird der *Mengencharakter* der Indexvektoren durch eine Abbildung jedes Index auf einen Status (in einem `DataArray` gespeichert) implementiert. Lediglich für die Basisindizes ist eine Reihenfolge wichtig. Dies wird erreicht, indem die Basisindizes zusätzlich in einem `DataArray` gespeichert werden.

3.2.3 Algorithmische Klassen

Innerhalb eines Simplex-Algorithmus kann eine Reihe von Teilproblemen identifiziert werden, für die jeweils unterschiedliche Implementierungen denkbar sind. Diese sind:

- das Pricing,
- der Quotiententest,
- das Generieren einer Startbasis,
- die Lösung linearer Gleichungssysteme und
- eventuell ein Preprocessing des LPs.

Für jedes dieser Teilprobleme können verschiedene Algorithmen eingesetzt werden. Dies soll durch den Software-Entwurf so unterstützt werden, daß die Implementierung offen für zukünftige Varianten ist. Wie in Abschnitt 3.1.1.2 erläutert, eignet sich dafür ein objektorientierter Entwurf besonders gut.

Jedes Teilproblem wird durch eine *abstrakte Basisklasse* repräsentiert. Eine solche weist nur `virtual`-Methoden auf und beschreibt damit die *Schnittstelle*, die ein Algorithmus zur

Lösung eines Teilproblems implementieren muß. Die Implementierung eines Algorithmus zur Lösung eines Teilproblems geschieht daher in einer von der jeweiligen abstrakten Basisklasse abgeleiteten Klasse. Solche Klassen werden *Implementierungsklassen* genannt.

Für jedes Teilproblem gibt es einen eigenen Klassenbaum. Die Wurzel bildet jeweils die abstrakte Basisklasse, die die Schnittstelle für das Teilproblem definiert. Verschiedene abgeleiteten Klassen implementieren diese Schnittstelle mit unterschiedlichen Algorithmen zur Lösung des Teilproblems.

Die Klasse `SoPlex` implementiert den Simplex-Algorithmus nur unter Verwendung der abstrakten Basisklassen für die zu lösenden Teilprobleme. Zum Lösen der Teilprobleme werden SoPlex Instanzen der Implementierungsklassen zur jeweils gewünschten algorithmischen Variante übergeben. SoPlex arbeitet mit jeder Implementierungsklasse korrekt. Dies gilt insbesondere auch für jede in Zukunft entwickelte Implementierungsklasse. Dadurch ist es möglich, z.B. problemspezifische Pricer zu entwickeln und mit SoPlex einzusetzen.

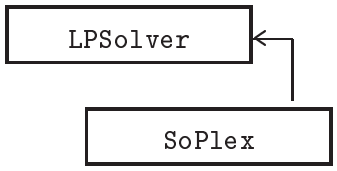


Abbildung 3.5: Klassengraph für SoPlex

Da SoPlex in größeren Programmen, bei denen LPs gelöst werden müssen, eingesetzt werden soll, ist es selbst auch nur eine mögliche Implementierung eines LP-Lösers. Deshalb wurde die Klasse `SoPlex` derselben Struktur unterzogen. Sie ist eine Implementierungsklasse der abstrakten Basisklasse `LPSolver`. Algorithmen, die mit Hilfe der Schnittstellenklasse `LPSolver` formuliert werden, können somit ohne Änderung auf andere Implementierungsklassen dieser Schnittstelle umgestellt werden.

3.2.3.1 Quotiententest Klassen

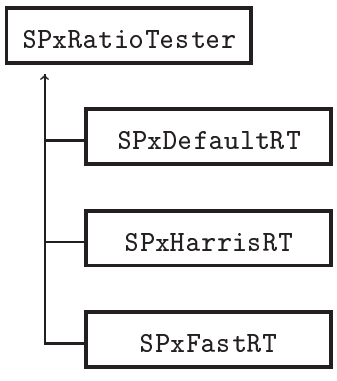


Abbildung 3.6: Klassengraph der Quotiententest-Klassen

Verschiedene Implementierungen des Quotiententests zeichnen sich durch unterschiedliche Effizienz und numerische Stabilität aus. Die optimale Wahl kann von den zu lösenden LPs abhängen. Für SoPlex wurden die drei in Klassengraph 3.6 angegebenen Implementierungsklassen der abstrakten Basisklasse für den Quotiententest `SPxRatioTester` entwickelt. Die zugrundeliegenden Algorithmen wurden inhaltlich bereits in Abschnitt 1.3.4 behandelt. Klasse `SPxDefaultRT` implementiert die einfachste stabilisierte Version, also den „Textbook“ Quotiententest. Die Stabilisierung nach Harris wird von Klasse `SPxHarrisRT` implementiert, während Klasse `SPxFastRT` den für SoPlex entwickelten stabilen Quotiententest realisiert.

3.2.3.2 Pricing Klassen

Der Klassengraph 3.7 der Pricing-Klassen weist `SPxPricer` als abstrakte Basisklasse für das Pricing aus. Derzeit existieren 6 verschiedene Implementierungsklassen. Sie implementieren folgende Verfahren aus Abschnitt 1.6:

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

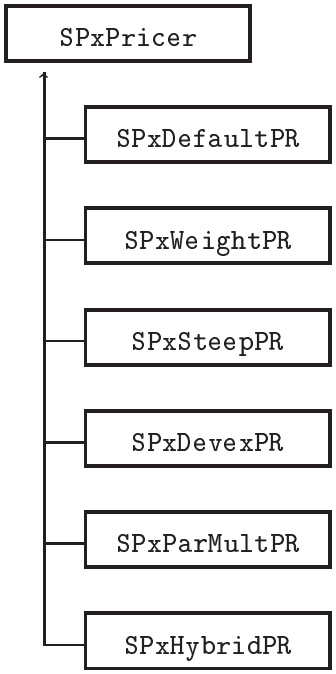


Abbildung 3.7: Klassengraph der Pricing-Klassen

3.2.3.3 Startbasis Klassen

Die Generierung einer geeigneten Startbasis kann einen großen Einfluß auf die Iterationszahl bei der Lösung des LPs haben. Eventuell können dabei Informationen aus dem mathematischen Modell, das zu dem LP führt, genutzt werden, um so eine bessere Startbasis zu konstruieren. Um solch eine problemspezifische Startbasis in SoPlex integrieren zu können, wurde die abstrakte Basisklasse `SPxStarter` eingeführt. Derzeit gibt es lediglich eine Implementierungsklasse, nämlich `SPxWeightST`. Sie versucht, anhand von Gewichten für die Indizes eine möglichst „leichte“ Basis zu konstruieren, so wie es in Abschnitt 1.8.6 beschrieben wurde. Mittels abgeleiteter Klassen können auch andere Gewichte verwendet werden, indem die Methode zur Berechnung der Gewichte überlagert wird.

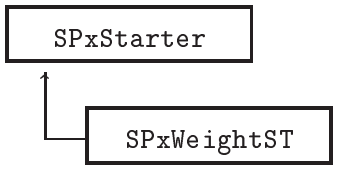


Abbildung 3.8: Klassengraph der Startbasis-Klassen

3.2.3.4 Löser für lineare Gleichungssysteme

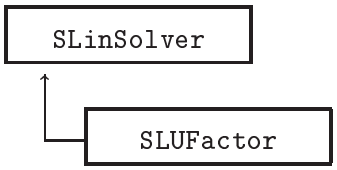


Abbildung 3.9: Klassengraph der Löser für dünnbesetzte lineare Gleichungssysteme

Die Lösung linearer Gleichungssysteme ist einer der wichtigsten und rechenintensivsten Teilschritte von Simplex-Algorithmen. Auch wenn für SoPlex derzeit lediglich die in 1.7.3 beschriebene Implementierung vorliegt, wurde auch hier das Entwurfskonzept für algorithmische Komponenten zugrunde gelegt (vgl. Klassengraph 3.9).

3.2.3.5 Preprocessing Klassen

Oft werden LPs maschinell von anderen Programmen erzeugt. Dabei kommt es z.B. vor, daß Ungleichungen (oder Spalten) mehrfach erzeugt werden. Mathematisch ist dies kein Problem, algorithmisch erhöht sich damit jedoch der Rechenaufwand bei der Lösung. Aber auch weniger offensichtliche „überflüssige“ Zeilen oder Spalten können auftreten, wenn z.B. lediglich gültige Ungleichungen statt Facetten verwendet werden. In solchen Fällen kann eine Vereinfachung des LPs zu erheblichen Geschwindigkeitsgewinnen bei der Lösung führen [18, 7]. Obwohl für SoPlex noch keine Implementierung dafür vorliegt, ist mit der abstrakten Basisklasse `SPxSimplifier` eine Schnittstelle für ein Preprocessing der LPs vorgesehen.

3.3 Klassen für parallele Implementierungen

In diesem Abschnitt werden die zusätzlichen Klassen für die Implementierungen SMoPlex für Parallelrechner mit gemeinsamem Speicher und DoPlex für solche mit verteiltem Speicher beschrieben. Beiden parallelen Implementierungen gemeinsam ist, daß sie wesentliche Teile per Ableitung von der sequentiellen Version SoPlex erben und lediglich die Zusätze für die Datenverteilung, Kommunikation und Synchronisation hinzufügen. Dadurch übertragen sich wichtige Eigenschaften von SoPlex wie z.B. die numerische Stabilität direkt auf die parallelen Versionen.

3.3.1 Gemeinsamer Speicher

Die Parallelisierung des Simplex-Algorithmus für Rechner mit gemeinsamem Speicher SMoPlex arbeitet direkt auf den Datenstrukturen der sequentiellen Implementierung SoPlex. Lediglich die Arbeit muß zwischen den beteiligten PEs aufgeteilt werden, was einer Synchronisation derselben bedarf. Dazu wurde die Klasse `ShmemObj` entwickelt, die im folgenden Abschnitt beschrieben wird. Anschließend wird ihr Einsatz für die Parallelisierung

beschrieben.

3.3.1.1 ShmemObj

Multiprocessor-Architekturen erfreuen sich z.Z. wachsender Beliebtheit, zumal sie immer preisgünstiger angeboten werden. Meist werden sie als Server eigesetzt, jedoch bietet jeder Hersteller auch eine Bibliothek, mit deren Hilfe parallele Programme entwickelt werden können. Eine solche bietet Funktionen zur Erzeugung und Terminierung sog. *Threads*, sowie Schloßvariablen, Semaphoren o.ä. Primitive zu deren Synchronisation. Threads sind „leichtgewichtige“ Prozesse, die alle Ressourcen, also Speicherplatz, File-Descriptoren etc. teilen.

Inzwischen wurde von der Normungskommission POSIX ein Standard für solche Funktionen definiert, der jedoch bislang nur von wenigen Anbietern implementiert ist. Deshalb wurde die Klasse `ShmemObj` entwickelt, die die wesentliche Funktionalität in einer objektorientierten Schnittstelle zur Verfügung stellt. Dadurch beschränkt sich der Portierungsaufwand für verschiedene Parallelrechner mit verteiltem Speicher auf eine Anpassung dieser Klasse.

Die wichtigere Aufgabe der Klasse `ShmemObj` ist jedoch die Implementierung eines Programmiermodells, das eine Parallelisierung objektorientierter Implementierungen auf Multiprozessoren mit gemeinsamem Speicher unterstützt. Dabei geht es darum, einzelne Methoden durch Parallelität zu beschleunigen. Dazu bietet `ShmemObj` die Methode `doParallel(f)` an, mit der eine Funktion bzw. Methode `f` gleichzeitig von mehreren Threads aufgerufen wird. Dadurch wird folgende Vorgehensweise bei der Parallelisierung unterstützt (vgl. Abbildung 3.10):

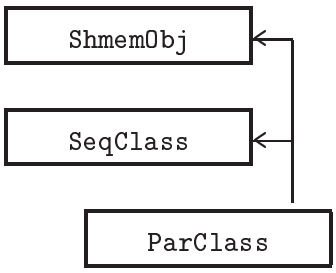


Abbildung 3.10: Einsatz der Klasse `ShmemObj` zur Überlagerung von Methoden einer sequentiellen Klasse `SeqClass` mit parallelen Implementierungen in eine Klasse `ParClass`

Ausgehend von einer sequentiellen Implementierung in einer Klasse `SeqClass` wird eine abgeleitete Klasse `ParClass` implementiert, in der eine zu parallelisierenden Methode von einer neuen überlagert wird. Diese veranlaßt mit `doParallel(f)` die parallele Ausführung der Methode `f(thread)` mit mehreren (auf unterschiedlichen PEs laufenden) Threads, wobei jeweils die zugehörige Threadnummer `thread` übergeben wird. Anhand der Threadnummer und der Anzahl der beteiligten Threads kann die Arbeit in der Methode `f` aufgeteilt werden. Bei der Implementierung der Methode `f` für die Klasse `ParClass`, wird idealerweise die Fuktionalität von `SeqClass` wiederverwendet. Durch die Ableitung von `SeqClass` kann die parallelisierte Klasse `ParClass` transparent eingesetzt werden, ohne daß andere Teile im Programm dies „bemerken“. Dadurch wird eine schrittweise Parallelisierung der rechenintensiven Methoden unterstützt.

Über die Bereitstellung der Methode `doParallel` hinaus ist die Klasse `ShmemObj` auch für die Erzeugung, Vernichtung und Synchronisation der Threads verantwortlich. Bei der Konstruktion eines `ShmemObj`-Objektes wird die Anzahl der bei `doParallel` zu verwendenden Threads angegeben. Diese werden sofort erzeugt, warten jedoch bis auf einen an einer Schloßvariablen. Der verbleibende Thread führt das weitere sequentielle Programm aus. Erst bei Aufruf der Methode `doParallel(f)` wird die Schloßvariable freigegeben, so daß die anderen Threads bei der parallelen Bearbeitung von `f` kollaborieren. Nach Termination synchronsisieren sich die Threads wieder über eine Schloßvariable.

Eine andere Möglichkeit die Methode `doParallel` zu implementieren wäre die Erzeugung der Threads bei jedem Aufruf und deren Vernichtung nach Termination. Da jedoch die Erzeugung und Vernichtung von Threads einen hohen Aufwand gegenüber der Synchronisation an einer Schloßvariablen bedeutet, wurde der oben beschriebene Ansatz gewählt.

Mithilfe der Klasse `ShmemObj` kann somit eine objektorientierte sequentielle Implementierung auf Methodenaufrufsebene parallelisiert werden. Dabei bleibt die sequentielle Reihenfolge der Methodenaufrufe erhalten, so daß bei dieser Parallelisierung zu einem Zeitpunkt immer nur ein paralleler Methodenaufruf in einem Objekt erfolgen kann. Damit nun nicht unnötig viele Threads verbraucht werden, die die meiste Zeit darauf warten, angestoßen zu werden, können verschiedene `ShmemObj`-Objekte dieselben Threads verwenden.

3.3.1.2 Die Klassen zu SMoPlex

Die parallele Implementierung `SMoPlex` für Parallelrechner mit gemeinsamem Speicher stützt sich auf die Klasse `ShmemObj` (vgl. Abbildung 3.11). Da `SoPlex` nur einen Teil des Simplex-Algorithmus implementiert, bietet `SMoPlex` auch nur für diesen Teil eine parallele Implementierung. Dies ist die Berechnung des Matrix-Vektor-Produktes für die Berechnung des Vektors Δg sowie die Aktualisierung der Vektoren. Die entsprechenden Methoden von `SoPlex` werden in `SMoPlex` durch parallelisierte Methoden überlagert, die `ShmemObj` zur Parallelisierung verwenden.

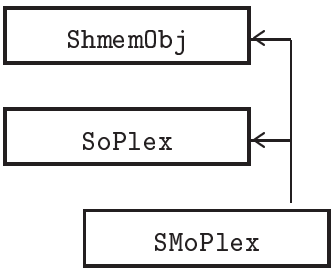


Abbildung 3.11: Parallelisierung von `SoPlex`

Andere zur Parallelisierung geeignete Funktionalität sind wie in Abschnitt 2.2 beschrieben das Pricing, der Quotiententest sowie die Lösung linearer Gleichungssysteme. Diese Funktionalität wurde ja in spezielle Klassen ausgelagert, so daß ihre Parallelisierung auch in entsprechend abgeleiteten Klassen erfolgt.

Abbildung 3.12 zeigt den Klassengraphen für die parallelisierte Version der steepest-edge Pricing-Klasse `SMxSteepPR`. Diese Klasse ist sowohl von `SPxSteepPR` als auch von `SMxPricer` abgeleitet. Die Klasse `SMxPricer` implementiert die Pricing-Methoden ihrer Basisklasse `SPxPricer`. Dadurch sind Objekte von `SMxSteepPR` als Pricing-Objekte für

`SoPlex` oder `SMoPlex` einsetzbar.

Die Klasse `SMxPricer` implementiert zwei Parallelisierungskonzepte für das Pricing. Zum einen ist dies das parallele Ausführen einer Pricing-Operation. Dazu werden die Indizes in soviele Teilmengen wie von `ShmemObj` bereitgestellte Threads unterteilt, so daß jeder Thread einen Kandidatenindex aus einer Teilmenge auswählt. Um dies zu tun, ruft die Pricing-Methode eine weitere Methode auf, die das Pricing nur auf der Teilmenge durchführt. Diese wurde bereits in `SPxSteepPR` implementiert, so daß die Pricing-Strategie bereits vollständig von der sequentiellen Klasse implementiert wird. Anschließend wählt `SMxPricer` den besten Kandidaten aus.

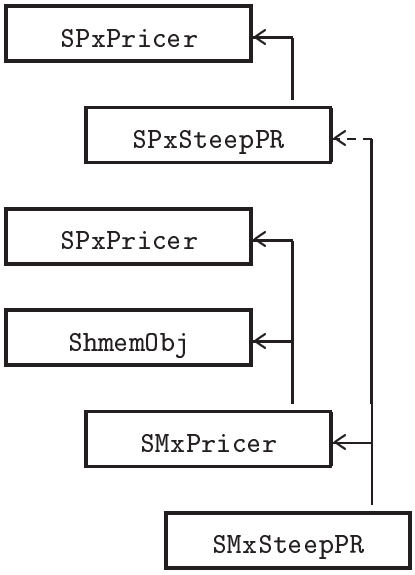


Abbildung 3.12: Parallelisierung des steepest-edge Pricings

Zum anderen implementiert die Klasse `SMxPricer` das Block-Pivoting. Dazu wird für jeden ausgewählten Kandidaten das zugehörige Lineare Gleichungssystem gelöst, und der beste Kandidat dem Simplex-Löser übergeben. In den folgenden Iterationen werden aber die anderen zuvor gewählten Kandidaten benutzt, für die die Lösung der Linearen Gleichungssysteme lediglich aktualisiert werden muß.

Der Vorteil dieser Vorgehensweise ist, daß sämtliche Komplikationen und neuen Prinzipien, die durch Parallelität ins Spiel gelangen, in der Klasse `SMxPricer` realisiert werden können. Dadurch kann mit Hilfe von abgeleiteten Klassen eine neue Pricing-Strategie einfach hinzugefügt werden, ohne daß man sich dabei Gedanken über Parallelität machen müßte. Existiert bereits eine sequentielle Version, wie das beim Beispiel des steepest-edge Pricings mit `SPxPricer` der Fall ist, kann die dort implementierte Funktionalität direkt verwendet werden.

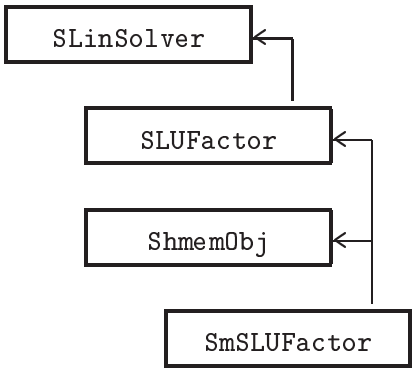


Abbildung 3.13: Klasse `SmSLUFactor` bietet die parallele Lösung zwei linearer Gleichungssysteme

Der soeben vorgestellten Strukturierung wurden auch die Quotiententest-Klassen unterzogen. Dabei wurde jedoch lediglich eine Parallelisierung über der Indexmenge implementiert, da es kein Äquivalent zum Block-Pivoting für den Quotiententest gibt.

Auch für die Lösung von linearen Gleichungssystemen wurde mit `SmSLUFactor` eine parallelisierte Klasse bereitgestellt, die die in Abschnitt 2.2.4 beschriebene parallele Lösung verschiedener Gleichungssysteme implementiert. Sofern in Zukunft parallele Löser für dünnbesetzte lineare

Gleichungssysteme verfügbar werden, die schneller sind als die sequentielle Implementierung, können sie an dieser Stelle auch zur parallelen Lösung einzelner Gleichungssysteme eingestetzt werden.

3.3.2 Verteilter Speicher

Auch die Implementierung für Parallelrechner mit verteiltem Speicher `DoPlex` nutzt per Ableitung den bestehenden sequentiellen Code von `SoPlex`. Lediglich die für die Verteilung notwendige Kommunikation wird hinzugefügt. Die dazu benötigte Funktionalität wird von der Klasse `DistrObj` bereitgestellt.

3.3.2.1 DistrObj

Die Klasse `DistrObj` spielt für Implementierungen auf Parallelrechnern mit verteiltem Speicher eine ähnliche Rolle wie `ShmemObj` für solche mit gemeinsamem Speicher. Sie bietet eine Abstraktion von der zu der zugrundeliegenden Architektur gebotenen Kommunikationsbibliothek, so daß sich eine Portierung lediglich auf eine Anpassung dieser Klasse beschränkt.

Darüberhinaus implementiert die Klasse `DistrObj` eine wichtige Operation, nämlich das Gossiping (vgl. Algorithmus 11). Dabei wird die Kommunikationsstruktur sowie die Verwaltung der Kommunikationspuffer implementiert; lediglich die auszuführende Gossiping-Operation muß von dem Benutzer bereitgestellt werden. Damit eignen sich von `DistrObj` abgeleitet Klassen besonders zur Implementierung datenparalleler Algorithmen.

Schließlich unterstützt die Klasse `DistrObj` eine Partitionierung des Parallelrechners. `DistrObj`s werden SPMD-artig instantiiert, wobei eine Teilmenge von PEs angegeben werden kann. Dabei wird auf jedem PE ein Objekt der Klasse `DistrObj` erzeugt; die Gesamtheit der lokalen `DistrObj`-Objekte wird *verteiltes Objekt* genannt. Bei der Implementierung von parallelen Algorithmen in abgeleiteten Klassen, „sehen“ lokale `DistrObj`-Objekte nur die zu ihrem verteilten Objekt zugehörigen `DistrObj`-Objekte. So liefert etwa die Methode `nPes()` die Anzahl der zum verteilten Objekt gehörenden PEs, die als logisch von 0 bis `nPes()-1` durchnumeriert erscheinen. Auch die Kommunikation erfolgt lediglich innerhalb eines verteilten Objektes. Insbesonder kommt es zu keinen Fehlern, wenn Nachrichten verschiedener verteilter Objekte mit derselben Identifikationssnummer verschickt werden; die Klasse `DistrObj` sorgt für eine eindeutige Zuordnung zu den verteilten Objekten. Schließlich arbeitet die Gossiping-Methode nur auf den zu einem verteilten Objekt gehörenden PEs. Somit können als `DistrObj` formulierte Algorithmen sicher in einem anderen Zusammenhang verwendet werden, eine wichtige Anforderung an Software.

3.3.2.2 DoPlex

Die Klasse `DoPlex` dient zur Parallelisierung von `SoPlex` für Parallelrechner mit verteiltem Speicher als ein verteiltes Objekt. Als von `DistrObj` abgeleitete Klasse wird auf jeder zum verteilten Objekt gehörende PE eine lokales `DoPlex`-Objekt instanziiert. Jedes dieser lokalen Objekte verwaltet eine Sub-LP bestehend aus einer Teilmenge der Zeilen oder Spalten des gesamt LPs. Ob eine zeilen- oder spaltenweise Verteilung vorgenommen wird, richtet sich nach der Wahl der Basisdarstellung. Die Verteilung der Vektoren auf die PEs erfolgt gemäß (2.5).

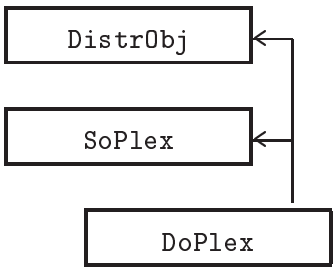


Abbildung 3.14: Parallelisierung von `SoPlex`

Zusätzlich zum jeweiligen Teil-LP verwaltet jedes PE die vollständige Basis samt ihrer Faktorisierung. Dadurch wird jedes PE in die Lage versetzt, lineare Gleichungssysteme mit der Basismatrix zu lösen, wie es für das Block-Pivoting erforderlich ist. Dazu ist es jedoch notwendig, daß jedes PE einen Puffer für Basisvektoren verwaltet, die nicht zum lokalen Teil-LP gehören.

Auf jedem PE wird im Prinzip der in `SoPlex` implementierte Simplex-Algorithmus ausgeführt. Da der sequentielle Code nur das jeweilige Teil-LP sieht, wird so das Matrix-Vektor-Produkt implizit parallel ausgeführt. Das Pricing und der Quotiententest werden ohnehin von algorithmischen Klassen implementiert, deren Parallelisierung im Anschluß diskutiert wird. Demnach müssen von `DoPlex` lediglich die Methoden von `SoPlex` überlagert werden, die sich auf die Verwaltung der Basis beziehen. Wann immer ein auf einem anderen PE befindlicher Vektor in die Basis eintritt, muß dieser empfangen und in den Puffer einsortiert werden. Entsprechend verschickt dasjenige PE, das den in die Basis eintretenden Vektor verwaltet, diesen an alle anderen PEs des verteilten `DoPlex`-Objektes. Diese Funktionalität wird durch Überlagerung der Zugriffsmethode auf LP-Vektoren implementiert. Damit der Puffer für fremde Basisvektoren nicht überläuft, müssen fremde Vektoren, die die Basis verlassen, aus dem Puffer entfernt werden. Dies wird mittels Überlagerung der Basistausch-Methode implementiert.

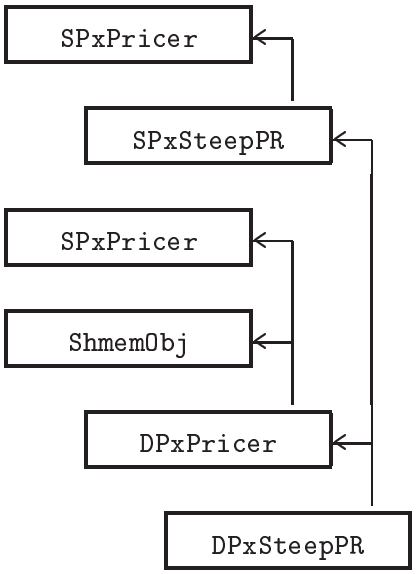


Abbildung 3.15: Parallelisierung des steepest-edge Pricings

Die Implementierung der Pricer- und Quotiententest-Klassen für `DoPlex` gestaltet sich

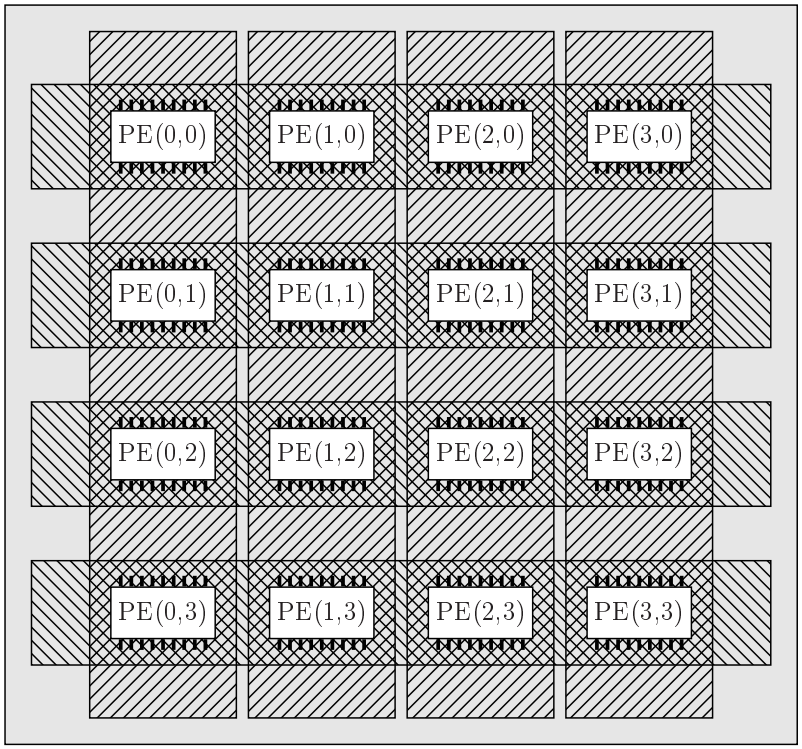


Abbildung 3.16: Beispiel eines 4×4 Prozessorgitters für die parallele LU-Zerlegung. Die PEs des globalen verteilte Objekts sind grau hinterlegt. Entsprechend sind zu zeilen- bzw. spaltenweisen verteilten Objekten zugehörige PEs schraffiert hinterlegt.

---

analog zu dem Vorgehen bei `SMoPlex`, wobei die Klasse `DistrObj` die Rolle von `ShmemObj` übernimmt (vgl. Abbildung 3.15). Dabei wird zur Bestimmung eines maximalen bzw. minimalen Elementes die Gossiping-Methode von `DistrObj` benutzt. Die Klasse `DPxPricer` dient als abstrakte Basisklasse für mit `DoPlex` zu verwendende Pricing-Klassen. Sie implementiert wieder das Block-Pivoting. Entsprechend wird eine Klasse `DPxRatioTester` für die Quotiententest-Klassen definiert. Ferner wird in einer Klasse `DSLinSolver` eine Gleichungssystemlöser unter Nutzung von `SLUFactor` implementiert, der die parallele Lösung zweier Gleichungssysteme mit derselben Matrix unterstützt. Wichtig bei dem Einsatz dieser Klassen mit `DoPlex` ist, daß alle zugehörigen verteilten Objekte physikalisch auf derselben Menge von PEs instantiiert werden.

Die Klasse `DoPlex` und die zugehörigen parallelen Pricing- oder Quotiententest-Klassen wurden von `DistrObj` abgeleitet. Dies erlaubt es, mehrere LP-Solver `DoPlex` auf verschiedenen Teilmengen der PEs eines Parallerechners zu instanziieren, was besonders für die Realisierung von Branch-and-Cut-Algorithmen geeignet ist.

3.3.2.3 Parallele LU-Zerlegung

Bei der parallelen LU-Zerlegung für dünnbesetzte Matrizen nach Abschnitt 2.3 werden die Prozessoren logisch als ein Gitter angeordnet. Für unterschiedliche Operationen kooperieren jeweils unterschiedliche Gruppen von Prozessoren. So berechnen jeweils die Prozessoren einer Zeile die globalen Anzahlen der NNEs pro Zeile sowie ihren jeweils größten Absolutbetrag, und die Prozessoren einer Spalte bestimmen die Anzahl NNEs für jede Spalte der Matrix. Alle Prozessoren gemeinsam bestimmen die Menge der Pivot-Kandidaten und die Liste der Inkompatiblilitäten.

All diese Operationen können als Gossiping-Operationen implementiert werden. Dies wird von der Klasse `DistrObj` unterstützt, weshalb die PEs in verschiedene verteilte Objekte unterteilt werden. Alle PEs zusammen bilden das globale verteilte Objekt der LU-Zerlegung. Zusätzlich werden alle PEs einer Prozessor-Zeile oder -Spalte in jeweils einem verteilten (Sub-)Objekt zusammengefaßt (vgl. Abb. 3.16). Dadurch kann jede Operation ohne viel Verwaltungsaufwand in dem verteilten Objekt implementiert werden, dessen Verteilungschema der Operation entspricht.

Kapitel 4

Ergebnisse

In diesem Kapitel werden die Implementierungen anhand von Probeläufen auf ihren Nutzen zur Lösung von LP-Problemen hin untersucht. Die dazu herangezogenen LPs werden Abschnitt 4.1 vorgestellt. In den folgenden drei Abschnitten werden die jeweiligen Laufzeiten für die drei Implementierungen SoPlex, SMoPlex und DoPlex angegeben. Dabei werden auch die Auswirkungen verschiedener algorithmischer Parameter aufgezeigt. Der letzte Abschnitt steht etwas außerhalb dieser Betrachtungen, da er die parallele Lösung von Gleichungssystemen behandelt, die nicht in die parallelen Implementierungen DoPlex und SMoPlex aufgenommen wurden.

Die folgende Tabelle gibt einen Eindruck über den Umfang des Codes samt Dokumentation:

	SoPlex	SMoPlex	DoPlex
Zeilen	52000	+3900	+11300
Umfang	1.2MB	95kB	133kB

Man erkennt, daß der objektorientierte Entwurf tatsächlich die Mehrfachverwendung von Code unterstützt. Außerdem wird deutlich, daß die Parallelisierung für einen Parallelrechner mit gemeinsamem Speicher weniger aufwendig ist.

4.1 Die Testprobleme

In Tabelle A.1 sind die für die Testläufe herangezogenen LPs zusammengestellt und zwar jeweils mit einigen statistische Werten. Die Anzahl der Zeilen oder Spalten bestimmt je nach Wahl der Basisdarstellung die Dimension der Basismatrix. Die Anzahl der NNEs in der Nebenbedingungsmatrix bestimmt in etwa den Rechenaufwand für das Matrix-Vektor-Produkt bei der Berechung des Vektors Δg. Die Kondition der optimalen Basismatrix gibt schließlich einen Anhaltspunkt für die numerischen Schwierigkeiten, die bei der Lösung des

jeweiligen LPs zu erwarten sind. Sie wurde jeweils mit CPLEX berechnet und bezieht sich auf die optimale Spaltenbasis des *skalierten* LPs.

Die Testmenge der LPs besteht aus Problemen aus der Netlib [107] und verschiedenen am ZIB behandelten Projekten [106]. Aus der Netlib stammen die LPs 1-11. Die meisten wurden in die Testmenge aufgenommen, weil von ihnen die Problembeispiele 12-20 abgeleitet wurden. Dies gilt jedoch nicht für „stocfor3“, das LP mit den meisten Zeilen und Spalten aus der Netlib, „pilots“, ein numerisch besonders schwieriges Netlib-Problem, und „fit2d“, das das Problem mit den meisten Nicht-Null-Elementen der Netlib ist.

Die Probleme 12-20 mit der Endung „.ob4“ sind LP-Relaxierungen eines verallgemeinerten Packing-Problems. Sie stammen aus dem Wurzelknoten eines Branch-and-Cut Verfahrens zur optimalen Dekomposition von Matrizen in unabhängige Blöcke [19]. Sie enthalten bereits Schnittebenen und haben mehr Zeilen als Spalten. Die Namen vor „.ob4“ bezeichnen den Namen des Netlib-Problems, dessen optimale Basismatrix dekomponiert werden soll; bei „agg3.ob4“ also die optimale Basis-Matrix von „agg3“.

Beim Frequenzzuweisungsproblem im Mobilfunk geht es darum, Basisstationen Frequenzen so zuzuweisen, daß sich möglichst wenig Interferenzen ergeben, wobei gewisse technologische und rechtliche Einschränkungen berücksichtigt werden müssen. Die LPs „SM-50.k-68a“ und „SM-50.k-68b“ sind LP-Relaxierungen, die im Rahmen eines Branch-and-Cut Verfahrens zur Lösung dieses Problems entstehen. Sie enthalten reale Daten des Mobilfunk-Anbieters E-Plus.

Mit „kamin1807“ und „kamin2702“ wurden zwei LPs bezeichnet, die einem Projekt zur Kommissionierung von Glückwunschkarten entstammen. Sie enthalten reale Daten von dem Kooperationspartner Herlitz AG.

Bei den Problemen „chr15c“ und „scr12“ handelt es sich um LP-Relaxierungen von quadratic assignment Problemen aus der QAPLIB [23, 81].

Das Problem „stolle“ hat seinen Ursprung in der Planung von Netzwerken. Es handelt sich um ein Steinerbaumproblem mit Längenrestriktionen und Kosten auf den Pfaden. Auch für diese Problem stammen die Daten aus einer realen Anwendung.

Einem Projekt zur optimalen Fahrzeugumlaufplanung entstammen die LPs „hansecom2“ und „hansecom17“ [53]. Die beiden Probleminstanzen entstehen bei der Ausführung eines column-generation Verfahrens. Sie enthalten Daten aus realen Anwendung der Hamburger Hochbahn AG.

Die LPs 30-35 stammen von set-partitioning Problemen etwa zur Einsatzplanung von Crews bei großen Flugunternehmen (aa = American Airlines). Sie wurden bereits in [13] benutzt, um die Parallelisierung des dualen Simplex-Verfahrens zu bewerten.

Die Auswahl der LPs ist etwas voreingenommen, da häufiger als üblich LPs mit mehr Zeilen als Spalten vorkommen. In diesem Fall sollte sich die Verwendung einer Zeilenbasis auszahlen, was auch zu beobachten ist. Der Grund dafür, daß „typische“ LPs eher mehr Spalten als Zeilen aufweisen, ist jedoch auch darin zu sehen, bis dato keine Implementie-

rungen verfügbar waren, die eine Zeilenbasis nutzen. Gute LP-Modellierer haben daher bereits bei der Modellierung darauf geachtet, daß die LPs mehr Spalten als Zeilen haben, notfalls durch den Übergang zu einer dualen Formulierung. Durch Einführung der Zeilenbasis in state-of-the-art Codes ist es jedoch wahrscheinlich, daß zukünftig häufiger LP-Modelle entstehen, die mehr Zeilen als Spalten haben.

Tabelle A.2 enthält die Laufzeiten und die Anzahl der Iterationen die CPLEX 4.07 für die Lösung der Test-LPs benötigt, und zwar sowohl mit dem primalen als auch mit dem dualen Algorithmus. CPLEX wurde jeweils mit den Voreinstellungen verwendet. Da diese Implementierung sicherlich als state-of-the-art angesehen werden darf, dient diese Tabelle als Referenz für die Bewertung von SoPlex.

Für eine bessere Vergleichbarkeit seien nun Informationen über CPLEX zusammengefaßt, soweit diese öffentlich verfügbar sind [15, 16, 13]. Alle LPs werden vor ihrer Lösung skaliert, indem zunächst jede Spalte durch den größten darin auftretenden Absolutbetrag geteilt und anschließend jede Zeile entsprechend behandelt wird. CPLEX verwendet ausschließlich eine Spaltenbasis. Somit ist der duale Algorithmus stets ein entfernender und der primale ein einfügender. Als Pricing-Strategie für den dualen Algorithmus wird das steepest-edge Pricing eingesetzt, wobei die initialen Normen nicht exakt bestimmt sondern auf 1 gesetzt werden. Beim primalen Algorithmus kommt ein dynamisches Pricing, das zwischen partial multiple Pricing und Devex Pricing umschaltet, zum Einsatz. Dabei sind jedoch keine weiteren Informationen über die Details beider Pricing-Verfahren oder der Umschaltstrategie bekannt. Für Phase 1 wird eine Implementierung der Composite Simplex Methode (vgl. (1.95)) verwendet. Die von SoPlex benutzte Crashbasis ist der von CPLEX sehr ähnlich [15]. Dies wird besonders an den Test-Problemen deutlich, bei denen sowohl CPLEX als auch SoPlex direkt mit Phase 2 beginnen können. Diese Fälle eignen sich besonders für eine vergleichende Beurteilung beider Implementierungen.

4.2 SoPlex

In den ersten Abschnitten 4.2.1 und 4.2.2 werden Testläufe von SoPlex mit einer Spaltenbasis betrachtet, um so eine besseren Vergleich mit CPLEX zu ermöglichen. Da sich beide Implementierungen in der Art der Behandlung von Phase 1 unterscheiden (vgl. Abschnitt 1.4), werden zunächst nur die Probleme betrachtet betrachtet, bei denen aufgrund der Startbasis keine Phase 1 benötigt wird.

Die sich anschließenden Abschnitte heben die Besonderheiten von SoPlex gegenüber anderen Implementierungen hervor. Als erstes ist in diesem Zusammenhang die Zeilenbasis zu nennen, deren Nutzen in Abschnitt 4.2.3 bewertet wird. Anschließend wird in Abschnitt 4.2.4 der Effekt der dynamischen Faktorisierungsfrequenz aufgezeigt. Der Vorteil des Gleichungssystemlösers für besonders dünnbesetzte Matrizen wird in Abschnitt 4.2.5 aufgezeigt.

In den darauf folgenden Abschnitten werden die Auswirkungen verschiedener weiterer

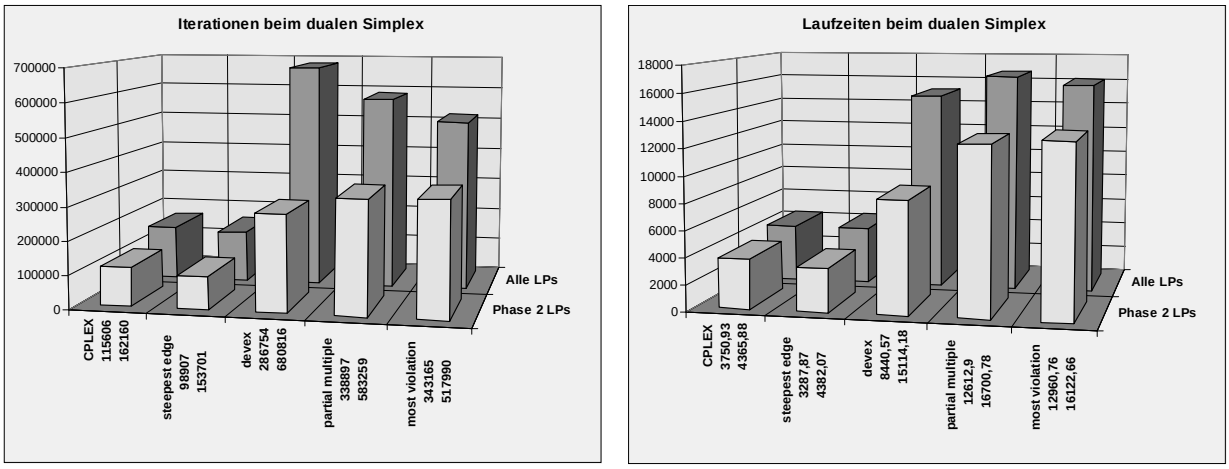


Abbildung 4.1: Summe der Iterationen und Laufzeiten beim dualen Simplex über Mengen von Test-LPs. Bei den mit „Phase 2 LPs“ bezeichneten Werten wurden die Summen über alle LPs der Testmenge gebildet, die keine Phase 1 benötigen. Die mit „alle LPs“ bezeichneten Werte enthalten die Zahlen zu allen LPs, sofern diese verfügbar waren.

---

algorithmischer Parameter untersucht. Es beginnt in Abschnitt 4.2.6 mit dem Einfluß der Skalierung auf die Anzahl der Iterationen. Als nächstes wird in Abschnitt 4.2.7 der Effekt der Crashbasis gegenüber einer Schlupfbasis studiert, und in Abschnitt 4.2.8 folgt eine Untersuchung der Kreiselvermeidungsstrategie von SoPlex. Schließlich wird in Abschnitt 4.2.9 die Wichtigkeit eines numerisch stabilen Quotiententests unterstrichen.

4.2.1 Duale Algorithmen mit Spaltenbasis

Es wird nun die Auswirkung der Pricing-Strategie auf die Geschwindigkeit von SoPlex untersucht und eine Bewertung im Vergleich zu CPLEX beim dualen Simplex vorgenommen. Die Testergebnisse zu CPLEX sind in Tabelle A.2 aufgelistet. Für den Vergleich mit CPLEX werden zunächst Testläufe betrachtet, bei denen eine größtmögliche Übereinstimmung der algorithmischen Parameter erreicht wird. Es werden also die Ergebnisse für SoPlex mit Spaltenbasis, Crashbasis und skaliertem LP betrachtet. Sie sind in den Tabellen A.3 bis A.6 aufgeführt.

Ein wichtiger Unterschied zwischen CPLEX und SoPlex ist die Behandlung der Phase 1. Um auch hier eine gute Vergleichbarkeit zu erzielen, beschränken wir uns zunächst auf die LPs, bei denen beide Implementierungen keine Phase 1 benötigen, weil die gefundene Startbasis bereits dual zulässig ist. Dabei handelt es sich um 22 Problembeispiele, nämlich die LPs mit den Nummern 3-5, 13-24, 27 und 30-35. Sie werden im folgenden Phase 2 LPs genannt.

Die Summen der Anzahl der Iterationen und der Laufzeiten über diese Probleme für

SoPlex mit verschiedenen Pricing-Strategien sowie für CPLEX sind in den ersten Reihen von Abbildung 4.1 aufgeführt und grafisch dargestellt. Man beachte jedoch, daß solche Summen immer nur statistische Aussagen erlauben, wogegen bei einzelne Probleme die Ergebnisse vollkommen anders ausfallen können.

Beide Implementierungen, CPLEX und SoPlex, starten bei den Phase 2 LPs mit einer Schlupfbasis und benutzen beim steepest-edge Pricing mit 1 initialisierte Normen. Somit sind die zugehörigen Werte gut miteinander vergleichbar. SoPlex benötigt deutlich weniger Iterationen als CPLEX. Dies wirkt sich auch auf die Laufzeit aus, obwohl CPLEX etwas weniger Zeit pro Iteration benötigt (vgl. Abb. 4.2).

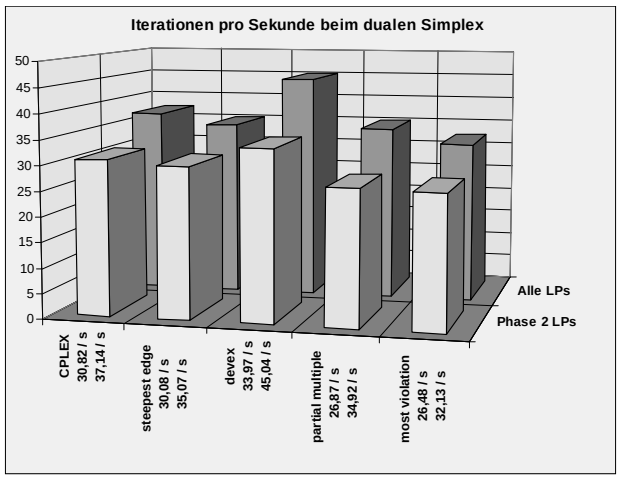


Abbildung 4.2: Durchschnittliche Iterationszahl pro Sekunde beim dualen Simplex für CPLEX und SoPlex mit verschiedenen Pricing-Startegien.

Das Devex-Pricing führt zu einer geringeren Iterationszahl als das partial multiple Pricing. Da letzteres ein Pricing-Verfahren für den einfügenden Simplex ist, wurde für den entfernenden Simplex die most-violation Pricing-Strategie implementiert. Die geringfügigen Unterschiede in den Iterationen und Laufzeiten kommen von den einfügenden Simplex-Iterationen, die zur Behebung eines Shiftes notwendig wurden, sowie aus numerischen Abweichungen.

Eine interessante Beobachtung ist, daß obwohl der Rechenaufwand pro Iteration (bei gleicher Basis) beim steepest-edge und Devex Pricing höher ist als beim most-violation Pricing, letzteres weniger Iterationen pro Sekunde erreicht. Dies liegt an der geringeren Iterationszahl: In den ersten Iterationen ist die Basismatrix fast eine Einheitsmatrix. Dann werden die anfallenden linearen Gleichungssysteme schneller gelöst und der Lösungsvektor hat weniger NNEs als bei einer späteren Basismatrix, die mehr NNEs enthält.

In Abbildung 4.1 finden sich auch die über alle LPs gebildeten Summen der Iterationen und Laufzeiten. Dabei sind jedoch die Werte für das Devex, partial multiple und most-violation Pricing untere Schranken, da bei den Testläufen einige LPs nicht innerhalb des Zeitfensters von 3000 Sekunden gelöst werden konnten (vgl. Anhang A).

Während CPLEX auch in Phase 1 einen entfernenden Algorithmus ausführt, benutzt SoPlex dabei den primalen und somit einfügenden Simplex. Beim steepest-edge Pricing ist der Rechenaufwand pro Iteration beim einfügenden Simplex höher als beim entfernenden, da ein zusätzliches Matrix-Vektor-Produkt berechnet werden muß. Aus diesem Grund übertrifft nun die Laufzeit von SoPlex auch beim steepest-edge Pricing die von CPLEX geringfügig, obwohl die Iterationszahl weiterhin niedriger ausfällt. Dies wird auch in Abbildung 4.2 an der nunmehr höheren Iterationsgeschwindigkeit von CPLEX deutlich.

Wie bei den Phase 2 LPs sind die anderen Pricing-Strategien sowohl von der Iterati-

onszahl als auch von der Laufzeit her weit abgeschlagen. Das Devex Pricing scheint eine höhere Iterationszahl als das most-violation oder partial multiple Pricing zu benötigen. Dies kann jedoch nicht festgestellt werden, da alle drei Pricer einige LPs nicht im vorgegebenen Zeitfenster lösen konnten. Beim Devex Pricing handelt es sich dabei lediglich um ein LP, während mit partial multiple und most-violation Pricing drei bzw. vier LPs nicht innerhalb von 3000 s gelöst werden konnten.

Insgesamt erwartet man, daß das Devex Pricing weniger Iterationen benötigt, als das partial multiple oder most-violation Pricing. Dies trifft auch meistens zu. Eklatante Ausnahmen sind die LPs 25 und 26. Offenbar enthalten diese LPs Strukturen, die sich negativ auf das Devex Pricing auswirken. Solche „Überraschungen“ zeigen, daß es sich bei allen Pricing-Strategien nicht um Verfahren mit garantierter Iterationszahl handelt. Man kann also immer Pech — aber auch Glück — haben.

Zusammengefaßt kann folgendes für den dualen Simplex mit Spaltenbasis festgestellt werden:

- Das steepest-edge Pricing ist die erfolgreichste Pricing-Strategie.
- Phase 1 von SoPlex unterliegt aufgrund des höheren Rechenaufwandes beim steepest-edge Pricing für den einfügenden Simplex dem Composite Simplex [100].
- SoPlex übertrifft beim entfernenden Algorithmus CPLEX — jedenfalls bei den zugrundegelegten Test-LPs.

4.2.2 Primale Algorithmen mit Spaltenbasis

Wie im vorigen Abschnitt beschränken wir uns bei der Bewertung des primalen Simplex-Algorithmus zunächst auf die LPs, für die eine primal zulässige Startbasis gefunden und nur der Phase 2 Algorithmus durchgeführt werden muß. Dabei handelt es sich um 11 LPs, nämlich die Probleme mit den Nummern 3, 5 und 12-20. Die über diese LPs gebildeten Summen der Iterationszahlen und Laufzeiten sind im Vordergund von Abbildung 4.3 für CPLEX und SoPlex mit verschiedenen Pricern dargestellt.

Bei den Phase 2 LPs zeigt SoPlex mit steepest-edge Pricing die geringste Interationszahl während CPLEX die geringste Laufzeit aufweist. CPLEX benutzt beim primalen Simplex ein dynamisches Pricing-Verfahren, das zur Laufzeit zwischen partial multiple und Devex Pricing umschaltet. Dabei sind die Umschaltstrategie und die Interna beim partial multiple Pricing nicht bekannt. Um einen Hinweis darauf zu erhalten, wie SoPlex mit einer entsprechenden Pricing-Strategie arbeiten würde, sind mit „MIN“ die Werte für einen hypothetischen semi-dynamischen Pricer aufgeführt, der zu Beginn der Lösung eines LPs die jeweils für ein Problem schnellere Pricing-Strategie zwischen Devex und partial multiple auswählt. Für die Summen wurden also, je nachdem welche Laufzeit niedriger ausfällt,

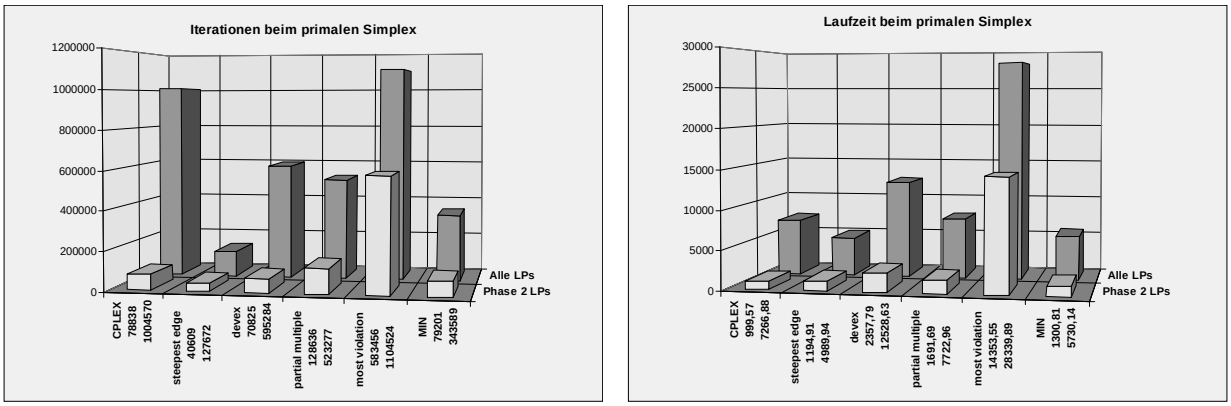


Abbildung 4.3: Summe der Iterationen und Laufzeiten des primalen Simplex über Mengen von Test-LPs. Bei den mit „Phase 2 LPs“ bezeichneten Werten wurden die Summen über alle LPs der Testmenge gebildet, die keine Phase 1 benötigen. Die mit „alle LPs“ bezeichneten Werte enthalten die Zahlen zu allen LPs bis auf die Probleme 27 und 29, sofern diese verfügbar waren. Die beiden LPs wurden weggelassen, da CPLEX sie nicht innerhalb des vorgegebenen Zeitrahmens von 3000 Sekunden lösen konnte.

---

jeweils die Werte vom Devex oder vom partial multiple Pricing benutzt. Ein echter dynamischer Pricer, der die Entscheidung während der Laufzeit trifft, könnte die Laufzeit noch weiter senken.

Per Definition zeigt dieser hypothetische Pricer eine geringer Laufzeit sowohl als das Devex wie auch das partial multiple Pricing. Jedoch erreicht er noch nicht die Geschwindigkeit des Pricing-Verfahrens von CPLEX. Dieses scheint meist ein partial multiple Pricing durchzuführen, wofür die hohe Anzahl von Iterationen pro Sekunde spricht (vgl. Abb. 4.4). Jedoch gelingt es CPLEX, dabei signifikant weniger Iterationen auszuführen als SoPlex mit partial multiple Pricing.

Interessant ist die Beobachtung, daß das most-violation Pricing zu einer etwa fünfmal höheren Iterationszahl führt als das partial multiple Pricing. Eigentlich erwartet man von einem Pricing-Verfahren, das stets das gesamte LP berücksichtigt, eine geringere Iterationszahl als von einem, das immer nur einen Teil betrachtet. Eine plausible Erklärung für dieses Phänomen könnte sein, daß das most-violation Pricing immer wieder denselben Fehler macht, scheinbar „gute“ Pivot-Elemente auszuwählen, die jedoch de facto keinen Fortschritt bringen. Vor solchen Fehlern ist das partial multiple Pricing geschützt, da es solche irreführenden Pivot-Elemente gar nicht erst „sieht“, wenn sie nicht zufällig im aktuellen Teil-LP liegen.

Dieser Erklärungsversuch trifft nicht in allen Fällen zu: Bei den Problemen im vorigen Abschnitt zeigte das partial multiple Pricing eine höhere Anzahl von Iterationen als das most-violation Pricing. Somit liegt es wohl eher an den zugrundeliegenden LPs, welche Pricing-Strategie am erfolgreichsten ist. Dies gilt auch für das steepest-edge Pricing. Auch wenn dafür die akkumulierte Laufzeit bei SoPlex am geringsten ausfällt, gilt dies nicht

bei allen LPs. Ein besonders eklatantes Beispiel hierfür ist das Problem 3 (fit2d): Mit steepest-edge Pricing benötigt SoPlex für dessen Lösung 345,74 s, mit partial multiple Pricing lediglich 21,02 s.

Insgesamt zeigen diese Ergebnisse und die des vorigen Abschnittes, daß die Pricing-Strategie einen extremen Einfluß auf die Lösungsgeschwindigkeit eines LPs hat. Dabei gibt es jedoch keine allgemeine „optimale“ Pricing-Strategie. Vielmehr sollte für jede Problemklasse nach einer optimalen Strategie gesucht werden. Mit seinem objektorientierten Software-Entwurf bietet SoPlex hierfür optimale Voraussetzungen.

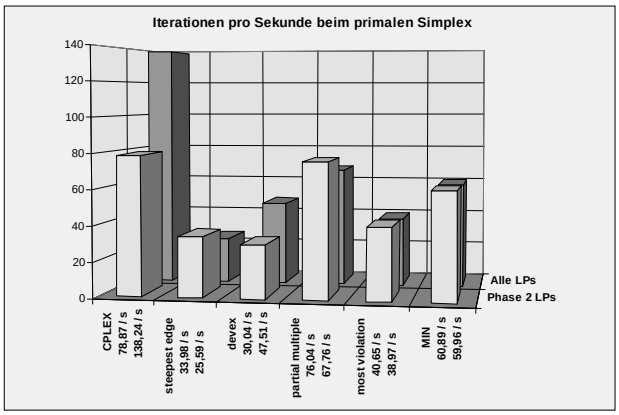


Abbildung 4.4: Durchschnittliche Iterationszahl pro Sekunde beim primalen Simplex für CPLEX und SoPlex mit verschiedenen Pricing-Startegien.

Wenden wir uns nun den verbleibenden LPs zu, bei denen auch ein Phase-1 Problem gelöst werden muß. Aus dem Vergleich mit den Werten der Phase 2 LPs können Rückschlüsse auf den Unterschiede bei der Phase 1 zwischen CPLEX und SoPlex für den dualen Simplex gewonnen werden.

Die über alle Test-Beispiele aufsummierten Iterationszahlen und Laufzeiten sind in der zweiten Reihe von Abbildung 4.3 angegeben. Dabei wurden jedoch die Probleme 27 und 29 weggelassen, da CPLEX sie nicht in dem vorgegebenen Zeitrahmen von 3000 Sekunden lösen konnte. Auch sind die Werte für das Devex und most-violation Pricing lediglich untere Schranken, da einige weitere LPs nicht innerhalb der Zeitbeschränkung gelöst werden konnten. Diese Probleme wurden jedoch nicht aus der Summenbildung ausgeschlossen, um die Anzahl der Problembeispiele nicht noch weiter zu verringern. Eine aussagekräftige Bewertung der Zahlen ist auch so möglich.

Die wichtigste Beobachtung ist, daß sich das Verhältnis der Iterationszahlen gegenüber den Phase 2 LPs signifikant zugunsten von SoPlex verändert hat. Somit zahlt sich aus, daß SoPlex bei seinem Phase-1 Problem soviel Information des Ausgangs-LPs beibehält wie möglich. Insbesondere liegt bei einigen Problemen (Tabelle A.3, Probleme: 2,21,22,23,24,34) die Lösung des Ausgangs-LPs bereits vor, nachdem die Phase 1 abgeschlossen ist. CPLEX gelingt dies nur in einem Fall (Tabelle A.2, Problem 24).

Durch die geringere Iterationszahl löst SoPlex mit steepest-edge Pricing alle LPs in deutlich kürzerer Zeit als CPLEX. Dabei wirkt sich zusätzlich die Tatsache positiv aus, daß in der Phase 1 eine entfernender Simplex benutzt wird, bei dem das steepest-edge Pricing weniger Rechenaufwand benötigt als im einfügenden Fall.

Die Phase 1 von SoPlex scheint für den primalen Simplex also günstiger als eine Implementierung nach [100]. Wieder ist dies keine beweisbare Aussage, sondern spiegelt lediglich die Ergebnisse bei der gewählten Testmenge wider. Es wird immer Probleme geben, bei

denen sich ein gegenteiliger Effekt zeigt (z.B. Problem 8).

Sowohl mit Devex Pricing als auch mit partial multiple Pricing bleibt SoPlex hinter der Geschwindigkeit von CPLEX zurück. Der hypothetische Pricer „MIN“ könnte hingegen CPLEX überholen, ohne jedoch die Geschwindigkeit des steepest-edge Pricing zu erreichen.

Aufgrund der hohen Iterationszahl pro Sekunde ist anzunehmen, das CPLEX vorwiegend ein partial multiple Pricing verwendet. Daß die Iterationsgeschwindigkeit beim partial multiple Pricing von SoPlex deutlich niedrieger ausfällt, liegt wieder daran, daß in Phase 1 ein entfernender Algorithmus ausgeführt wird, bei dem im partial multiple Pricer eine most-violation Pricing-Strategie implementiert ist.

4.2.3 Zeilen- versus Spaltenbasis

Die Untersuchung der Abhängigkeit der Simplex-Algorithmen von der gewählten Basisdarstellung geschieht anhand von Abbildung 4.5. Jeder Punkt beschreibt das Verhältnis von Zeit, Iterationszahl oder Iterationszeit bei der Lösung eines LPs mit einer Zeilenbasis gegenüber der Lösung mit einer Spaltenbasis. Dazu wurden die Werte aus Tabelle A.7 zu denen aus A.3 in Beziehung gesetzt, wobei sowohl die Ergebnisse vom primalen als auch vom dualen Algorithmus berücksichtigt wurden.

Als Abszisse ist der Logarithmus (zur Basis 10) des Verhältnisses der Anzahl von Zeilen zur Anzahl von Spalten jedes LPs aufgetragen: log(Zeilen / Spalten). Somit repräsentieren negative Werte LPs mit mehr Spalten als Zeilen und positive solche mit mehr Zeilen als Spalten.

Die Ordinate dient dem Vergleich der Laufzeit, Iterationszahl oder Iterationszeit (Zeit pro Iteration) zur Lösung eines LPs bei der Verwendung eine Zeilen- oder einer Spaltenbasis: Bezeichne z.B. mit t_z und t_s die Lösungszeit eines LPs für eine Zeilen- bzw. Spaltenbasis, so wird als Ordinate $\log(t_z/t_s)$ aufgetragen. Ein Punkt überhalb der Achse beschreibt also eine Lösung, bei der die Verwendung der Zeilenbasis mehr Zeit benötigt als bei Verwendung der Spaltenbasis. Entsprechend wurden die Iterationszahlen und die Iterationszeit aufgetragen.

Für die Iterationszeit wurde eine Ausgleichsgerade bestimmt und in Abbildung 4.5 eingezeichnet. Sie zeigt deutlich den zu erwartenden Zusammenhang zwischen der vorzuziehenden Basisdarstellung und der Gestalt des LPs: Die Darstellung sollte so gewählt werden, daß die Dimension der Basismatrix geringer ausfällt.

Für die Iterationszahl und Lösungszeit wurden die Ausgleichsgeraden nicht eingetragen, da sie aufgrund der Streuung der Datenpunkte im Gegensatz zur Iterationszeit nur sehr ungenau bestimmt sind. Daher ist es auch nicht verwunderlich, daß sich die höhere Iterationsgeschwindigkeit bei der Wahl der geeigneteren Basisdarstellung nicht immer bei der Gesamtlaufzeit auszahlt. Insbesondere für den dualen Simplex bei den Problemen 12-20 kommt es zu einer drastischen Erhöhung der Iterationszahl bei Verwendung der Zeilenbasis,

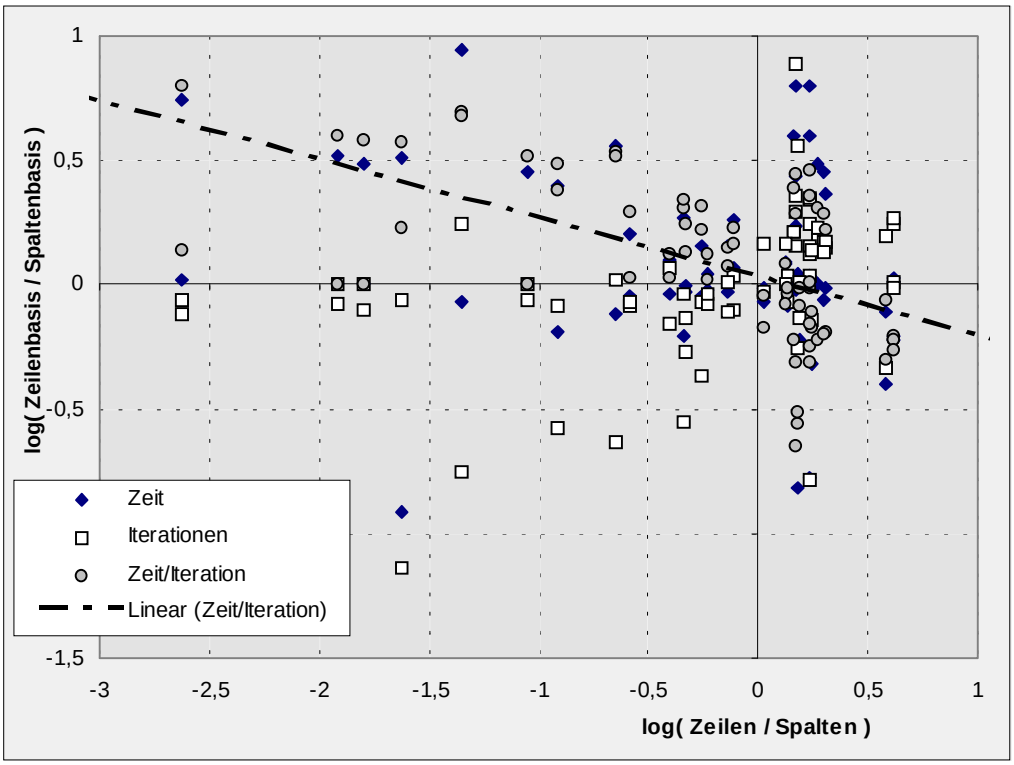


Abbildung 4.5: Logarithmisches Verhältnis der Iterationszahl, Lösungszeit und Iterationszeit für SoPlex mit Zeilenbasis gegenüber Spaltenbasis. Alle Werte wurden mit steepest-edge Pricing bestimmt

---

wodurch bei diesen Problemen die Laufzeiten höher als mit einer Spaltenbasis ausfallen.

Unterschiede bei der Iterationszahl kommen zustande, wenn beim Pricing „gleich gute“ Pivot-Elemente zur Verfügung stehen. Durch die unterschiedliche Traversierung der Indizes in den Schleifen bei Verwendung einer Zeilen- oder Spaltenbasis kommt es zu einer unterschiedlichen Auswahl und somit zu differierenden Iterationszahlen. Diese Unterschiede können sich positiv für den einen oder anderen Fall auswirken; im Mittel ist jedoch keine Präferenz zu erwarten.

Ein wichtiger Aspekt bei der Wahl der Basisdarstellung wird von Abbildung 4.5 nicht beschrieben, nämlich ihre Auswirkung auf die verwendbaren Pricing-Strategien. Soll ein LP mit mehr Ungleichungen als Variablen mit dem dualen Simplex gelöst werden, so wird bei einer Zeilenbasis ein einfügender Algorithmus benutzt. Für das steepest-edge Pricing bedeutet dies einen erhöhten Rechenaufwand pro Iteration, weshalb es günstiger sein kann, dennoch eine Spaltenbasis zu verwenden.

Die Zeilenbasis ermöglicht in diesem Zusammenhang auch die Verwendung anderer Pricing-Verfahren beim dualen Simplex, insbesondere das partial multiple Pricing. Dies kann zu bemerkenswerten Leistungssteigerungen verhelfen. Beispiele hierfür, die einzeln gerechnet wurden und somit nicht aus den Tabellen im Anhang entnommen werden können,

sind (dualer Simplex mit Zeilenbasis):

Problem	steepest-edge	partial multiple
5	12,92 s	10,45 s
11	203,02 s	108,93 s
27	530,83 s	186,42 s
34	2147,42 s	909,41 s

4.2.4 Dynamische Refaktorisierung

Im Gegensatz zu [64] darf sich zur Bewertung des Erfolges der dynamischen Faktorisierungsfrequenz der Vergleich nicht auf eine einzelne Faktorisierungsfrequenz beschränken, sondern muß sich an dem „Optimum“ für eine konstante Faktorisierungsfrequenz messen. Dazu wurden die Zeiten pro 100 Iterationen für eine Reihe fester Faktorisierungsfrequenzen zwischen 10 und 320 gemessen. Sie sind im einzelnen in Tabelle A.12 aufgeführt und die Summen in Abbildung 4.6 dargestellt.

Auch für die dynamische Faktorisierungsfrequenz wurde eine Testreihe über verschiedene Werte von $\eta\phi$ zwischen 1 und 32 durchgeführt. Die ermittelten Iterationszeiten stehen in Tabelle A.11. Die mittleren Geschwindigkeiten wurden mit in Abbildung 4.6 aufgenommen, obwohl die Werte $\eta\phi$ nicht mit den statischen Faktorisierungsfrequenzen vergleichbar sind. Dennoch bietet die gewählte Darstellung eine gute Gegenüberstellung beider Ansätze.

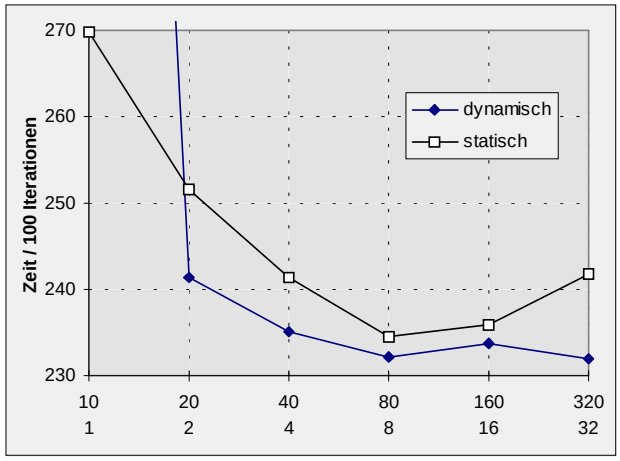


Abbildung 4.6: Summe der Iterationsgeschwindigkeit über alle Testprobleme für dynamische und statische Faktorisierungsfrequenz der Basis-Matrix nach den Tabellen A.11 und A.12.

Für die statischen Faktorisierungsfrequenz ist ein eindeutiges Minimum der Zeit pro Iteration bei einer Faktorisierung der Basismatrix alle 80 Iterationen festzustellen. Bei seltenerer oder häufigerer Faktorisierung sinkt die Iterationsgeschwindigkeit, weil im ersten Fall der Rechenaufwand für die Faktorisierung zunimmt und im zweiten Fall der Rechenaufwand für die Lösung der Gleichungssysteme steigt.

Dagegen zeigt die dynamische Einstellung der Faktorisierungsfrequenz eine deutlich geringere Abhängigkeit von dem Einstellparameter $\eta\phi$, sofern er größer als etwa 6 gewählt wird. Darunter kommt es zu einem steilen Anstieg der Zeit pro Iteration, da die Basismatrix zu häufig faktorisiert wird. Für $\eta\phi \geq 6$ ist die Iterationsgeschwindigkeit höher als die höchste mit fester Faktorisierungsfrequenz erreichbare.

Schließlich wäre eine Heuristik denkbar, die vor Beginn des Simplex-Algorithmus

z.B. anhand der Anzahl von Zeilen und Spalten des zu lösenden LPs den optimalen Parameter für die statische oder dynamische Faktorisierungsfrequenz einstellt. Wir simulieren dies, indem wir die mittlere Iterationszeit über alle LPs bestimmen, wobei für jedes LP der jeweils optimale Parameter Wert benutzt wird. Dann erhält man als mittlere Laufzeit pro 100 Iterationen 230,5 für den statischen und 227,5 für den dynamischen Fall, d.h. die dynamische Refaktorisierungsfrequenz führt auch dann noch zu einer höheren Iterationsgeschwindigkeit.

4.2.5 Gleichungssystemlöser

Es sei nun auf ein einzelnes Problem hingewiesen, bei dem SoPlex von der effizienten Implementierung des Lösers für gestaffelte Gleichungssysteme für besonders dünnbesetzte Vektoren profitiert. Es handelt sich um das Problem 27 (stolle), das im Schnitt 4,9 NNEs pro Spalte enthält. Insgesamt sind weniger als 0.004% der Matrixelemente seiner Nebenbedingungsmatrix von Null verschieden. Die optimale Spaltenbasis-Matrix hat durchschnittlich sogar nur 1,4 NNEs pro Spalte.

Für derart dünnbesetzte Matrizen lohnt es sich bei der Lösung der gestaffelten Gleichungssysteme über die Positionen der NNEs im Arbeitsvektor Buch zu führen, wie es in Abschnitt 1.7.4 beschrieben wurde. Dies zeigt ein Vergleich der Laufzeiten beim dualen Simplex zwischen CPLEX und SoPlex mit Spaltenbasis. Da keine Phase 1 nötig ist, führen CPLEX und SoPlex mit steepest-edge Pricing denselben Algorithmus aus. Nun ist zwar die Iterationszahl bei SoPlex deutlich höher, die Laufzeit hingegen wesentlich geringer (437,68s vs. 735,19s).

4.2.6 Skalierung

Die Auswirkungen der Skalierung des LPs können durch Vergleich von Tabelle A.3 mit A.8 untersucht werden. Oft stimmen Iterationszahlen und Laufzeiten mit und ohne Skalierung überein, jedoch gibt es auch Probleme, bei der die Skalierung einen großen Einfluß hat. Sie kann sich sowohl positiv als auch negativ auf die Lösungszeit niederschlagen. Beispiele für eine Verschlechterung durch Skalierung sind die Probleme 3 und 27, während sie sich z.B. bei den Problemen 8 und 14 positiv auswirkt. Für einzelne Probleme ist es somit nicht möglich, eine Vorhersage über den Nutzen der Skalierung des LPs zu machen.

Eine statistische Aussage gewinnen wir wiederum durch Summenbildung. Dazu sind in der folgenden Tabelle die Summen der Iterationen über alle Testläufe aufgeführt:

	primaler Simplex	dualer Simplex
ohne Skalierung	149943	154784
mit Skalierung	165922	153701
Verhältnis	110,7%	99,3%

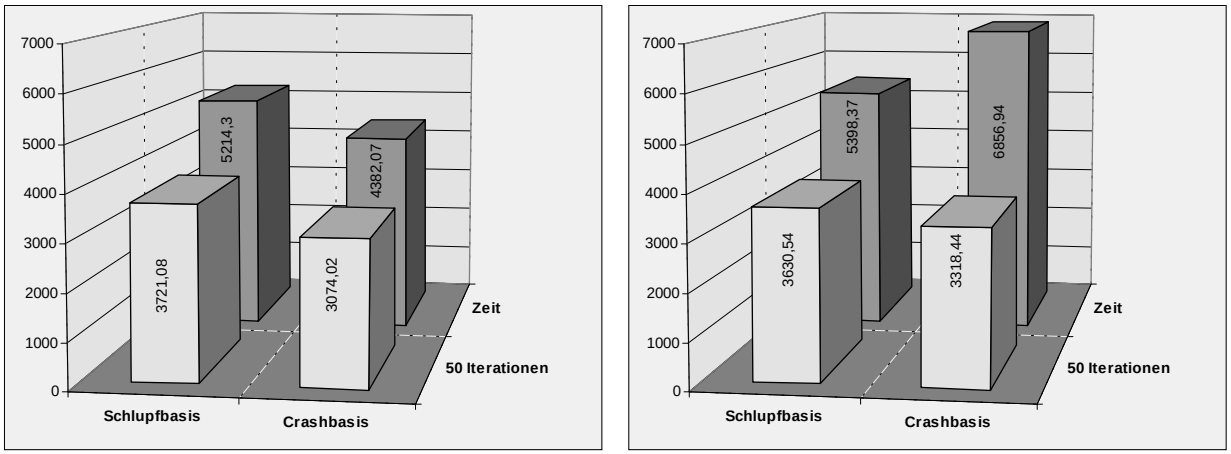


Abbildung 4.7: Laufzeit und Iterationszahl/50 beim dualen (links) und primalen (rechts) Simplex mit steepest-edge Pricing für eine Crash- und eine Schlupfbasis.

Beim primalen Simplex führt das Skalieren des LPs somit im Schnitt zu 10% mehr Iterationen, während es beim dualen Simplex kein signifikanter Einfluß ausgemacht werden kann.

4.2.7 Die Startbasis

In Tabelle A.9 sind die Laufzeiten und Iterationszahlen für die Lösung der Test-LPs bei Verwendung einer Schlupfbasis aufgelistet. Diese Startbasis ist für die Probleme mit den Nummern 3-5, 12-24 sowie 27-35 dual zulässig. Aufgrund der Phase 1 von SoPlex unterscheiden sich (bis auf numerische Verschiedenheiten) der primale und duale Algorithmus für diese Probleme nicht. In der Sprache des primalen Simplex bedeutet dies, daß die LPs bereits in der Phase 1 gelöst wurden.

Zur statistischen Bewertung des Nutzen der Crashbasis gegenüber einer Schlupfbasis wurden wiederum die Summen der Laufzeiten und Iterationen über alle LPs bestimmt und in Abbildung 4.7 dargestellt. Sowohl für den primalen als auch für den dualen Simplex führt die Crashbasis zu einer geringeren Iterationszahl als die Schlupfbasis, wobei der Unterschied beim dualen Simplex höher ausfällt.

Auf die Laufzeit wirkt sich die Reduktion der Iterationszahl jedoch nur beim dualen Simplex aus. Beim primalen Algorithmus fällt die akkumulierte Lösungszeit trotz höherer Iterationszahl für die Schlupfbasis geringer aus als bei der Crashbasis. Der Grund dafür ist wiederum beim steepest-edge Pricing zu suchen. Bei vielen Probleme ist die Schlupfbasis dual zulässig, so daß SoPlex sie bereits mit Phase 1 vollständig löst. Dabei wird jedoch der duale Simplex benutzt. Bei der verwendeten Spaltenbasis ist dies der entfernende Algorithmus, bei dem das steepest-edge Pricing einen geringeren Rechenaufwand

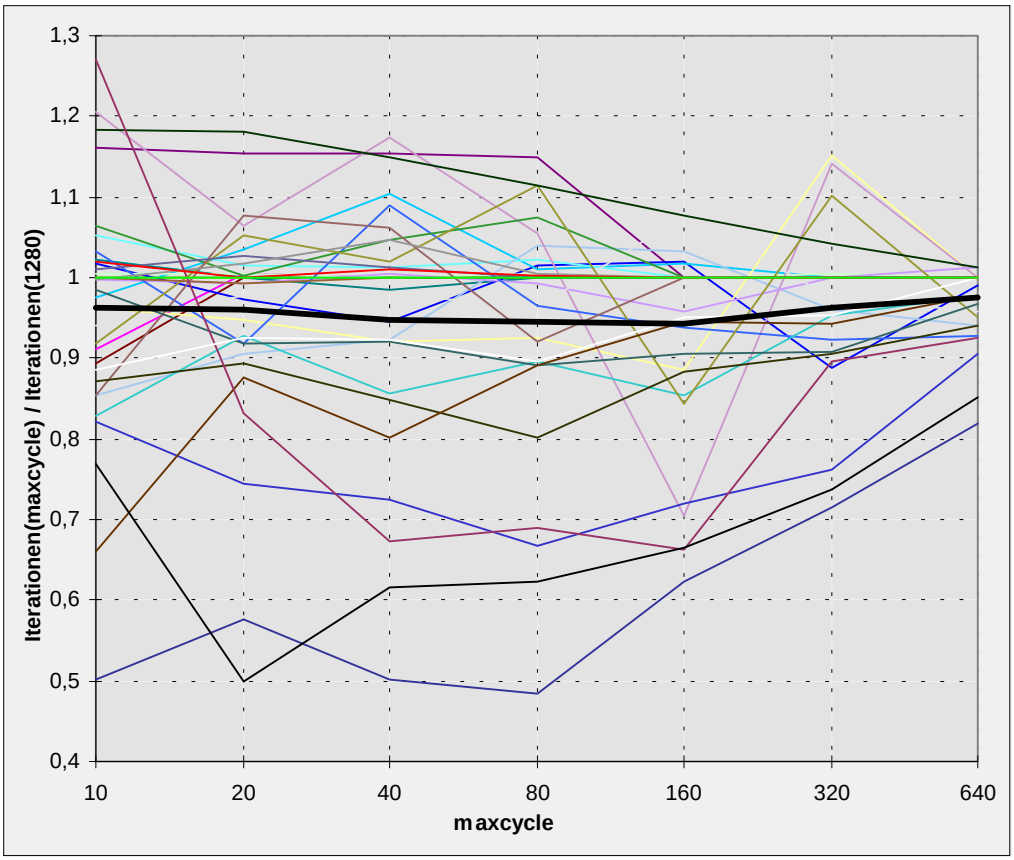


Abbildung 4.8: Iterationszahl in Abhängigkeit von der Anzahl `maxcycle` degenerierter Pivot-Schritten bis zur Perturbation des LPs, gemessen beim primalen Simplex mit steepest-edge Pricing. Die dick gezeichnete Linie repräsentiert die Summe über alle Testprobleme.

---

pro Iteration benötigt als beim einfügenden. Mit der Crashbasis werden zwar insgesamt weniger Iterationen benötigt, jedoch ist der Anteil von einfügenden Iterationen höher.

4.2.8 Kreiseln

Um den Effekt der Kreiselvermeidungsstrategie von SoPlex zu beurteilen, wurden die Iterationszahlen für verschiedene Parameter `maxcycle` (vgl. Abschnitt 1.5) gemessen (Tabelle A.10) und in Abbildung 4.8 dargestellt.

An der Summe über alle Probleme (fettgedruckte Linie) ist zu erkennen, daß sich sowohl zu häufiges als auch zu seltenes Perturbieren im Mittel negativ auf die Iterationszahl auswirkt. Bei zu häufiger Perturbation wird nach Beenden einer Phase ein stark verändertes LP vorliegen, so daß für die Lösung des Ausgangs-LPs eine weitere Phase mit den dafür nötigen Simplex-Iterationen erforderlich wird. Entsprechend können bei zu seltenem

Aufbrechen der Degeneriertheit lange Pivot-Sequenzen ohne Fortschritt entstehen. Das Optimum liegt für die gegebenen Testmenge bei etwa `maxcycle` = 160 Iterationen.

4.2.9 Der Quotientest

In den Tabellen A.13 und A.14 sind die Test-Ergebnisse von SoPlex bei Verwendung des Quotiententests nach Harris bzw. des Textbook-Quotiententests verzeichnet. Dabei sind die Ergebnisse im einzelnen von geringer Bedeutung. Vielmehr soll deutlich werden, daß in beiden Fällen nicht alle LPs gelöst werden konnten.

In den Fällen, die von beide Versionen mit gleicher Iterationszahl korrekt gelöst werden, stellt sich der Textbook-Quotiententest als der effizientere heraus. Dies ist aufgrund des geringeren Rechenaufwandes im Vergleich zum zweiphasigen Harris Verfahren auch zu erwarten. In diesem Zusammenhang ist die Beobachtung interessant, daß bei gleicher Iterationszahl der Textbook-Quotiententest nur selten effizienter als der von SoPlex ist (Ausnahmen bei den Problemen 11 und 23). Dies liegt am Weglassen von Phase 2, sofern in Phase 1 bereits ein stabiles Pivot-Element gefunden wurde.

Es ist jedoch zu bemerken, daß die drei Implementierungen des Quotiententests nicht wirklich miteinander vergleichbar sind. Es ist zu erwarten, daß die Stabilität auch für den Harris und den Textbook-Quotiententest durch eine dynamische Toleranzanpassung und das Zurückweisen von Pivot-Elementen (vgl. Abschnitt 1.8.2) weiter gesteigert werden kann. Dergleichen wird vermutlich auch bei CPLEX vorgenommen, dessen Quotiententest auf dem Ansatz von Harris basiert [15, 16].

4.3 SMoPlex

Die in diesem Abschnitt vorgestellten Testergebnisse wurden auf einer SGI Challenge mit 2 MIPS 4000 Prozessoren a 150 MHz durchgeführt. Die benutzte Maschine wird als File-Server genutzt, so daß bessere Ergebnisse auf einem ausschließlich für SMoPlex zur Verfügung stehenden Rechner zu erwarten sind. Ferner zeigt die Hardware Engpässe beim Speicherbus: Die sequentielle Lösung von „nw16“ beansprucht 129.21s, während die zweifache sequentielle Lösung von „nw16“ auf je einem Prozessor jeweils 144.63s dauerte (user time). Somit kann man für einen Parallelrechner, der keinen solchen Engpaß zeigt, eine Verbesserung der Ergebnisse für 2 Prozessoren um etwa 10% erwarten.

Die Basisdarstellung wurde bei allen Testläufen so gewählt, daß die Dimension der Basismatrix geringer ausfällt. Es wurde immer eine Schlupfbasis verwendet und der duale Simplex-Algorithmus ausgeführt. Als Pricing-Strategie wurde das steepest-edge Pricing verwendet, denn nur für dieses kann die parallele Lösung zweier Gleichungssysteme verwendet werden. Alle Zeiten wurden als „wallclock time“ (real verstrichene Zeit) gemessen.

4.3.1 Aufteilungsschema

In Abschnitt 2.2.1 wurde für die Parallelisierung des Matrix-Vektor-Produktes eine blockweise Aufteilung der Arbeit und somit der Nebenbedingungsmatrix vorgeschlagen. Um die optimale Anzahl von Blöcken zu bestimmen, wurden die Laufzeiten und Iterationen für eine Aufteilung in 2, 4, 8 und 16 Blöcke gemessen; sie sind in Tabelle A.15 vezeichnet. Dabei wurden die anderen Parallelisierungsstrategien (Block-Pivoting und das gleichzeitige Lösen zweier linearer Gleichungssysteme) nicht eingesetzt. In Abbildung 4.9 sind nun die daraus gewonnenen Ergebnisse grafisch dargestellt. Es wird für jede Anzahl von Blöcken die mittlere Iterationsgeschwindigkeit als der Quotient aus der Summe aller Iterationszahlen und der Summe der Laufzeiten über alle Probleme aufgetragen. Ersichtlich wird mit der geringsten Blockzahl die höchste Iterationsgeschwindigkeit erreicht. Als Grund dafür wurde bereits in Abschnitt 2.2.1 auf die Dünnbesetztheit des Vektors Δh angeführt. Schließlich ist festzustellen, daß eine Geschwindigkeitssteigerung durch die parallele Berechnung des Matrix-Vektor-Produktes erzielt werden kann.

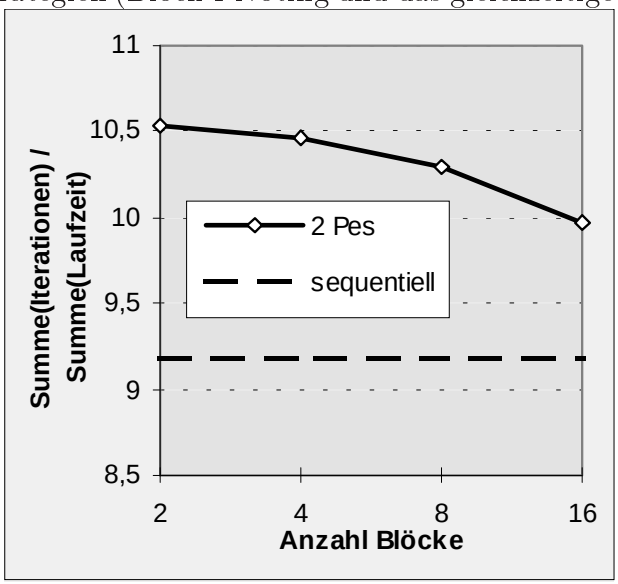


Abbildung 4.9: Durchschnittliche Iterationszahl pro Sekunde in Abhängigkeit von der Anzahl von Blöcke, in die das LP aufgeteilt wird.

4.3.2 Erzielte Beschleunigung

Für die Testläufe zur Bewertung der in SMoPlex eingesetzten Parallelisierungskonzepte wurde aufgrund der oben geschilderten Ergebnisse eine Aufteilung in 2 Blöcke benutzt. Es wurden Testläufe für alle vier möglichen Kombinationen der Verwendung des Block-Pivotings und der parallelen Lösung zweier Gleichungssysteme durchgeführt. Die Meßergebnisse sind in Tabelle A.16 aufgeführt.

Abbildung 4.10 zeigt die Summen der Laufzeiten und der Iterationen sowie das Verhältnis derselben zueinander in bezug auf die entsprechenden Werte bei der Lösung mit nur einen Prozessor. Letztere stammen aus Tabelle A.15.

In bezug auf die Laufzeit kann nur mit den Versionen ohne Block-Pivoting eine Beschleunigung erzielt werden, jedenfalls wenn man alle Testprobleme betrachtet. Eine genauere Untersuchung des Block-Pivoting erfolgt in Abschnitt 4.3.3. Der Grund für das schlechte Abschneiden der parallelen Versionen mit Block-Pivoting liegt in der Erhöhung der Iterationszahl. Während sie ohne Block-Pivoting im Mittel gegenüber dem sequentiel-

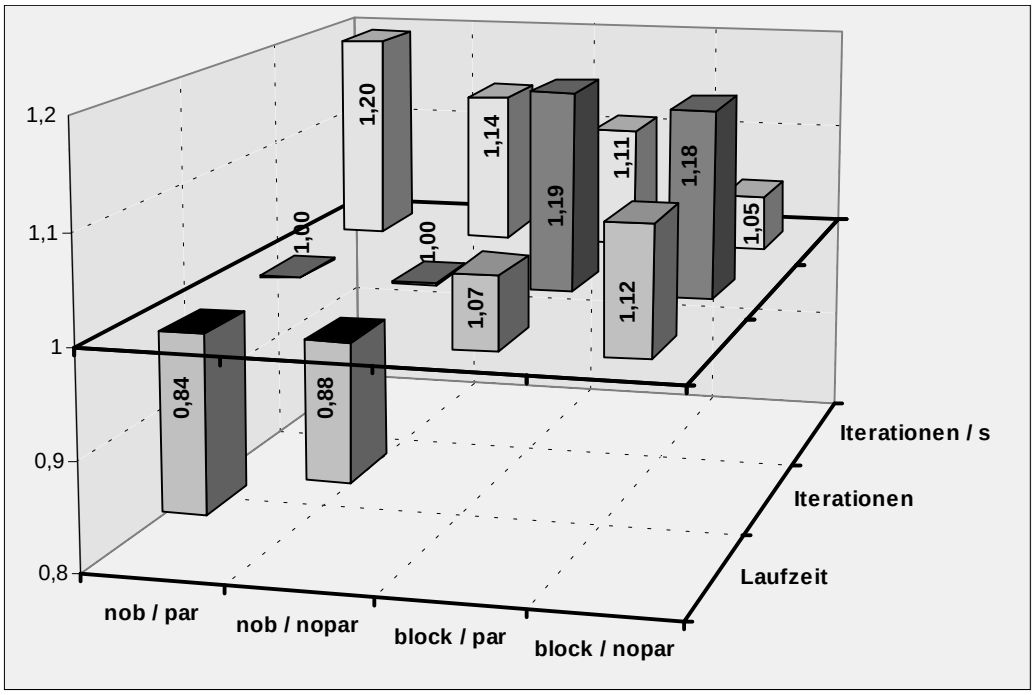


Abbildung 4.10: Summen der Lösungszeit und Iterationszahl sowie beider Quotient für den parallelen Algorithmus auf 2 PEs im Verhältnis zu den entsprechenden Werten bei 1 PE.

len Algorithmus konstant bleibt, zeigt sich eine durchschnittliche Erhöhung der Iterationszahl von etwa 20% durch das Block-Pivoting. Diese ist offenbar zu groß, um dennoch eine Beschleunigung zu gewährleisten.

Bei der Iterationsgeschwindigkeit (dritte Reihe in Abbildung 4.10) ist zweierlei zu beobachten. Zum einen zeigt sich eine Beschleunigung durch die parallele Lösung von zwei Gleichungssystemen, und zwar sowohl für die Testläufe mit als auch für die ohne Block-Pivoting. Den Daten kann jedoch kein Zusammenhang zwischen der erreichbaren Beschleunigung und anderen Größen wie die Dimension oder die Anzahl von NNEs der Basismatrix entnommen werden.

Zum anderen zeigt sich ein ähnlicher Effekt wie er bereits bei SoPlex festgestellt wurde, daß nämlich die Iterationsgeschwindigkeit mit zunehmender Anzahl von Iterationen sinkt. Aus diesem Grund fallen die Iterationsgeschwindigkeiten mit Block-Pivoting geringer aus als ohne. Im folgenden Abschnitt wird jedoch gezeigt, daß das Block-Pivoting in der Tat zusätzliche Parallelität erschließt, die sich in einer Beschleunigung auswirken kann.

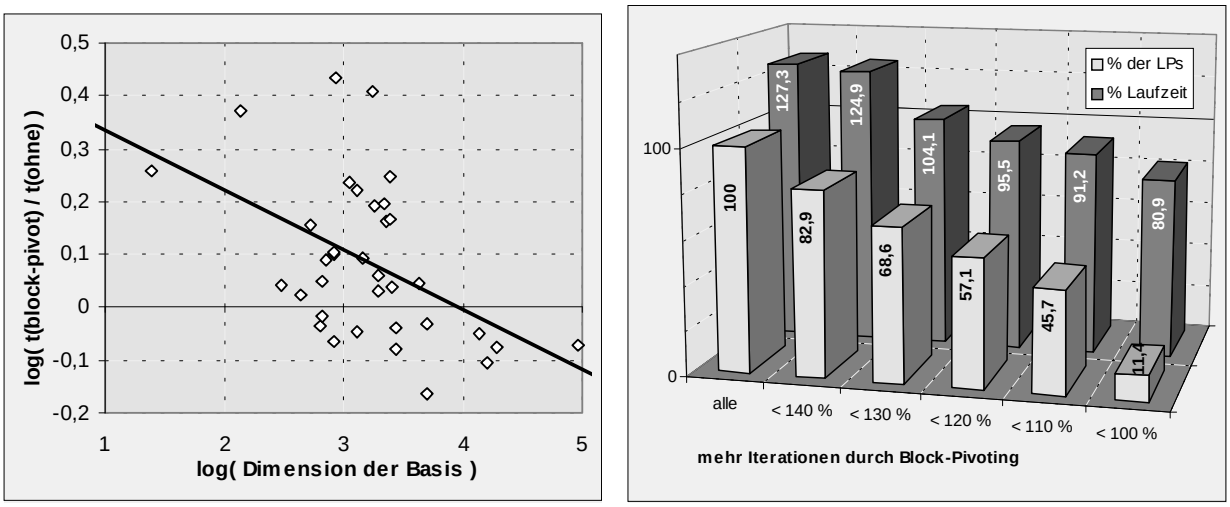


Abbildung 4.11: Beschleunigung durch Block-Pivoting in Abhängigkeit von (a) der Dimension der Basismatrix (b) der Zunahme der Iterationszahl.

4.3.3 Block-Pivoting

Abbildung 4.11.a zeigt das Verhältnis der Laufzeit für die Lösung der Testprobleme mit Block-Pivoting zu ohne Block-Pivoting in Abhängigkeit von der Dimension der Basismatrix. Die eingetragene Ausgleichsgerade zeigt an, daß je höher die Dimension ist, desto eher eine Beschleunigung durch das Block-Pivoting erzielt werden kann. Wie so oft ist auch dies kein streng gültiger Zusammenhang. Anschaulich leuchtet er ein, denn die Lösungszeit für die Gleichungssysteme korreliert mit der Dimension der Matrizen, so daß sich eine Parallelisierung mit Hilfe vom Block-Pivoting für große Matrizen eher lohnt.

Die unbefriedigenden Ergebnisse zum Block-Pivoting wurden auf die Zunahme der Anzahl der Iterationen zurückgeführt. Um diese These zu untermauern, wurden aus Tabelle A.16 verschiedene Teilmengen der LPs betrachtet, bei denen die Iterationszahl um weniger als einen vorgegebenen Prozentsatz zunimmt. In Abbildung 4.11.b wurden zwei Größen für verschiedene Schranken der maximalen prozentualen Zunahme der Iterationszahl aufgetragen. Die erste Reihe zeigt den Prozensatz der Testprobleme, bei denen sich die Iterationszahl beim Block-Pivoting um weniger als vorgegeben erhöht. Z.B. erhöht sich die Iterationszahl bei über 57% der Testprobleme um nicht mehr als 20%.

In der zweiten Reihe von Abbildung 4.11.b wird angegeben, wieviel Prozent der Laufzeit für die Lösung der jeweiligen Teilmenge von Testproblemen mit gegenüber ohne Block-Pivoting benötigt wird. Für 11,4% der Probleme nimmt die Iterationszahl nicht zu, und die Laufzeit kann mit Hilfe des Block-Pivotings auf etwa 81% herabgesetzt werden. Dies ist beachtlich, wenn man exemplarisch die Verteilung der Laufzeit bei der Lösung von stocfor3 in Tabelle 2.2 betrachtet: 60% wird im sequentiellen Fall auf die Lösung von Gleichungssystemen verwendet. Davon sind 3 Stück pro Iteration zu lösen, so daß für

jedes 20% der Laufzeit angesetzt werden kann. Geht man von optimalen Verhältnissen aus, so wird durch die anderen Parallelisierungen (Pricing, Quotiententest, Matrix-Vektor-Produkt und gleichzeitige Lösung von zwei Gleichungssystemen) 80% der Gesamtlaufzeit halbiert, so daß ohne Block-Pivoting die Laufzeit auf 60% = 80% / 2 + 20% reduziert wird. Das Block-Pivoting kann die 20% Laufzeit für das verbleibende Gleichungssystem auf 10% reduzieren, so daß mit Block-Pivoting idealerweise ein vollständig paralleler Algorithmus nur 50% der sequentiellen Lösungszeit benötigt. Gegenüber den 60% ohne Block-Pivoting entspricht dies einer Reduktion auf 83%. Das dieser Wert sogar unterschritten wird, liegt an den Beispielen, bei denen die Iterationenszahl mit Block-Pivoting niedriger als ohne ausfällt (Probleme 26, 28, 29 und 31 beim primalen Simplex und 2, 5 23, 28 und 29 beim dualen Simplex).

Bei mehr als 57% der Probleme kann im Mittel eine Beschleunigung um knapp 5% erreicht werden. Somit ist das Block-Pivoting zwar nicht für den allgemeinen Einsatz sinnvoll, jedoch lohnt es sich bei jeder Klasse von LPs zu überprüfen, wie sehr die Iterationszahl durch das Block-Pivoting zunimmt. In über 50% der Fälle ist damit zu rechnen, daß es sich lohnt, einen parallelen Algorithmus mit Block-Pivoting einzusetzen.

4.4 DoPlex

Zur Auswertung der Implementierung für Parallelrechner mit verteiltem Speicher, DoPlex, wurden die Test-Probleme auf einem Cray T3D jeweils mit verschiedenen Anzahlen von Prozessoren gelöst. Die Meßergebnisse sind in den Tabellen A.17 bis A.19 aufgelistet. Die dort auftretenden „Lücken“ haben mehrer Ursachen. Um den Rechenaufwand in Grenzen zu halten, wurden immer nur dann mehr PEs eingesetzt, wenn dies eine weitere Beschleunigung versprach. Außerdem konnten nicht alle LPs in dem von der Systemadministration vorgegebenen Zeitfenster von 1800 Sekunden gelöst werden. Schließlich konnten einige Probleme nicht gelöst werden, weil der Speicherplatz pro PE nicht ausreichte. Ein Beispiel dafür ist das Problem 35. Werden dafür 4 oder mehr PEs eingesetzt, so kann das LP trotzdem gelöst werden, weil das LP auf die PEs verteilt wird. Bei den Problemen 14 und 17 hilft auch die Verteilung auf mehrere PEs nicht, weil die Basis-Matrix, die ja auf allen PEs repliziert wird, zu groß wird.

Für die Meßreihen wurde wie bei SMoPlex der duale Simplex mit steepest-edge Pricing und Schlupfbasis verwendet. Für LPs mit mehr Zeilen als Spalten wurde eine Zeilen- andernfalls eine Spaltenbasis verwendet.

4.4.1 Parallele Lösung von Gleichungssystemen

Wir untersuchen zunächst die Auswirkungen der parallelen Lösung von linearen Gleichungssystemen. Dieser Ansatz nutzt lediglich Parallelität für zwei PEs aus, so daß keine Abhängigkeit von der Anzahl der eingesetzten Prozessoren zu erwarten ist.

Dies wird bestätigt von der in Abbildung 4.12 dargestellten durchschnittlichen Laufzeit pro Iteration mit und ohne parallele Lösung von Gleichungssystemen. Die Nutzung dieser Parallelität reduziert die Arbeit pro Iteration um einen festen Betrag, unabhängig von der Anzahl der PEs.

Die Darstellung wurde nur anhand der LPs gewonnen, für die auch noch mit 16 PEs Meßwerte bestimmt wurden. Betrachtet man hingegen alle LPs, die mit einem oder zwei PEs gelöst wurden, so wird mit 2 PEs im Durchschnitt eine Reduktion der Laufzeit auf 77% ohne und auf 70% mit paralleler Lösung von Gleichungssystemen erreicht (ohne Block-Pivoting). Diese Werte sind besser als die von SMoPlex erreichten (88% bzw. 84% vgl. Abbildung 4.10). Das ist darauf zurückzuführen, daß der T3D ein Parallelrechner mit verteiltem Speicher ist, bei dem der Zugriff auf den Speicher zu keiner Verstopfung des Busses führen kann, wie es bei Architekturen mit gemeinsamem Speicher geschieht.

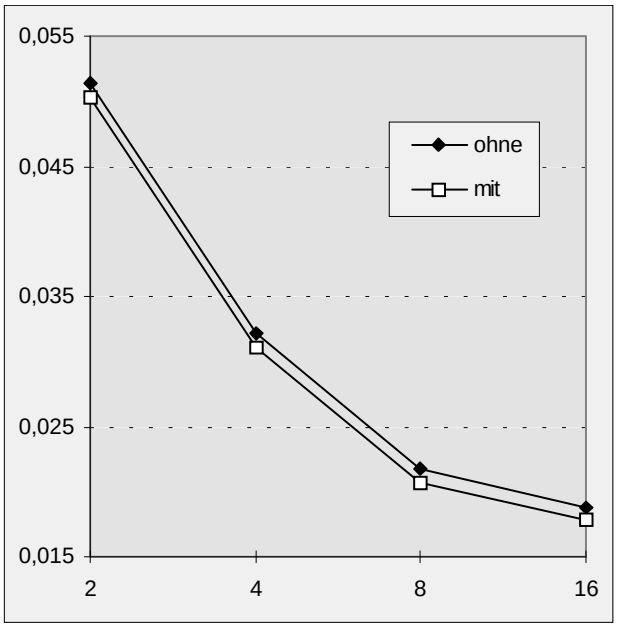


Abbildung 4.12: Summe der Zeit pro Summe der Iterationen mit und ohne gleichzeitigem Lösen zweier Gleichungssysteme in Abhängigkeit von der Anzahl der PEs.

4.4.2 Block-Pivoting

Das Problem beim Einsatz des Block-Pivotings ist die häufig damit verbundene Erhöhung der Iterationszahl. Sind mehr als 2 PEs beteiligt, so werden die Blöcke größer und es kann eine weiter Verschärfung des Problems erwartet werden. Die Auswirkungen auf die Laufzeit werden in Abbildung 4.13 gezeigt, in der die Zunahme der Laufzeit beim Block-Pivoting in Abhängigkeit von der Anzahl der PEs dargestellt ist. Es scheint ein logarithmisches Wachstum der Laufzeit in Abhängigkeit von der Anzahl der PEs vorzuliegen.

Bereits bei SMoPlex wurde festgestellt, daß es für das Block-Pivoting auch „gutartige“ LPs gibt, bei denen es zu keinem oder nur zu einem geringfügigen Wachstum der Iterationszahl kommt. Wir untersuchen daher, wie sich solche LPs bei Hinzunahme weiterer PEs verhalten.

In Abbildung 4.14 werden für verschiedene Anzahlen von PEs Mittelwerte über die LPs betrachtet, bei denen sich die Iterationszahl durch das Block-Pivoting um weniger als 10% erhöht. Da mit zunehmender Anzahl von PEs die Anzahl der betrachteten Test-LPs sinkt, wächst die Unsicherheit bei den beobachteten Ergebnissen. Für 16 PEs wurden nur noch 4 Testbeispiele betrachtet.

Der prozentuale Anteil der LPs für die das oben gesagte zutrifft ist in der mittleren Zeile von Abbildung 4.14 dargestellt. Es gibt also LPs, bei denen auch bei größeren Pivot-Blöcken die Anzahl der Iterationen nicht wesentlich zunimmt. Der Anteil solcher LPs scheint sich nicht zu ändern, so daß es es sinnvoll ist, zwischen gutartigen und und nicht gutartigen LPs zu unterscheiden.

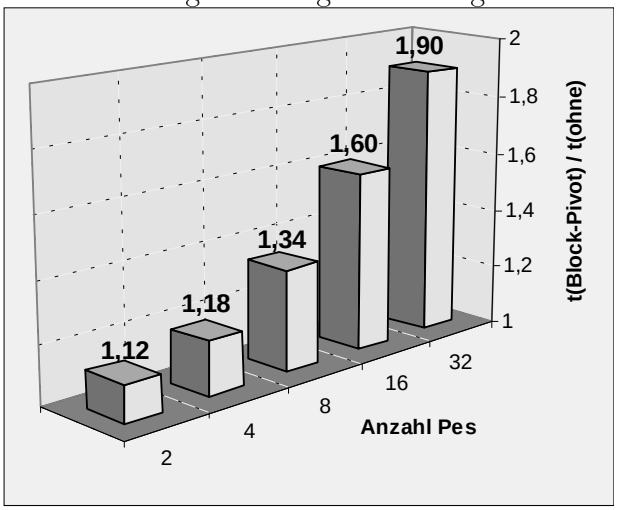


Abbildung 4.13: Verhältnis der Laufzeit mit zu ohne Block-Pivoting in Abhängigkeit von der Anzahl der PEs. Für jede Anzahl von PEs wurde die jeweils maximale Menge von LPs benutzt, für die Meßwerte vorliegen. Es wurden die Zeiten ohne paralleles Lösen von Gleichungssystemen verwendet.

Wie die hintere Zeile von Abbildung 4.14 zeigt, führt bei gutartigen LPs das Block-Pivoting auch bei mehreren PEs zu einer Beschleunigung von 80-90%. Die Beschleunigung scheint nicht von der Anzahl der PEs abzuhängen. Damit kann das Block-Pivoting bei gutartigen LPs zu einem skalierbaren parallelen Algorithmus führen, wie in der ersten Zeile in Abbildung 4.14 dargestellt ist.

4.4.3 Paralleles Matrix-Vektor-Produkt

Schließlich betrachten wir die Beschleunigung, die durch Parallelisierung des Matrix-Vektor-Produktes erzielt wird. Die Parallelisierung geschieht implizit durch die Verteilung der Daten, die gleichzeitig eine Parallelisierung der Vektor-Updates für g und des Pricings bzw. Quotiententests bedingt.

Die parallele Ausführung des Matrix-Vektor-Produktes und der Aktualisierung des Vektors g erfordert keine Kommunikation oder Synchronisation. Deshalb ist dafür eine gute Skalierbarkeit zu erwarten. Sofern diese Operationen den größten Anteil der sequentiellen Laufzeit einnehmen, kann die Parallelisierung dieser Schritte nach dem Amdahl'schen Gesetz auch zu einer Beschleunigung des gesamten Algorithmus führen.

In Abbildung 4.15 wurde die Beschleunigung des Simplex-Algorithmus ohne Block-Pivoting und paralelles Lösen von Gleichungssystemen für einige Test-Beispiele in Abhängigkeit von der Anzahl der PEs aufgetragen. Die Test-Beispiele wurden nach aufsteigendem Wert

$$s = \log\left(n \cdot \frac{c}{d}\right)$$

sortiert, wobei n die Anzahl der NNEs der Nebenbedingungsmatrix und d die Dimension der Basis-Matrix bezeichnen. Bei einer Zeilenbasis ist c die Anzahl der Zeilen der Nebenbedingungsmatrix, andernfalls die Anzahl der Spalten.

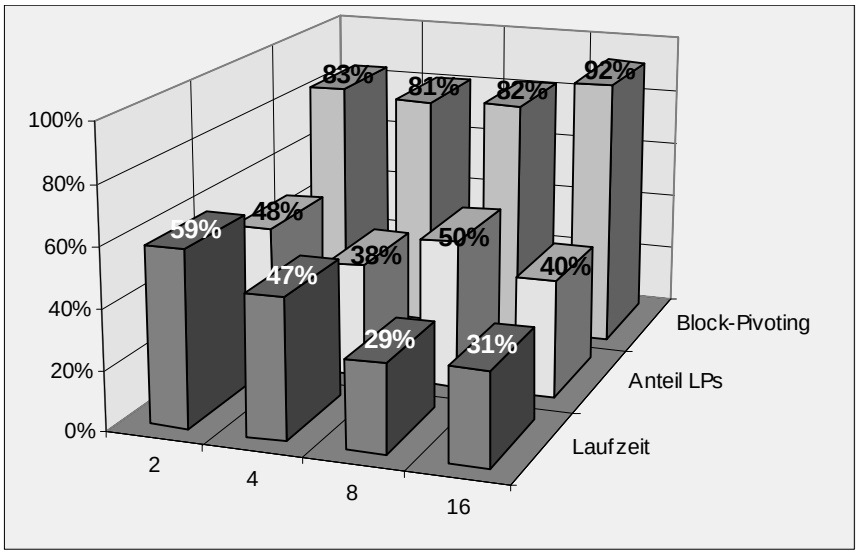


Abbildung 4.14: Verschiedene Werte für LPs, bei denen sich die Iterationszahl beim Block-Pivoting nicht mehr als um 10% erhöht, in Abhängigkeit von der Anzahl der PEs. Es werden dargestellt: Die Laufzeit im Verhältnis zur Laufzeit für 1 PE, der Anteil der PEs, die diese Bedingung erfüllt und die Laufzeit mit Block-Pivoting im Verhältnis zu ohne.

Insgesamt ist s ein Maß für die zu erwartende Skalierbarkeit durch Ausnutzung der oben genannten Parallelität bei der Lösung eines LPs: n ist eine Abschätzung für die Arbeit zur Berechnung des Matrix-Vektor-Produktes. $\frac{c}{d}$ gibt den Anteil der Laufzeit für die Berechnung des Matrix-Vektor-Produktes an der Gesamtlaufzeit an: Je kleiner die Dimension der Basismatrix gegenüber c ausfällt, desto geringer ist der Anteil der Arbeit für die Lösung der Gleichungssysteme mit der Basis-Matrix. Der Logarithmus wird nur benutzt, damit die Werte s kleiner ausfallen.

Abbildung 4.15 bestätigt die zu erwartende gute Skalierbarkeit von LPs mit hohem s. Bei dem fünften Beispiel von hinten handelt es sich um das Problem 34. Hier tritt eine ungewöhnliche Änderung der Iterationszahl bei unterschiedlichen Anzahlen von PEs, was den „Zacken“ verursacht. Bei den LPs mit $s > 6$ ist eine weitere Beschleunigung durch Hinzunahme weiterer Prozessoren zu erwarten.

4.5 Paralleler Löser für lineare Gleichungssysteme

Zur Bewertung der Implementierung des parallelen Lösers für unsymmetrische dünnbesetzte lineare Gleichungssysteme wurden Testläufe auf einem Cray T3D durchgeführt. Die dazu verwendete Menge von Test-Matrizen wird in Abschnitt 4.5.1 vorgestellt. In Abschnitt 4.5.2 wird der parallele LU Zerlegungs-Algorithmus evaluiert und mit der sequentiellen Version verglichen, die in SoPlex eingesetzt wird. Schließlich wird in Abschnitt 4.5.3 der parallele Lösungs-Algorithmus für gestaffelte Gleichungssysteme begewertet. Die Implementierun-

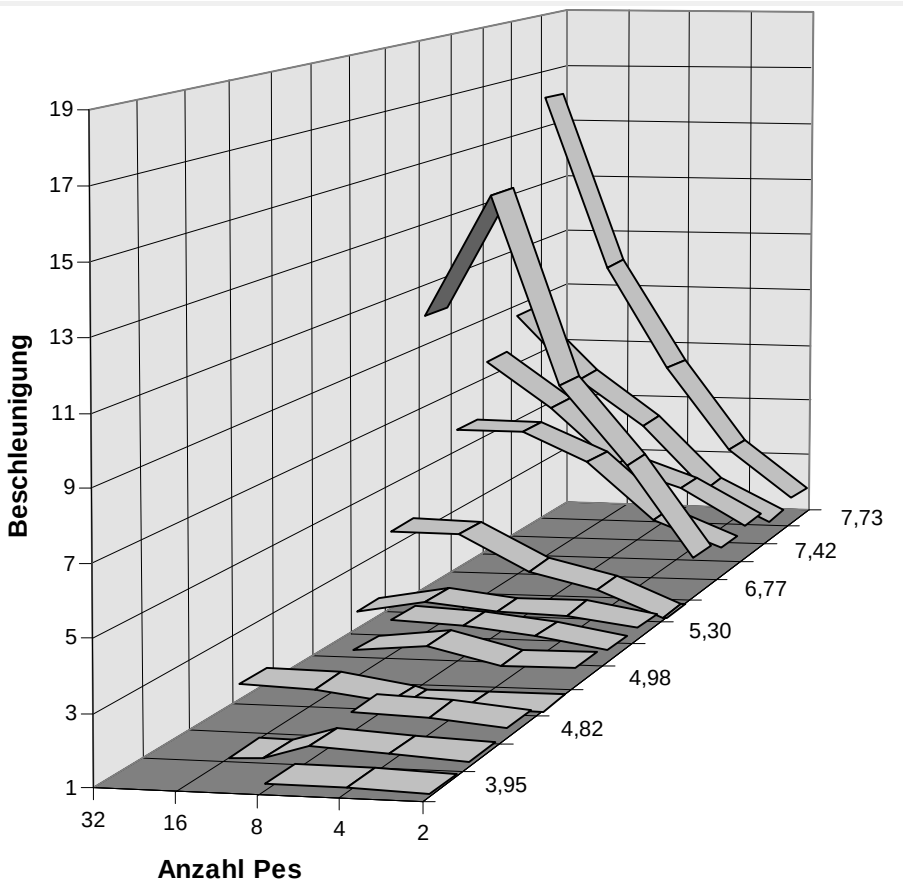


Abbildung 4.15: Beschleunigung in Abhängigkeit von der Anzahl von PEs für einige Test-LPs.

gen dieses Abschnittes wurden von Michael Ganss vorgenommen.

4.5.1 Test-Matrizen

Die Dimensionen und Anzahlen der NNEs der Test-Matrizen sind in den ersten Spalten von Tabelle A.20 aufgelistet. Die Matrizen stammen von zwei Quellen: Bei den Matrizen SM-50a bis stocfor3 handelt es sich um die mit SoPlex bestimmten optimalen Spaltenbasismatrizen der gleichnamigen LPs. Dabei wurden nur Matrizen höherer Dimension ausgewählt. Sie dienen zur Beurteilung der Anwendbarkeit des parallelen Löseres für den Simplex-Algorithmus.

Die restlichen Matrizen kommen aus dem Harwell-Boeing Testset [40]. Er ist fast schon ein Standard bei der Evaluierung von Gleichungssystemlösern und ermöglicht somit einen Vergleich mit anderen Implementierungen. Hierbei ist vor allem die in [5] vorgestellte Implementierung für den T3D interessant.

4.5.2 Parallele LU-Zerlegung

Tabelle A.20 ermöglicht zunächst einen Vergleich zwischen der sequentiellen Implementierung für SoPlex und der auf einem Prozessor laufenden parallelen Version des LU-Zerlegungs-Algorithmus'. Dieser Vergleich ist interessant, denn durch Kompatibilitätscheck und Bereithalten der maximalen Absolutbeträge für jede Zeile der aktiven Submatrix leisten beide Algorithmen unterschiedlich viel Arbeit.

Es wäre zu erwarten, daß die parallele Version mehr Arbeit leistet und somit höhere Laufzeiten aufweist. Dies trifft auch für alle Basis-Matrizen zu; die sequentielle Implementierung für SoPlex ist bis zu einem Faktor 5.46 schneller als die parallele Version (bnl1). Interessanterweise gilt dies nicht mehr für alle Probleme aus dem Harwell-Boeing Testset. Für „watt1" benötigt der sequentielle Algorithmus doppelt so lange wie der parallele mit einem Prozessor. Der Grund dafür liegt in der verstärkten Suche nach Pivot-Elementen mit geringer Markowitzzahl. Während die sequentielle Implementierung pro Iteration nicht mehr als 1-2 Zeilen oder Spalten nach guten Pivot-Elementen durchsucht, versucht die parallele Implementierung auch bei einem PE eine Menge von kompatiblen Pivot-Elementen zu konstruieren. Dazu untersucht sie mehr (hier 4) Zeilen oder Spalten und hat somit eine größere Chance Pivot-Elemente geringerer Markowitzzahl zu finden. Offenbar führt dies zu einer Folge von Pivot-Schritten mit weniger Fill, so daß der parallele Algorithmus insgesamt weniger Arbeit leistet.

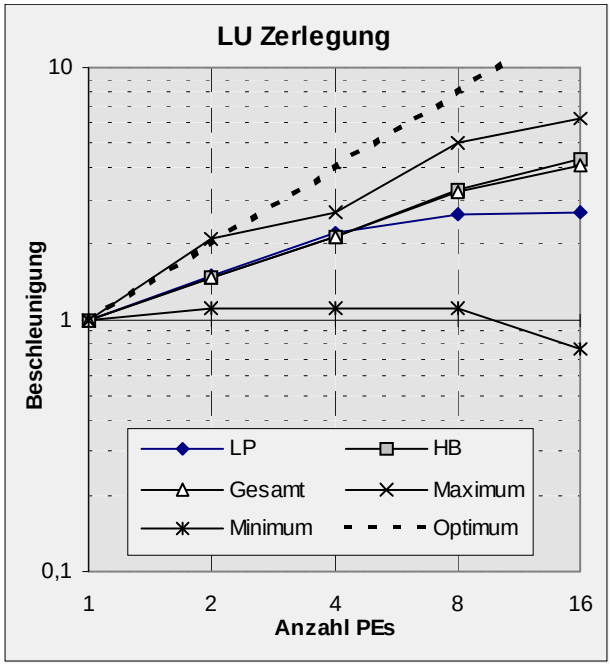


Abbildung 4.16: Beschleunigung des verteilten LU-Zerlegungsalgorithmus' in Abhängigkeit von der Anzahl von Prozessoren. Es werden die Beschleunigung für alle LP-Matrizen, die Harwell-Boeing Matrizen und alle Matrizen zusammen sowie für die am besten (35) und die am schlechtesten (1) skalierende Matrix dargestellt.

Die verbleibenden Spalten von Tabelle A.20 erlauben eine Beurteilung der Skalierbarkeit des parallelen LU Zerlegungs-Algorithmus'. Verschiedene Durchschnitte der erzielten Beschleunigungen sind in Abbildung 4.16 dargestellt. Man erkennt, daß selbst bei den LP Basis-Matrizen eine Beschleunigung erzielt werden kann. Sie ist vornehmlich für die Probleme, bei denen die parallele Version auf einem PE signifikant langsamer läuft als die sequentielle (fit1p, grow22, agg3, pilots, bnl1, scfxm2, ganges.ob4, maros, nesm, pilotwe) zu beobachten. Der Geschwindigkeitsgewinn durch Parallelität kann dies kaum wett machen. Somit bleibt festzustellen, daß der Einsatz der parallelen LU-Zerlegung für den Simplex-Algorithmus inadequat ist — jedenfalls bei den gewählten Probleminstan-

zen. Dies gilt umso mehr, als optimale Basis-Matrizen in der Regel die dichtbesetztesten Matrizen während der Ausführung des Simplex-Algorithmus' sind. Gleichungssysteme mit zwischenzeitlichen Basen werden somit meist mit weniger Aufwand gelöst, was den Nutzen von Parallelität für den Simplex-Algorithmus weiter einschränkt.

Liegt das unerfreuliche Ergebnis für die parallele Lösung von Basis-Matrizen vielleicht an einer schlechten Implementierung? Die Antwort ist Nein, was anhand der Testmatrizen aus dem Harwell-Boeing Testset im Vergleich zu anderen publizierten Ergebnissen auch nachgeweisen werden kann. R. Asenjo und E.L. Zapata berichten in [5] folgende Laufzeiten ihres Faktorisierungs-Algorithmus' auf einem Cray T3D:

Problem	1 PE	4 PEs	16 PEs
steam2	9.89	3.57	1.90
jpwh991	16.22	6.74	3.16
sherman1	3.02	1.93	1.35
sherman2	81.08	22.81	8.12
lns3937	204.55	81.89	28.58

Lediglich bei „sherman2“ und 16 PEs überholt die Implementierung von R. Asenjo und E.L. Zapata die hier vorgestellte Version. In allen anderen Fällen sind die Laufzeiten mitunter erheblich höher. Zwar ist ihre Effizienz nach (2.3) besser, jedoch ist die Bedeutung dieser Maßzahl angesichts der absoluten Laufzeiten in Frage zu stellen.

Insgesamt ist somit festzustellen, daß die im Rahmen dieser Arbeit erstellte parallele Implementierung vergleichsweise günstige Laufzeiten erreicht und somit nicht für den Mißerfolg bei den LP Basis-Matrizen verantwortlich gemacht werden kann. Die Skalierbarkeit der Implementierung steigt mit zunehmender Arbeit, was sie für den Einsatz in anderen Anwendungsfeldern interessant macht. Der Einsatz für die Lösung partieller Differentialgleichung ist geplant.

Als besonders interessantes Phänomen ist in Tabelle A.20 eine superlineare Beschleunigung für „lns2927“ bei 2 PEs zu erkennen. Dies ist darauf zurückzuführen, daß der parallele Algorithmus für unterschiedliche Anzahlen von PEs meist unterschiedlich viel Arbeit leistet. Je mehr PEs eingesetzt werden desto größer wird die Menge der Pivot-Kandidaten, was jeweils zu einer anderen Eliminationsfolge führt. Durch die größere Kandidatenmenge können zweierelei Effekte eintreten. Zum einen können so Pivot-Elemente mit geringerer Markowitz-Zahl gefunden werden, was zu einer Verringerung der Gesamtarbeit führt und die superlineare Beschleunigung für „lns2927“ erklärt.

Leider kann zum anderen auch der gegenteilige Effekt eintreten, was leider häufiger geschieht. In die größere Kanidatenmenge werden mangels Alternativen auch NNEs mit höherer Markowitz-Zahl aufgenommen. Fallen diese Kandidaten nicht durch den Kompatibilitätscheck, führen sie zu Eliminationsschritten, bei denen viel Fill erzeugt wird. Dadurch erhöht sich die Gesamtarbeit. Beim Problem „sherman3“ zeigt sich dieser Effekt so deutlich, daß die Laufzeit für 2 PEs die für 1 PE sogar übertrifft. Hier könnte eine Begrenzung der maximalen Markowitz-Zahl weiterhelfen.

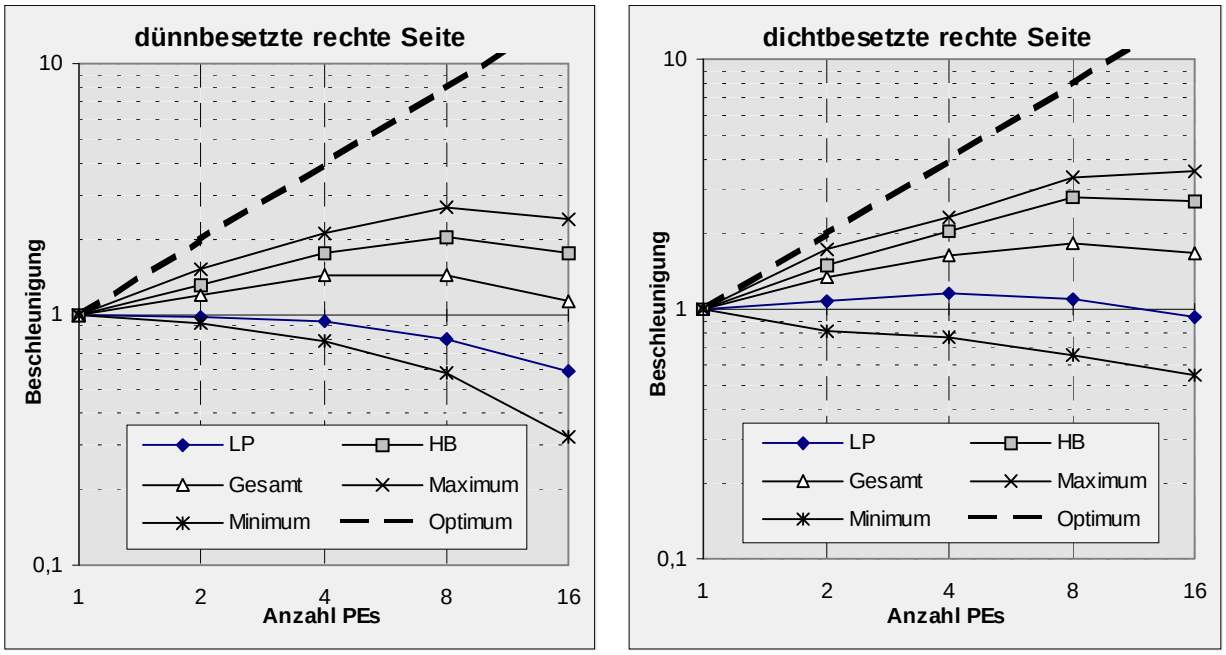


Abbildung 4.17: Beschleunigung des verteilten Lösung von Gleichungssystemen mit dünn- und dichtbesetztem Rechte-Seite-Vektor in Abhängigkeit von der Anzahl von Prozessoren. Es werden die Beschleunigung für alle LP-Matrizen, die Harwell-Boeing Matrizen und alle Matrizen zusammen sowie für die am besten (35) und die am schlechtesten (1) skalierende Matrizen dargestellt.

4.5.3 Lösen gestaffelter Gleichungssysteme

Bei der Bewertung der Implementierungen der Gleichungssystemlöser bei gegebener LU-Zerlegung müssen zwei Fälle unterschieden werden: Der Rechte-Seite-Vektor kann dicht- oder dünnbesetzt sein. In letzteren Fall ist weniger Arbeit zu verrichten und somit eine schlechtere Skalierbarkeit zu erwarten. Die Gleichungssysteme, die im Rahmen von Simplex-Algorithmen zu lösen sind, gehören fast immer zu dieser Kategorie.

Tabelle A.21 enthält Laufzeiten für die verteilte Lösung von Gleichungsystemen der Form

$$LUx = b \tag{4.1}$$

für verschiedenen Anzahlen von PEs. In den ersten Spalten wurde für b ein Einheitsvektor benutzt. Die Ergebnisse werden in Abbildung 4.17.a zusammengefaßt. Bei den LP Basis-Matrizen zeigt sich in fast allen Fällen eine Verlangsamung schon bei 2 PEs. Dagegen kann bei den meisten Harwell-Boeing Matrizen eine Beschleunigung bis zu 8 PEs erzielt werden. Diese spiegelt jedoch nicht die reine Beschleunigung bei der Lösung von (4.1) wieder. Vielmehr stammen die Faktoren L und U aus einer zuvor verteilten LU-Zerlegung. Wie im vorigen Abschnitt beschrieben wurde, ist zu erwarten, daß die Faktoren bei verschiedenen

Anzahlen von PEs unterschiedlich viele NNEs haben. Damit fällt bei der Lösung von Gleichungssystemen auch unterschiedlich viel Arbeit an, typischerweise also mehr Arbeit, je mehr PEs beteiligt sind. Demnach würden die Beschleunigungen für eine fest vorgegebene LU-Zerlegung leicht besser ausfallen als in Abbildung 4.17 dargestellt.

Dasselbe gilt auch für die Lösung von Gleichungssystemen mit dichtbesetztem Rechte-Seite-Vektor. Die entsprechenden Ergebnisse sind in Abbildung 4.17.b zusammengefaßt. Sie unterscheiden sich zu den Ergebnissen mit dünnbesetztem Rechte-Seite-Vektor lediglich darin, daß insgesamt eine bessere Skalierbarkeit erreicht wird. Dies ist aufgrund der höheren zu verrichtenden Arbeitsleistung auch zu erwarten.

Zusammenfassung

Es wurden drei Implementierungen des revidierten Simplex-Algorithmus vorgestellt, je eine für sequentielle Workstations, für Parallelrechner mit gemeinsamem Speicher und für solche mit verteiltem Speicher. Sowohl der primale als auch der duale Algorithmus wurden implementiert. Es werden dünnbesetzte Vektoren und Matrizen eingesetzt, insbesondere auch bei der LU-Zerlegung der Basismatrizen. Beim Pricing kommen effiziente Strategien, u.a. das steepest-edge Pricing, zum Einsatz, und der auf einer theoretischen Untersuchung der numerischen Stabilität basierende Quotiententest zeichnet sich durch eine hervorragende numerische Stabilität aus. Alles zusammen führt zu einer Implementierung, die sich mit state-of-the-art LP-Lösern wie CPLEX messen kann und für den täglichen Einsatz z.B. in Schnittebenenverfahren geeignet ist.

Die Implementierungen basieren auf einer neuen Darstellung, die den primalen und dualen Algorithmus mit einer Zeilen- und Spaltenbasis einheitlich zusammenfaßt. Dadurch können beide Algorithmen mit beiden Basisdarstellungen verwendet werden. Der Einsatz der Zeilenbasis bei LPs mit mehr Nebenbedingungen als Variablen kann den Implementierungen zu einer weiteren Geschwindigkeitssteigerung verhelfen. Dies gelingt auch mit Hilfe einer neuen Phase 1, die durch das Umschalten zwischen primalen und dualem Algorithmus funktioniert. Dadurch kann in Phase 1 ein LP verwendet werden, das sich so wenig wie möglich von dem Ausgangs-LP unterscheidet, wodurch insgesamt weniger Iterationen verwendet werden als bei den sonst verwendeten Phase 1 LPs. Desweiteren wird der Zeitpunkt der erneuten Faktorisierung der Basismatrix dynamisch so festgesetzt, daß die Iterationsgeschwindigkeit maximiert wird. Schließlich kommt ein Löser für lineare Gleichungssyteme zum Einsatz, der für extrem dünnbesetze Matrizen und Rechte-Seite-Vektoren optimiert wurde.

Den Implementierungen liegt ein objektorientierter Software-Entwurf zugrunde. Dieser ermöglicht es, für die parallelen Versionen wesentliche Teile durch Ableitung von der sequentiellen Version zu übernehmen. Lediglich die zusätzlichen Aufgaben zur Synchronisation, Kommunikation und Verteilung der Arbeit müssen hinzu implementiert werden. Außerdem unterstützt der objektorientierte Entwurf die Integration etwa von Pricing-Verfahren, die vielleicht in Zukunft entwickelt werden.

Bei den parallelen Versionen wurden vier Parallelisierungsansätze verfolgt. Beim Pricing und Quotiententest werden parallele Such-Algorithmen zur Bestimmung eines maximalen

oder minimalen Elementes eingesetzt. Die Parallelisierung des Matrix-Vektor-Produktes und der Vektoraddition lohnt sich besonders für Probleme mit einer sehr unterschiedlichen Anzahl von Zeilen und von Spalten. Für derartige LPs können durch den Einsatz vieler Prozessoren hohe Beschleunigungen erzielt werden, etwa um den Faktor 20 mit 32 PEs.

Wird das steepest-edge Pricing verwendet, so fällt ein zusätzliches lineares Gleichungssystem pro Iteration an. Dieses wird parallel zu einem der weiteren Gleichungssysteme jeder Iteration gelöst. Die Nutzung dieser Parallelität zahlt sich jedoch nicht immer aus. Dies liegt zum einen an dem damit verbundenen Kommunikations- oder Synchronisationsaufwand, zum anderen an der effizienten sequentiellen Implementierung, die beide Gleichungssysteme mit einer Traversierung der LU-Zerlegung löst. Sie benötigt weniger Zeit, als die zweifache Ausführung des Lösungs-Algorithmus für ein Gleichungssystem.

Die interessanteste Parallelisierung ist das Block-Pivoting. Dazu wird der Algorithmus so umstrukturiert, daß Gleichungssysteme aus an sich aufeinanderfolgenden Iterationen gleichzeitig parallel gelöst werden. Dies hat jedoch oft eine Erhöhung der Iterationszahl zur Folge, was eine Beschleunigung zunichte macht. Für die Problembeispiele, bei denen sich die Iterationszahl jedoch nur geringfügig erhöht, können für wenige PEs gute Beschleunigungen erzielt werden.

Schließlich wurde die verteilte LU-Zerlegung dünnbesetzter und unsymmetrischer Matrizen sowie die darauf basierende parallele Lösung linearer Gleichungssysteme untersucht. Es wurde eine Implementierung für den Cray T3D vorgenommen, die kompatible Pivot-Elemente für die parallele Elimination verwendet. Außerdem wird ein neues Lastausgleichs-Verfahren benutzt, das keinen zusätzlichen Kommunikations-Aufwand erfordert. Die Implementierung hat zwar eine geringere parallele Effizienz als vergleichbare Implementierungen, benötigt aber insgesamt weniger Zeit. Im Rahmen des Simplex-Algorithmus kommt sie jedoch nicht zum Einsatz, da für die extrem dünnbesetzten Basismatrizen heutiger LPs keine sinnvolle Beschleunigung erzielt werden konnte. Treten bei der Lösung von LPs dichter besetzte Basis-Matrizen auf, so könnte der parallele Löser für lineare Gleichungssysteme auch eine Beschleunigung des Simplex-Algorithmus bewirken.

Abschließend kann festgestellt werden, daß mit dieser Arbeit die Fortentwicklung von Simplex-Algorithmen auf mehreren Ebenen weiter vorangetrieben wurde. Die Parallelisierung für 2 PEs scheint für allgemeine LPs einsetzbar zu sein, wobei das Block-Pivoting jedoch nur für Problemklassen verwendet werden sollte, bei denen es zu keiner großen Erhöhung der Iterationszahl führt. Mehr als zwei PEs können nur für LPs sinnvoll eingesetzt werden, bei denen ein großer Unterschied in der Anzahl von Zeilen zur Anzahl von Spalten vorliegt. Bei solchen Problemen skaliert der parallele Algorithmus gut, und es können große Beschleunigungen erzielt werden.

Anhang A

Tabellen

Dieser Anhang enthält alle Meßwert-Tabellen, die für die Bewertung der Implementierungen SoPlex, SMoPlex und DoPlex sowie für den parallelen Löser für dünnbesetzte lineare Gleichungssysteme erstellt wurden. Ihre Auswertung und ggfs. graphische Aufbereitung findet sich in Kapitel 4.

Die Ergebnisse zu SoPlex und CPLEX wurden auf einer Sun UltraSPARC 1 Model 170E gemessen, die zu SMoPlex auf einer SGI Challange mit zwei 150MHz MIPS 4000 Prozessoren. Die Werte für DoPlex sowie für die parallele LU-Zerlegung und Lösung dünnbesetzter linearer Gleichungssysteme wurden auf einem Cray T3D gemessen. Jede PE hat 64 MB Speicher und einen 150 MHz DEC alpha Prozessor ohne 2nd level cache.

Bei allen Simplex-Testläufen wurde die maximale Berechnungszeit auf 3000 Sekunden beschränkt, auf dem Parallelrechner Cray T3D (aus administrativen Gründen) sogar auf 1800s. Konnte ein LP in dieser Zeit nicht gelöst werden, so wird dies durch „>“ angezeigt. Konnte ein LP hingegen aufgrund anderer Probleme, insbesondere aufgrund der numerischen Stabilität, nicht gelöst werden, so wird dies durch „—“ gekennzeichnet. Auf dem T3D wurde ausgehend von 1 PE ein LP nur dann mit mehr PEs gelöst, wenn eine weitere Beschleunigung erwartet werden konnte.

Bei den Meßergebnissen zu SoPlex wurden folgende Einstellungen verwendet:

- steepest-edge Pricing
- SoPlex Quotiententest
- Spaltenbasis
- skaliertes LP
- Crashbasis
- `maxcycle = 100` (vgl. Abschnitt 1.5)
- $\eta\theta = 10$ (vgl. Abschnitt 1.8.5)
- $\delta = 1e^{-6}$ (vgl. Abschnitt 1.3)

Nur die Abweichungen davon werden bei den Tabellen aufgeführt. Für die Messungen zu SMoPlex und DoPlex wurden folgende Einstellungen verwendet:

- steepest-edge Pricing
- SoPlex Quotiententest
- Basisdarstellung jeweils so, daß die Dimension niedriger ausfällt
- unskaliertes LP
- Schlupfbasis
- `maxcycle = 100` (vgl. Abschnitt 1.5)
- $\eta\theta = 10$ (vgl. Abschnitt 1.8.5)
- $\delta = 1e^{-6}$ (vgl. Abschnitt 1.3)

Nr.	Name	Anzahl Zeilen	Anzahl Spalten	Anzahl NNEs	Kondition der opt. Basis
1	agg3	516	302	4300	$1.4 \cdot 10^4$
2	bnl1	643	1175	5121	$4.1 \cdot 10^6$
3	fit2d	25	10500	129018	$2.6 \cdot 10^3$
4	ganges	1309	1681	6912	$1.2 \cdot 10^5$
5	grow22	440	942	8252	$1.9 \cdot 10^3$
6	maros	846	1443	9614	$1.2 \cdot 10^5$
7	nesm	662	2923	13288	$1.8 \cdot 10^6$
8	pilots	1441	3652	43167	$3.6 \cdot 10^7$
9	pilotwe	722	2789	9126	$1.5 \cdot 10^8$
10	scfxm2	660	914	5183	$4.0 \cdot 10^3$
11	stocfor3	16675	15695	64875	$2.5 \cdot 10^5$
12	agg3.ob4	2312	1128	52353	$4.9 \cdot 10^5$
13	bnl1.ob4	3345	1792	76596	$9.8 \cdot 10^5$
14	fit1p.ob4	3766	2508	1588810	$2.6 \cdot 10^3$
15	ganges.ob4	3320	2304	88562	$6.2 \cdot 10^5$
16	grow22.ob4	2609	1760	115904	$7.8 \cdot 10^5$
17	maros.ob4	3750	2180	102804	$2.3 \cdot 10^6$
18	nesm.ob4	3712	2488	158458	$6.2 \cdot 10^5$
19	pilotwe.ob4	4192	2452	177300	$7.1 \cdot 10^6$
20	scfxm2.ob4	3819	1940	94261	$7.3 \cdot 10^5$
21	SM-50.k-68a	11272	2723	26894	$4.5 \cdot 10^4$
22	SM-50.k-68b	11288	2723	27083	$6.8 \cdot 10^4$
23	kamin1807	20911	13542	42668	$1.7 \cdot 10^5$
24	kamin2702	33197	19092	67882	$2.9 \cdot 10^5$
25	chr15c	4964	1295	24795	$3.6 \cdot 10^5$
26	scr12	2735	1992	26292	$3.4 \cdot 10^6$
27	stolle	123964	93288	459680	
28	hansecom2	4967	10753	31114	$4.3 \cdot 10^3$
29	hansecom17	4967	55919	166249	$3.5 \cdot 10^3$
30	aa100000.p	837	68428	544654	$1.3 \cdot 10^4$
31	aa50000.p	837	35331	276038	$1.1 \cdot 10^4$
32	aa75000.p	837	52544	415820	$1.8 \cdot 10^4$
33	aa6.p	532	4316	24553	$5.7 \cdot 10^3$
34	osa030.p	4279	96119	262872	$4.7 \cdot 10^1$
35	nw16	139	148633	1501820	$4.4 \cdot 10^2$

Tabelle A.1: Die Test-LPs mit einigen Kenngrößen. Die Konditionswerte für die optimalen Basismatrizen wurden mit Hilfe von CPLEX bestimmt und gelten somit für das von CPLEX skalierte LP. Aufgrund der hohen Dimension von Problem „stolle“ konnte die zugehörige Konditionszahl von CPLEX nicht in einer akzeptablen Zeit berechnet werden; sie liegt vermutlich um 10^4.

Nr.	Lösung mit CPLEX 4.0.7					
	primaler Simplex			dualer Simplex		
	Zeit in Sekunden	Iterationen gesamt	Iterationen Phase 1	Zeit in Sekunden	Iterationen gesamt	Iterationen Phase 1
1	0.12	145	36	0.08	128	16
2	2.85	2219	1897	1.31	801	7
3	9.80	13926	0	78.41	5563	0
4	0.62	605	325	0.77	546	0
5	3.02	1061	0	6.42	1775	0
6	1.43	1209	605	3.12	1384	454
7	3.12	4077	1050	8.02	2887	59
8	100.80	7458	4269	70.53	4735	424
9	8.54	3037	1075	9.26	2634	1130
10	0.55	681	411	0.70	702	167
11	211.93	12913	7070	287.97	12728	8834
12	33.44	4645	0	55.67	4003	6
13	124.33	7756	11	118.76	5564	3
14	24.17	628	0	50.65	1445	0
15	42.72	5118	14	87.59	5705	2
16	27.16	3135	0	35.75	3395	0
17	105.52	9125	0	118.44	5821	1
18	256.23	11232	0	254.55	7142	7
19	212.93	11974	0	297.03	7717	10
20	160.25	10238	0	110.16	5494	2
21	7.04	1509	826	4.94	779	0
22	17.17	2530	1278	11.23	1270	0
23	468.45	20745	17473	171.10	10253	0
24	1431.64	31275	31275	668.53	18502	0
25	136.17	9174	2023	51.96	3585	597
26	136.50	13600	2536	116.52	9633	2224
27	$>$	$>$ 15000	$>$ 15000	735.19	9236	0
28	258.01	20979	13485	67.57	7337	0
29	$>$	$>$ 250000	233848	367.85	10889	0
30	1962.59	375293	4707	185.90	2157	0
31	427.45	134160	3924	34.50	1050	0
32	926.37	242696	4914	71.63	1352	0
33	54.52	12695	2449	14.44	2585	0
34	28.81	4852	1609	209.00	2927	0
35	84.89	23880	18577	66.93	436	0

Tabelle A.2: Laufzeiten und Anzahl der Iterationen für Phase 1 und 2 bei Lösung der Test-LPs mit CPLEX 4.0.7. Es wurden die Voreinstellungen verwendet, die man bei Aufruf des Programms erhält.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.12	129	43	0.57	396	127
2	1.83	944	942	2.04	1078	124
3	345.74	6335	0	90.68	5118	0
4	1.70	1264	1132	0.98	1131	0
5	3.93	1002	0	13.71	3445	0
6	4.05	1307	647	5.14	1805	1113
7	7.56	2617	2133	8.70	2694	43
8	128.31	4431	2625	210.15	6757	5638
9	16.21	2135	571	24.55	3499	2281
10	0.95	686	595	0.97	776	270
11	272.79	6659	1840	240.49	9931	6188
12	49.04	3044	0	46.61	2425	55
13	129.10	5148	0	104.29	3446	0
14	89.94	700	0	147.32	1822	0
15	51.31	3656	0	56.75	3142	0
16	44.72	2708	0	38.14	2100	0
17	88.24	3919	0	69.89	2848	7
18	124.23	4524	0	114.58	3061	0
19	153.46	4801	0	137.41	3193	0
20	117.02	4772	0	91.29	3371	0
21	3.77	810	810	3.88	838	0
22	8.42	1138	1136	10.35	1328	4
23	143.29	10206	10206	168.54	10657	0
24	707.45	19316	19316	817.51	19924	0
25	124.92	4401	1495	278.14	11048	486
26	149.26	8696	3349	255.17	9974	1820
27	1277.93	24740	9370	437.68	10503	0
28	79.03	4997	3683	68.87	6836	104
29	589.22	13510	8945	333.45	10596	27
30	1214.93	5414	1758	178.94	2007	0
31	152.52	1756	719	32.16	990	0
32	458.45	3115	1148	79.47	1426	0
33	34.78	3850	2483	17.52	2424	0
34	272.37	3178	3178	245.17	2796	36
35	15.31	14	5	56.74	316	0

Tabelle A.3: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.08	127	56	0.52	554	213
2	1.43	1077	1074	3.04	1989	553
3	822.77	23018	0	205.32	11939	0
4	1.30	1377	1219	0.82	1150	0
5	3.53	1103	0	8.08	3780	0
6	7.49	3136	1126	11.10	5144	3756
7	10.30	5009	3946	10.71	6077	0
8	271.80	12362	4200	466.91	21655	19531
9	74.80	12086	670	65.75	12295	10151
10	1.13	1155	1068	1.49	1489	860
11	186.30	9963	1812	128.73	12963	5320
12	45.35	3333	0	94.43	7403	66
13	165.50	6896	2	350.04	16460	0
14	64.21	628	0	137.47	2824	0
15	69.66	4117	0	75.91	6694	0
16	22.40	2147	0	27.10	3712	0
17	65.21	3640	0	106.61	6900	6
18	582.00	11281	54	163.24	8235	0
19	365.14	8116	5	270.31	11385	0
20	154.46	6546	12	196.49	10597	1
21	2.39	901	901	2.37	899	0
22	8.71	2279	2272	7.79	1975	1
23	171.46	19310	19310	243.75	23405	0
24	1212.41	54878	54878	1704.89	70816	0
25	>	>119732	7743	2859.40	199539	3646
26	2535.19	214685	43458	>	>147721	27058
27	>	>58218	11794	383.18	12296	0
28	126.58	10113	9587	128.78	12620	104
29	1754.89	46268	41609	983.38	29956	27
30	1306.72	11268	9618	933.50	8774	0
31	130.58	2812	2330	99.51	2402	0
32	486.64	6001	4813	310.86	4447	0
33	224.04	31459	29772	219.09	30574	0
34	402.61	5720	5720	255.94	3968	37
35	13.63	11	5	1668.36	6163	0

Tabelle A.4: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex mit Devex Pricing.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.09	134	55	0.64	736	349
2	3.59	2614	2614	3.92	2880	283
3	21.02	27004	0	285.35	16144	1
4	1.36	1528	1356	0.71	1137	0
5	4.19	1793	0	5.05	2483	0
6	5.77	3131	1590	6.48	3911	2035
7	5.37	3795	2626	5.94	3358	7
8	159.71	10501	4435	279.61	26828	24088
9	21.48	7577	1046	75.14	18657	9685
10	0.59	782	689	1.01	1238	526
11	187.37	16675	2022	73.89	12346	7377
12	80.82	10355	0	455.03	24096	58
13	312.24	15852	0	457.37	16089	0
14	83.72	733	0	122.67	1221	0
15	70.48	7875	0	387.42	20272	0
16	37.64	5194	0	44.39	3162	0
17	134.85	11686	0	723.27	28976	4
18	326.64	15671	0	710.37	21500	0
19	411.77	17219	0	1041.94	28045	0
20	213.50	15254	0	581.82	20697	0
21	2.85	1130	1130	2.45	980	0
22	22.26	4460	4459	14.30	3108	0
23	137.83	14918	14918	182.95	18284	0
24	611.51	30483	30483	609.98	31349	0
25	223.67	17842	10674	391.48	20278	1191
26	693.42	61410	31921	1511.94	63460	3685
27	1353.59	74303	51077	467.48	15174	0
28	1600.61	86108	82536	1740.21	90670	107
29	>	>38292	>38292	>	>43548	39
30	1106.25	58311	5388	1697.49	12413	0
31	165.52	16740	1804	255.87	4377	0
32	508.39	35135	3533	686.07	7086	0
33	109.80	15257	11080	148.55	17086	0
34	453.10	6099	6099	421.79	5595	45
35	16.91	11	3	320.38	1330	0

Tabelle A.5: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex mit partial multiple Pricing.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.07	122	55	0.48	604	259
2	3.45	2614	2614	4.66	3316	225
3	1385.72	61196	0	284.57	16144	1
4	1.52	1506	1356	0.82	1137	0
5	3.91	1205	0	5.09	2483	0
6	5.35	2910	1590	4.77	2892	1282
7	7.39	3906	2626	5.69	3358	7
8	240.94	12055	4435	380.94	21153	18456
9	51.82	10557	1047	64.20	12481	5593
10	0.57	768	689	0.85	982	348
11	211.22	10436	2022	95.55	10912	6348
12	807.87	71872	0	600.11	31233	36
13	2388.62	99801	0	455.79	16089	0
14	7.43	160	0	120.80	1221	0
15	171.49	11616	0	391.27	20272	0
16	221.37	16316	0	45.18	3162	0
17	637.20	31797	0	707.34	24669	5
18	2745.34	78259	1	720.22	21500	0
19	>	>83333	>0	1049.83	28045	0
20	>	>127904	>0	585.50	20697	0
21	3.22	1130	1130	2.74	980	0
22	24.07	4460	4459	15.60	3108	0
23	152.52	14918	14918	200.25	18284	0
24	663.33	30483	30483	662.12	31349	0
25	681.40	49006	10674	409.67	20554	570
26	1391.15	107307	31921	1514.45	64063	3028
27	2077.37	70501	51077	522.10	15174	0
28	1674.41	86234	82536	676.57	34140	104
29	>	>39079	>39079	>	>44659	>27
30	>	>18544	>5388	1666.28	12413	0
31	2136.91	27305	1804	251.34	4377	0
32	>			786.13	7846	0
33	198.20	30702	11080	142.66	17086	0
34	450.93	6099	6099	413.27	5554	43
35	13.32	6	3	320.39	1330	0

Tabelle A.6: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex mit most-violation Pricing.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.09	171	38	0.60	428	130
2	4.23	1056	891	3.43	927	128
3	455.70	6062	0	500.48	4468	9
4	2.15	1086	645	1.80	1227	0
5	5.30	1004	0	12.92	1835	0
6	4.40	1306	598	5.70	1513	872
7	32.44	3430	2610	31.41	2832	24
8	186.90	4810	2660	264.85	7911	6639
9	19.15	2371	714	39.53	2869	26
10	1.22	734	616	1.39	793	345
11	188.70	6809	1454	203.02	9245	5129
12	32.91	3150	0	108.33	3388	0
13	76.26	5113	0	319.98	5210	4
14	24.86	628	0	22.48	1012	0
15	31.61	3745	0	227.15	5097	0
16	5.68	1558	0	105.23	3010	0
17	39.68	3594	0	279.14	4990	0
18	63.76	4734	0	724.73	6973	13
19	106.49	4755	0	857.75	6945	9
20	86.01	5516	1	258.54	4974	0
21	2.14	767	767	2.40	861	0
22	4.53	1112	1112	6.23	1291	0
23	140.58	10361	10361	100.39	7766	504
24	516.11	18250	18250	391.61	14293	872
25	79.72	5637	2514	111.80	5073	707
26	175.38	10600	4454	254.11	10773	1556
27	869.15	20075	8408	530.83	10511	0
28	281.34	8042	4805	128.41	6284	147
29	>	>19755	>10933	950.40	9236	21
30	>	>9255	1304	585.73	1672	0
31	2089.62	14362	829	104.73	860	0
32	>	>13775	1208	241.57	1130	0
33	127.04	5955	2526	44.78	1980	0
34	1526.96	3704	3704	2147.41	4927	1169
35	92.91	83	0	>	>2077	0

Tabelle A.7: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex mit Zeilenbasis.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.16	120	40	0.22	230	106
2	1.43	783	783	1.77	971	107
3	350.48	6255	0	82.54	4921	0
4	1.63	1280	1155	0.99	1158	0
5	5.00	1109	0	10.89	2946	0
6	5.27	1611	602	8.01	2585	1368
7	7.60	3092	2544	7.73	2899	75
8	148.72	4997	2726	235.02	8203	7063
9	17.04	2349	744	20.71	2836	2187
10	1.01	714	643	1.38	981	397
11	300.21	6416	2043	239.64	9988	6282
12	47.51	3039	0	57.48	2558	0
13	129.82	5148	0	103.85	3446	0
14	75.73	620	0	553.90	2755	0
15	69.35	3656	0	56.96	3142	0
16	57.31	2708	0	38.09	2100	0
17	94.81	3669	0	72.84	2934	14
18	152.89	4524	0	114.56	3061	0
19	185.62	4801	0	139.02	3193	0
20	147.59	4772	0	91.59	3371	0
21	4.08	810	810	4.00	838	0
22	9.09	1138	1136	11.10	1328	4
23	145.76	10206	10206	171.73	10657	0
24	718.20	19316	19316	833.85	19924	0
25	126.99	4401	1495	279.27	11048	486
26	148.55	8696	3349	254.13	9974	1820
27	373.34	7826	6775	426.84	9351	0
28	78.37	4997	3683	68.19	6836	104
29	587.67	13510	8945	332.47	10596	27
30	1192.08	5414	1758	175.95	2007	0
31	152.42	1756	719	31.90	990	0
32	456.31	3115	1148	79.57	1426	0
33	34.79	3850	2483	17.48	2424	0
34	298.81	3231	3231	251.26	2781	0
35	14.11	14	5	55.17	316	0

Tabelle A.8: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex für das unskalierte LP.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.14	173	50	0.27	243	167
2	1.84	1047	1046	1.69	1021	4
3	88.80	5160	5160	88.62	5118	0
4	1.13	1466	1466	1.10	1458	0
5	12.63	3150	3150	9.60	2506	0
6	5.08	1840	1002	8.51	2672	1016
7	8.62	2798	2777	8.68	2875	14
8	129.87	4388	2546	207.73	6423	5700
9	16.44	2375	700	45.42	6198	2134
10	0.98	913	839	1.02	877	235
11	432.69	14153	6403	277.14	14261	10352
12	80.69	4277	4277	80.90	4277	0
13	195.03	7152	7152	195.20	7152	0
14	105.16	1436	1436	106.32	1436	0
15	129.00	7293	7293	127.96	7293	0
16	41.92	3562	3562	42.05	3562	0
17	221.70	8477	8477	224.65	8477	0
18	446.92	9760	9755	443.65	9760	5
19	507.34	10225	10219	506.83	10225	6
20	170.75	7238	7238	170.76	7238	0
21	3.64	838	838	3.69	838	0
22	12.40	1379	1376	12.38	1379	3
23	190.37	11790	11790	190.16	11790	0
24	797.55	19751	19751	797.61	19751	0
25	167.63	6068	603	61.46	3351	585
26	152.31	6408	1175	121.43	7765	1028
27	420.29	10487	10487	421.16	10488	0
28	74.46	6996	6996	73.91	6996	0
29	339.98	10196	10196	339.62	10196	0
30	178.57	1939	1939	178.71	1939	0
31	35.42	1031	1031	35.73	1031	0
32	73.88	1341	1341	74.31	1341	0
33	19.17	2618	2618	19.21	2618	0
34	286.32	3173	3173	285.98	3170	0
35	59.13	329	329	59.80	329	0

Tabelle A.9: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex mit Schlupfbasis.

	iterations (maxcycle)							
Nr.	**10**	**20**	**40**	**80**	**160**	**320**	**640**	**1280**
1	129	129	129	129	129	129	129	129
2	854	942	938	938	938	938	938	938
3	6335	6335	6335	6335	6335	6335	6335	6335
4	1291	1264	1264	1264	1264	1264	1264	1264
5	1097	1091	1091	1085	945	945	945	945
6	1124	1258	1258	1258	1258	1258	1258	1258
7	2674	2611	2573	2614	2614	2614	2614	2614
8	4804	4595	4464	4793	4814	4195	4670	4721
9	2054	2177	2325	2124	2141	2105	2105	2105
10	720	695	692	700	684	684	684	684
11	6687	6691	6650	6689	6662	6685	6656	6656
12	3385	3327	3231	3249	3114	4041	3516	3513
13	4246	4503	4597	5170	5144	4779	4676	4978
14	759	670	739	665	444	719	630	630
15	3668	3653	3698	3653	3524	3677	3719	3677
16	2461	2559	2569	2523	2820	2554	2512	2620
17	4105	3647	4329	3832	3731	3673	3687	3974
18	4215	4720	4354	4563	4352	4856	4981	5091
19	5110	4809	5027	5160	4801	4801	4801	4801
20	4667	5348	5183	5668	4285	5608	4838	5085
21	810	804	811	810	810	810	810	810
22	1165	1470	1450	1256	1365	1365	1365	1365
23	10303	10492	10339	10206	10206	10206	10206	10206
24	19248	19621	20202	19371	19292	19292	19292	19292
25	5363	4851	4726	4348	4698	4965	5916	6523
26	9378	8741	8768	8482	8632	8663	9219	9528
27	26540	26506	25772	24972	24155	23349	22695	22421
28	4939	5063	4813	4545	5012	5140	5335	5672
29	10894	14443	13227	14666	15585	15540	16109	16483
30	9818	6426	5205	5322	5119	6930	7147	7725
31	1671	1925	1676	1616	2077	2386	2735	3336
32	3919	2541	3135	3177	3387	3754	4341	5097
33	3714	3876	3873	3756	3984	4000	4193	4193
34	3233	3174	3200	3178	3172	3172	3172	3172
35	14	14	14	14	14	14	14	14

Tabelle A.10: Anzahl der Iterationen von SoPlex bei verschiedenen Werten von `maxcycle`.

Nr.	time / (100 iterations) 1	2	4	8	16	32
1	0.22	0.11	0.09	0.10	0.10	0.09
2	0.50	0.21	0.20	0.19	0.19	0.21
3	5.82	5.63	5.65	5.64	5.64	5.66
4	0.55	0.15	0.14	0.13	0.13	0.13
5	1.51	0.42	0.40	0.37	0.38	0.37
6	0.86	0.31	0.31	0.29	0.28	0.29
7	0.63	0.30	0.29	0.32	0.31	0.32
8	31.57	4.02	3.38	2.84	3.07	2.79
9	1.87	0.85	0.82	0.78	0.79	0.80
10	0.50	0.15	0.14	0.14	0.14	0.14
11	11.41	4.21	4.06	4.12	4.10	4.13
12	10.10	1.85	1.68	1.61	1.55	1.73
13	12.16	2.80	2.49	2.50	2.61	2.46
14	87.05	15.63	13.82	13.10	12.62	12.37
15	8.09	1.63	1.51	1.57	1.53	1.51
16	9.35	1.67	1.61	1.58	1.63	1.67
17	10.88	2.03	2.02	2.27	1.92	1.88
18	15.39	3.28	2.95	2.70	2.92	2.77
19	20.56	3.96	3.53	3.23	3.35	3.52
20	14.03	2.63	2.48	2.23	2.27	2.32
21	3.01	0.55	0.51	0.47	0.46	0.46
22	3.82	0.95	0.90	0.79	0.86	0.85
23	8.21	1.50	1.46	1.45	1.38	1.45
24	>	3.91	3.88	3.68	3.60	3.63
25	7.29	3.17	2.88	2.70	2.73	2.88
26	5.17	1.79	1.71	1.71	1.77	1.71
27	>	5.35	5.16	5.08	5.17	5.21
28	4.56	1.60	1.67	1.63	1.58	1.64
29	8.68	5.31	4.36	4.21	4.21	4.09
30	27.83	22.42	21.83	22.24	23.57	22.73
31	11.21	9.44	9.82	8.81	8.72	9.91
32	17.04	14.64	14.98	15.11	15.43	13.69
33	2.03	0.98	0.91	0.89	0.91	0.96
34	10.84	8.76	8.71	8.66	8.65	8.72
35	110.71	109.07	109.50	109.07	109.14	108.93

Tabelle A.11: Zeit für 100 Iterationen von SoPlex bei dynamischer Refaktorisierung in Abhängigkeit von dem Parameter $\eta\theta$.

Nr.	time / (100 iterations) 10	20	40	80	160	320
1	0.12	0.12	0.10	0.10	0.09	0.09
2	0.23	0.20	0.20	0.21	0.24	0.36
3	5.40	5.42	5.42	5.44	5.47	5.52
4	0.21	0.17	0.14	0.14	0.13	0.13
5	0.48	0.38	0.36	0.43	0.53	0.66
6	0.36	0.32	0.27	0.29	0.35	0.47
7	0.33	0.32	0.29	0.29	0.39	0.48
8	5.78	3.97	3.44	2.81	2.92	3.13
9	0.87	0.80	0.80	0.83	0.90	1.02
10	0.19	0.15	0.14	0.14	0.16	0.26
11	5.36	4.64	4.34	4.19	4.15	4.31
12	2.79	2.17	1.78	1.68	1.72	1.69
13	4.41	3.24	2.71	2.40	2.59	2.71
14	27.97	19.97	14.80	13.45	12.64	13.06
15	2.46	1.83	1.70	1.48	1.55	1.54
16	2.95	2.17	1.78	1.59	1.73	1.71
17	3.61	2.66	2.42	2.19	2.05	2.05
18	4.97	3.82	3.35	2.87	2.96	2.86
19	6.48	4.48	3.94	3.20	3.48	3.36
20	4.29	3.17	2.65	2.39	2.25	2.52
21	0.95	0.68	0.55	0.49	0.46	0.45
22	1.32	1.08	0.93	0.82	0.81	1.04
23	2.63	2.01	1.61	1.49	1.52	1.56
24	6.25	4.38	4.04	3.78	4.17	4.16
25	3.28	3.16	2.90	2.90	3.15	3.83
26	2.26	1.98	1.82	1.69	1.91	2.37
27	>	8.63	6.82	5.84	5.54	5.17
28	1.99	1.74	1.65	1.60	1.64	1.95
29	4.87	5.08	4.70	4.22	4.63	4.63
30	23.27	22.44	22.85	23.41	22.35	22.24
31	8.05	7.87	8.87	8.32	9.34	10.59
32	15.56	14.43	15.02	14.78	14.98	15.60
33	1.04	0.92	0.90	0.93	1.13	1.40
34	8.88	8.71	8.63	8.76	8.87	9.61
35	110.21	109.07	109.36	109.29	109.00	109.14

Tabelle A.12: Zeit für 100 Iterationen von SoPlex bei statischer Refaktorisierung in Abhängigkeit von der Anzahl von Basis-Updates zwischen zwei LU-Zerlegungen.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.13	122	42	1.00	608	290
2	2.03	869	868	2.16	1083	121
3	361.66	6384	0	109.81	5073	0
4	1.92	1304	1172	1.19	1184	0
5	—	650	0	12.72	2825	0
6	—	1534	1023	—	667	667
7	10.64	2989	2380	—	4682	0
8	—	2908	2908	401.11	12131	7554
9	17.83	2293	645	26.30	3813	2858
10	1.04	696	606	1.16	787	295
11	274.25	6625	1842	>	>10146	5845
12	—	5884	0	58.24	3027	92
13	83.96	3565	0	99.12	3210	0
14	88.12	634	0	171.56	1946	0
15	—	3584	0	84.60	3938	0
16	30.01	1977	0	41.97	2344	0
17	—	3489	0	74.94	3085	7
18	100.30	3437	0	110.63	3367	0
19	125.60	3463	0	—	3783	0
20	—	2674	0	110.44	3747	0
21	3.85	810	810	3.81	815	0
22	9.13	1150	1146	13.48	1414	3
23	148.96	10105	10105	163.84	10364	0
24	790.12	19513	19513	626.53	18580	0
25	—	2207	1542	—	2928	2928
26	—	4024	3589	—	3407	3407
27	—	17454	9372	432.32	10490	0
28	—	4313	4087	76.78	7012	105
29	—	7667	7351	1119.22	20892	86
30	—	1922	1757	219.54	1935	0
31	—	831	705	37.29	951	0
32	—	1369	1233	119.47	1536	0
33	—	2869	2630	23.66	2702	0
34	415.80	3176	3176	367.09	2784	36
35	106.89	232	5	69.98	317	0

Tabelle A.13: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex mit Harris Quotiententest.

Nr.	primaler Simplex			dualer Simplex		
	Zeit in	Iterationen		Zeit in	Iterationen	
	Sekunden	gesamt	Phase 1	Sekunden	gesamt	Phase 1
1	0.10	122	42	0.64	440	131
2	1.93	952	951	1.96	979	121
3	357.58	6387	0	89.46	5063	0
4	1.74	1298	1170	1.10	1195	0
5	4.72	1114	0	—	877	0
6	—	348	348	4.48	1719	820
7	—	1074	1074	—	637	0
8	—	672	672	—	172	172
9	13.89	2075	641	—	1560	1560
10	—	140	140	1.13	797	288
11	—	5659	1838	229.02	10649	5637
12	43.10	2495	1	58.41	2918	71
13	102.32	3911	0	—	2527	0
14	83.35	632	0	225.01	1935	0
15	45.15	2894	0	68.58	3437	0
16	35.88	2253	0	44.98	2366	0
17	71.82	3233	0	—	2802	2
18	90.83	3172	0	—	2689	0
19	124.21	3472	0	—	2167	0
20	128.48	4674	0	—	3512	0
21	3.34	814	814	3.65	859	0
22	10.75	1284	1284	11.74	1358	6
23	152.65	10381	10381	140.56	10089	0
24	720.89	19279	19279	636.62	18275	0
25	—	2087	1846	—	371	371
26	—	4565	4076	—	542	542
27	—	16409	9376	432.43	10489	0
28	98.59	5885	4979	84.51	7569	105
29	—	8297	7802	862.76	20866	86
30	—	2082	1780	198.31	2110	0
31	—	1005	666	42.65	1157	0
32	—	1342	1147	82.00	1408	0
33	28.36	3313	2330	21.15	2769	0
34	333.25	3384	3384	284.87	2832	36
35	103.88	239	5	55.01	307	0

Tabelle A.14: Laufzeiten und Iterationszahlen bei der Lösung mit SoPlex mit der stabilisierten Version des „Textbook“ Quotiententests.

	1 PE		2 PEs / nr. of blocks							
			2		4		8		16	
Nr.	Zeit	Iter.	Zeit	Iter.	Zeit	Iter.	Zeit	Iter.	Zeit	Iter.
1	0.17	154	0.23	154	0.22	154	0.22	154	0.25	154
2	4.41	1013	5.17	1035	4.95	971	5.35	988	5.94	1050
3	192.22	4677	145.75	4677	143.34	4677	146.17	4677	142.23	4677
4	3.29	1480	3.33	1507	3.41	1511	3.42	1505	3.38	1491
5	28.93	2305	41.20	3267	37.30	3123	38.17	3123	49.24	3516
6	20.36	2298	21.34	2363	21.49	2288	22.17	2367	22.98	2288
7	22.13	3197	16.56	2593	21.52	2948	26.59	3134	23.01	2726
8	756.15	7413	694.28	7400	694.93	7400	714.49	7400	728.38	7400
9	109.69	5737	109.56	5614	112.02	5614	116.17	5614	121.45	5614
10	2.94	913	3.22	936	3.32	936	3.46	936	3.65	936
11	984.31	15077	931.13	14953	935.17	14930	994.98	14926	1075.57	14911
12	236.84	2945	238.03	3090	242.02	3090	245.58	3090	253.86	3090
13	983.25	5584	780.24	5299	802.08	5299	814.10	5299	843.84	5299
14	1224.20	1008	877.12	1008	871.24	1008	871.80	1008	882.16	1008
15	590.91	4632	681.25	5276	685.11	5276	696.57	5276	703.18	5276
16	233.07	2839	182.32	2856	180.75	2856	187.25	2856	189.41	2856
17	814.72	5059	695.24	5327	694.55	5327	711.08	5327	746.05	5327
18	2210.08	6918	2062.66	6796	2102.64	6796	2119.00	6796	2240.69	6796
19	2749.91	7691	2015.23	6590	2003.12	6590	2059.64	6590	2074.23	6590
20	814.13	5175	780.11	5492	801.90	5492	803.47	5492	838.67	5492
21	7.31	836	6.17	836	6.61	836	5.80	836	5.97	836
22	17.45	1288	17.88	1319	18.63	1319	18.35	1319	18.98	1319
23	364.45	10546	334.12	10823	334.62	10823	339.25	10823	354.26	10823
24	1354.04	18617	1291.97	18346	1287.62	18346	1316.42	18346	1377.88	18346
25	216.44	6706	154.31	4878	166.33	5376	169.65	5667	174.92	5617
26	295.73	6105	309.84	6459	293.40	6149	309.34	6310	313.49	6119
27	1233.04	8117	1084.57	8162	1081.37	8162	1089.39	8162	1085.66	8162
28	202.81	6706	262.83	7861	210.60	7199	241.09	7381	236.64	7165
29	887.11	10601	832.70	11199	897.89	11502	970.64	12137	897.96	10989
30	468.54	2121	347.83	2040	365.24	2113	397.97	2278	324.35	1970
31	80.62	1052	58.34	979	58.56	1010	72.95	1095	73.70	1082
32	181.22	1365	160.58	1424	153.22	1404	168.24	1491	167.88	1491
33	51.75	2669	46.00	2478	46.21	2517	47.61	2478	51.69	2511
34	627.97	3147	535.19	3120	538.18	3132	543.95	3110	562.98	3153
35	130.39	320	86.03	320	85.20	320	87.00	320	87.52	320

Tabelle A.15: Laufzeiten und Iterationen von SMoPlex bei 2 Prozessoren ohne Block-Pivoting und der parallelen Lösung zweier Gleichungssysteme in Abhängigkeit von der Anzahl von Blöcken, in die die Nebenbedingungsmatrix für das Matrix-Vektor-Produkt unterteilt wird. Außerdem wird die Laufzeit und Iterationszahl für SMoPlex auf einem Prozessor angegeben.

Nr.	parallel systems				no parallel systems			
	block pivoting		no block pivots		block pivoting		no block pivots	
	Zeit	Iter.	Zeit	Iter.	Zeit	Iter.	Zeit	Iter.
1	0.22	166	0.20	154	0.21	166	0.23	154
2	4.08	957	4.42	1004	4.90	1022	5.17	1035
3	259.71	7138	143.34	4677	245.72	6778	145.75	4677
4	2.86	1515	3.18	1513	3.15	1510	3.33	1507
5	22.66	2205	21.40	2084	36.37	3002	41.20	3267
6	53.07	5818	19.47	2337	59.92	6026	21.34	2363
7	24.44	3593	21.85	2911	25.85	3472	16.56	2593
8	769.86	11823	620.67	7400	855.93	11784	694.28	7400
9	129.74	6903	105.19	5614	132.70	6704	109.56	5614
10	3.13	1013	3.26	936	3.24	1013	3.22	936
11	682.99	15402	871.50	14932	742.57	15434	931.13	14953
12	374.65	4997	218.34	3090	405.66	4997	238.03	3090
13	1156.87	6906	745.90	5299	1263.04	6906	780.24	5299
14	897.07	1073	818.70	1008	892.99	1073	877.12	1008
15	939.94	6589	647.94	5276	995.89	6589	681.25	5276
16	452.14	3998	176.91	2856	470.98	3998	182.32	2856
17	1052.64	6706	668.76	5327	1117.91	6706	695.24	5327
18	2875.66	9028	1961.16	6796	3102.98	9028	2062.66	6796
19	3374.20	9102	1907.36	6590	3601.41	9102	2015.23	6590
20	825.65	5727	721.21	5469	886.00	5727	780.11	5492
21	5.29	855	5.76	836	5.05	855	6.17	836
22	15.25	1356	18.32	1319	15.10	1356	17.88	1319
23	286.85	10802	321.06	10823	303.06	10802	334.12	10823
24	1030.53	18397	1224.03	18364	1044.64	18380	1291.97	18346
25	252.77	8351	151.20	5132	218.23	6747	154.31	4878
26	320.82	7245	298.50	6490	354.92	7272	309.84	6459
27	936.22	8234	1104.75	8162	911.85	8234	1084.57	8162
28	171.36	7065	250.55	8242	191.99	7090	262.83	7861
29	759.79	11277	965.72	11869	697.15	10679	832.70	11199
30	444.09	2533	353.31	2114	448.91	2576	347.83	2040
31	59.63	1000	68.89	1051	62.35	1014	58.34	979
32	196.59	1719	155.31	1447	202.82	1733	160.58	1424
33	66.46	3608	46.41	2670	68.55	3464	46.00	2478
34	590.59	3669	529.87	3118	595.82	3703	535.19	3120
35	200.72	558	85.30	320	194.09	558	86.03	320

Tabelle A.16: Laufzeit und Iterationen von SMoPlex auf 2 PEs für verschiedene Einstellungen für das Block-Pivoting und die parallele Lösung von Gleichungssystemen (bei 2 Blöcken für das Matrix-Vektor-Produkt).

Nr	Name	Iters.	Blocks
1	agg3	.27	156
2	bnl1	6.95	1015
3	fit2d	198.27	5103
4	ganges	3.77	1481
5	grow22	47.43	2740
6	maros	26.18	2221
7	nesm	32.92	3183
8	pilots	1124.76	8627
9	pilotwe	322.65	9962
10	scfxm2	3.94	873
11	stocfor3	667.54	14364
12	agg3.ob4	262.32	2990
13	bnl1.ob4	1047.23	5834
14	fit1p.ob4	not enough memory	
15	ganges.ob4	662.98	5003
16	grow22.ob4	279.34	3090
17	maros.ob4	645.16	4668
18	nesm.ob4	>	
19	pilotwe.ob4	>	
20	scfxm2.ob4	886.84	5324
21	SM-50.k-68a	8.83	855
22	SM-50.k-68b	18.32	1235
23	kamin1807	526.85	11278
24	kamin2702	1605.19	18416
25	chr15c	210.35	3318
26	scr12	397.52	5934
27	stolle	not enough memory	
28	hansecom2	361.71	7936
29	hansecom17	1499.15	22857
30	aa100000.p	410.82	2295
31	aa50000.p	58.36	964
32	aa75000.p	154.85	1357
33	aa6.p	64.99	2647
34	osa030.p	1273.76	7578
35	nw16	not enough memory	

	2 PEs											
	no Block-Pivoting						Block-Pivoting					
	no parallel systems			parallel systems			no parallel systems			parallel systems		
Nr	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks
1	.29	153	153	.29	153	153	.28	161	92	.28	161	92
2	6.28	975	975	5.54	962	962	6.29	1041	585	5.61	1024	579
3	105.01	5102	5102	104.71	5102	5102	202.91	7180	3622	202.25	7180	3622
4	3.50	1467	1467	3.72	1467	1467	3.18	1510	772	3.42	1512	773
5	40.77	2280	2282	32.14	1918	1924	41.42	2470	1264	21.46	1689	866
6	32.82	2906	2941	31.65	2978	3003	47.50	3792	2154	26.34	2800	1532
7	26.23	3114	3116	23.86	3152	3158	27.23	3683	1968	30.10	4112	2253
8	913.3	8358	9178	789.4	8509	9285	1107.77	10022	6180	953.90	10046	6191
9	181.86	6808	9045	167.67	6727	9378	174.17	6337	4757	196.24	9147	5912
10	3.52	856	856	3.32	856	856	3.85	931	504	3.61	931	504
11	597.10	14284	14288	632.61	14299	14300	583.02	15723	8286	577.97	15588	8195
12	259.65	3228	3228	196.76	3085	3085	464.03	4214	2719	332.02	3510	2163
13	790.44	4996	4996	719.41	4996	4996	1364.82	6241	3992	1228.88	6241	3992
14												
15	440.19	4571	4571	479.28	4664	4664	1046.08	6276	4054	978.43	6276	4054
16	219.03	3031	3031	202.66	2981	2981	270.28	3476	2252	331.59	3642	2286
17	585.55	4901	4901	549.93	4901	4901	1171.81	6292	4126	1082.07	6292	4126
18	1416.62	5991	5991	1326.76	5991	5991	>			>		
19	1343.25	5692	5692	1250.03	5692	5692	>			>		
20	676.13	4964	4964	656.84	4822	4822	939.63	6042	4017	934.78	6029	4013
21	6.46	855	855	6.81	855	855	4.70	858	468	5.23	858	468
22	16.03	1275	1275	16.23	1275	1275	13.65	1349	826	13.88	1349	823
23	402.49	11322	11322	438.07	11512	11512	360.44	11575	8156	388.18	11743	8300
24	1142.90	17865	17865	1135.41	17865	17865	1178.39	19822	14874	1163.03	19806	14839
25	136.03	2872	2872	113.81	2706	2706	211.05	4228	2719	177.61	3863	2494
26	398.48	6131	6131	416.14	7053	7053	536.55	8307	4869	569.14	9284	5444
27												
28	476.16	10368	10368	280.69	7737	7737	189.35	7314	4244	166.43	7115	4141
29	1566.18	32156	32156	1108.34	24518	24518	886.28	21516	12000	1066.31	24958	13781
30	246.22	2374	2374	214.70	2193	2193	273.65	2518	1534	281.78	2639	1610
31	29.63	891	891	32.23	965	965	40.80	1044	659	37.91	1005	629
32	73.48	1319	1319	68.34	1286	1286	84.74	1396	873	96.18	1502	938
33	47.53	2519	2519	38.89	2271	2271	70.52	3532	2056	71.45	3529	2034
34	557.79	5839	5844	584.70	6352	6352	315.90	3657	2113	325.08	3736	2164
35	not enough memory			not enough memory			not enough memory			not enough memory		

Tabelle A.17: Laufzeiten, Iterationen und Anzahl der Block-Pivots für die Lösung der LPs auf einem Cray T3D mit 1 und 2 PEs.

	4 PEs											
	no Block-Pivoting						Block-Pivoting					
	no parallel systems			parallel systems			no parallel systems			parallel systems		
Nr	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks
1	.30	159	159	.30	159	159	.31	163	51	.31	163	51
2	6.05	992	992	5.24	953	953	6.37	1129	351	5.83	1127	344
3	55.76	5102	5102	55.48	5102	5102	148.24	9981	2575	142.27	9713	2501
4	3.32	1467	1467	3.57	1467	1467	3.13	1561	412	3.30	1550	406
5	37.39	2297	2303	21.92	1712	1714	47.32	3021	834	28.14	2367	642
6	19.57	2013	2034	30.88	3003	3050	28.14	2973	1001	35.82	3760	1258
7	29.49	3334	3341	24.06	3179	3181	32.18	4487	1273	26.64	4311	1328
8	1051.33	10347	11374	879.74	10244	11198	1476.48	14721	5750	1213.60	14569	5689
9	269.50	10449	14055	155.56	6825	9923				230.40	11969	4702
10	3.25	844	844	3.09	844	844	4.37	1078	331	3.95	1088	329
11	540.68	14063	14065	596.57	14266	14269	472.03	15963	4559	497.92	15646	4477
12	231.70	3149	3149	214.78	3266	3266						
13	689.25	4775	4775	680.49	4948	4948						
14												
15	533.83	4757	4757	463.00	4818	4818						
16	211.47	2973	2973	215.43	3221	3221	382.17	4344	1742	1082.07	6292	4126
17												
18												
19	1571.99	6088	6088	>			>					
20												
21	5.20	852	852	5.59	852	852	3.32	864	254	3.82	864	254
22	12.82	1230	1230	13.16	1262	1262	9.38	1323	455	9.76	1323	455
23	346.68	11059	11059	359.97	11143	11143	299.62	11924	6602	333.77	11900	6576
24	1195.04	18756	18756	1133.59	18750	18750	1152.55	20707	12334	1063.46	20603	12256
25	135.11	3215	3215	130.51	3291	3291	179.04	4291	1970	210.88	5186	2335
26	413.38	6776	6776	386.44	6798	6798	613.07	11173	3967	656.89	12723	4480
27												
28	848.46	22111	22111	236.78	7508	7508	601.80	18368	5700	147.28	7107	2411
29	823.17	20756	20756	839.81	21863	21863	572.89	17161	5416	526.31	16303	5148
30	150.39	2429	2429	143.66	2416	2416	184.10	2875	1145	186.45	2942	1169
31	20.36	903	903	16.95	854	854	25.88	1066	415	25.21	1088	423
32	46.16	1267	1267	43.24	1241	1241	61.35	1590	640	63.10	1665	665
33	38.43	2312	2312	42.59	2553	2553	61.35	2820	1257	82.01	4305	1445
34	242.60	4244	4247	251.77	4721	4721	193.84	4009	1922	199.47	4077	1962
35	45.92	582	582	58.00	643	643	493.57	3526	961	517.61	3756	1033

	8 PEs											
	no Block-Pivoting						Block-Pivoting					
	no parallel systems			parallel systems			no parallel systems			parallel systems		
Nr	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks
1												
2	5.56	969	969	5.42	1039	1039	6.71	1206	228	6.54	1257	239
3	30.85	5109	5109	30.35	5109	5109	104.19	12557	1661	102.40	12557	1661
4	3.30	1513	1513	3.52	1513	1513	3.25	1647	229	3.50	1647	229
5	34.63	2268	2280	33.10	2446	2457	57.58	3710	510	32.41	2350	322
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21	4.52	849	849	4.96	849	849	2.98	875	152	3.30	875	152
22	13.23	1333	1333	13.43	1333	1333	8.92	1436	343	9.43	1436	343
23	333.18	11190	11190	344.37	11212	11212	279.96	12303	5123	290.92	12080	5235
24												
25	103.48	2869	2869	95.13	2869	2869	312.68	7158	2417	307.86	7502	2567
26												
27												
28	327.88	8794	8794	272.06	8275	8275	181.23	7625	1637	176.91	7666	1668
29	641.62	18577	18577	525.04	16283	16283	452.28	15421	2750	454.28	16445	2998
30	83.44	2241	2241	80.45	2280	2280	168.54	3706	1019	152.97	3544	1015
31	12.34	839	839	11.28	839	839	13.12	968	259	14.28	1069	308
32	38.04	1427	1427	34.01	1435	1435	44.59	1631	452	52.93	1907	532
33	37.50	2484	2484	33.88	2488	2488	83.01	5528	1126	77.16	5711	1148
34	163.60	4603	4603	189.23	5389	5389	114.71	3833	1760	125.39	3994	1854
35	64.90	1343	1343	54.37	1141	1141	265.76	3708	560	322.95	4413	677

Tabelle A.18: Laufzeiten, Iterationen und Anzahl der Block-Pivots für die Lösung der LPs auf einem Cray T3D mit 4 und 8 PEs.

	16 PEs											
	no Block-Pivoting						Block-Pivoting					
	no parallel systems			parallel systems			no parallel systems			parallel systems		
Nr	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks
1												
2												
3	19.79	4945	4945	20.75	5362	5362	71.77	14417	1070	70.54	14417	1070
4												
5	65.83	3719	3737	61.40	3456	3482	48.09	3142	234	37.18	2437	189
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21	4.35	849	849	4.78	849	849	2.81	872	91	3.31	872	91
22												
23												
24												
25	115.82	3126	3126	112.98	3267	3267						
26												
27												
28	257.61	7631	7631	227.73	7477	7477	188.99	7866	1122	196.69	8166	1166
29	429.83	15210	15210	355.95	13734	13734	391.05	15364	1722	360.26	14807	1619
30	63.39	2230	2230	54.06	2126	2126	117.68	3548	758	115.80	3884	879
31	10.25	857	857	10.50	926	926	16.81	1257	268	16.56	1319	279
32	25.88	1395	1395	22.18	1363	1363	35.63	1848	382	46.88	2237	474
33	33.10	2332	2332	30.44	2398	2398	106.81	6961	897	119.00	7081	860
34	91.14	3735	3735	135.68	5340	5340						
35	22.67	915	915	20.41	810	810	258.57	6099	539	265.47	6404	535

	32 PEs											
	no Block-Pivoting						Block-Pivoting					
	no parallel systems			parallel systems			no parallel systems			parallel systems		
Nr	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks	Zeit	Iters.	Blocks
1												
2												
3	12.37	4928	4928	12.09	4928	4928	48.73	15000	773	44.99	14915	791
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												
26												
27												
28	246.60	7575	7575	222.06	7581	7581	222.38	8455	833	215.26	8606	883
29	429.12	13988	13988	333.42	12976	12976	366.88	15721	1170	353.12	15196	1085
30	47.55	2136	2136	43.44	2184	2184	134.42	5022	935	111.05	4681	831
31	10.24	954	954	8.70	941	941	15.52	1328	224	13.71	1348	239
32	20.61	1298	1298	15.18	1207	1207	41.57	2388	419	43.78	2600	479
33	42.43	2461	2461	28.83	2253	2253	122.73	7476	627	103.20	7157	601
34	128.03	6374	6374	125.42	6370	6370						
35	12.52	896	896	10.39	788	788	137.80	6119	328	179.95	8144	411

Tabelle A.19: Laufzeiten, Iterationen und Anzahl der Block-Pivots für die Lösung der LPs auf einem Cray T3D mit 16 und 32 PEs.

Problem	dim	NNEs	seq	Anzahl PEs 1	2	4	8	16
SM-50a	2723	4301	0.06	0.19	0.17	0.17	0.17	0.25
SM-50b	2723	4603	0.09	0.20	0.17	0.18	0.17	0.23
chr15c	1695	6077	0.08	0.23	0.2	0.2	0.19	0.28
scr12	1992	7950	0.13	0.34	0.3	0.28	0.25	0.33
fit1p	2508	9480	0.28	0.77	0.7	0.55	0.49	0.47
ganges	1681	7020	0.09	0.18	0.16	0.17	0.18	0.2
osa030	4279	8532	0.11	0.32	0.25	0.26	0.23	0.28
grow22	1760	11215	0.15	0.53	0.36	0.31	0.26	0.33
hansecom2	4967	13812	0.22	0.65	0.59	0.57	0.50	0.6
hansecom17	4967	14018	0.17	0.6	0.54	0.52	0.48	0.55
agg3	1128	22678	0.51	2.08	1.2	0.78	0.54	0.46
kamin1809	13542	22968	0.33	1.2	1.1	1.2	1.0	1.05
pilots	1441	18376	1.63	2.1	1.35	1.2	1.0	0.92
bnl1	1792	26845	0.79	4.3	2.5	1.61	1.0	0.79
scfxm2	1940	27546	0.75	3.5	2.1	1.33	0.85	0.67
ganges.ob4	2304	28199	0.67	2.4	1.5	0.93	0.63	0.57
maros	2180	34254	0.81	3.6	2.2	1.3	0.87	0.75
kamin2702	19092	33187	0.55	1.95	1.7	1.7	1.6	1.58
nesm	2488	39095	1.09	4.6	2.7	1.61	1.0	0.83
pilotwe	2452	49135	1.50	5.1	3.1	1.81	1.16	0.93
stocfor3	16675	51412	0.70	2.2	1.8	1.68	1.58	1.75
west2021	2021	7353	0.13	0.4	0.3	0.3	0.24	0.3
sherman4	1104	3786	1.07	1.2	0.8	0.6	0.5	0.5
steam2	600	13760	0.97	1.1	0.9	1.0	1.1	0.9
sherman1	1000	3750	1.6	1.6	1.1	1.0	0.7	0.6
mcfe	765	24382	2.9	4.0	2.7	2.0	1.5	1.3
jpwh991	991	6027	5.7	7.2	3.9	2.9	1.8	1.5
orani678	2529	90158	4.3	7.5	4.4	3.5	2.3	1.7
orsreg1	2205	14133	26	20	16	10	9	8
watt1	1856	11360	52	26	16	14	9	7
watt2	1856	11550	43	29	25	14	11	6.8
sherman5	3312	20793	18	33	17	11	8	7
sherman2	1080	23094	28	51	31	26	15	10
sherman3	5005	20033	117	63	64	36	23	16
lns3937	3937	25407	52	75	36	28	15	12

Tabelle A.20: Statistische Daten zu den Test-Matrizen für die parallele LU-Zerlegung und Laufzeiten für die verteilte und sequentielle LU-Zerlegung dünnbesetzter Matrizen auf dem Cray T3D. Die Spalte „seq“ enthält die Zeiten für den in SoPlex benutzten Solver auf einer PE. Die folgenden Spalten enthalten die Laufzeiten für den verteilten Algorithmus in Abhängigkeit von der Anzahl der eingesetzten PEs.

	dünnbesetzt / Anzahl PEs					dichtbesetzt / Anzahl PEs				
Problem	1	2	4	8	16	1	2	4	8	16
SM-50a	1.1	1.2	1.4	1.9	3.4	2.3	2.8	3.0	3.5	4.2
SM-50b	1.1	1.3	1.4	1.9	2.7	2.5	3.0	3.1	3.6	4.3
chr15c	1.1	1.2	1.3	1.5	2.2	2.6	2.5	2.3	2.4	2.9
scr12	1.8	1.8	1.8	2.0	2.5	3.5	3.2	2.9	3.0	3.9
fit1p	1.4	1.7	1.8	2.2	3.0	3.3	3.6	3.4	3.9	4.5
ganges	0.9	1.0	1.1	1.4	2.0	2.8	2.7	2.7	2.9	3.3
osa030	1.8	1.9	2.0	2.7	3.8	3.7	4.3	4.4	5.2	6.2
grow22	1.1	1.3	1.3	1.4	2.0	3.7	3.3	3.0	2.9	3.4
hansecom2	3.6	3.6	3.6	4.0	5.9	9.3	8.7	8.4	8.7	10.3
hansecom17	3.3	3.2	3.4	4.0	5.6	9.0	8.5	8.3	8.8	10.2
agg3	1.5	1.5	1.5	1.4	2.1	5.1	3.8	2.8	2.5	2.8
kamin1809	7.2	8.0	8.2	10.8	14.5	15.0	16.9	17.6	20.2	24.3
pilots	5.3	4.9	3.2	3.0	3.5	8.0	5.8	4.5	4.1	4.4
bnl1	2.4	2.2	2.0	2.1	2.7	7.3	5.4	4.3	3.9	4.1
scfxm2	2.2	2.6	2.7	2.4	2.7	7.1	5.3	4.4	4.0	4.4
ganges.ob4	2.1	2.3	2.1	2.5	3.1	7.1	5.5	4.6	4.4	5.0
maros	2.3	2.3	2.1	2.3	3.6	8.0	6.1	4.7	4.3	4.7
kamin2702	12.8	13.0	15.0	19.0	25.4	22.0	24.5	25.9	29.7	36.6
nesm	2.6	2.6	2.7	2.7	3.4	9.2	6.7	5.4	5.0	5.4
pilotwe	2.7	2.7	3.6	3.1	3.3	10.3	7.2	5.8	5.1	5.6
stocfor3	8.5	8.8	9.4	11.2	16.1	26.9	25.5	24.6	25.2	29.5
west2021	2.3	2.4	2.2	2.3	3.1	4.8	4.3	3.9	4.0	4.6
sherman4	1.7	1.7	1.3	1.8	2.4	3.9	3.1	2.8	2.6	3.1
steam2	2.6	2.2	2.4	1.9	2.2	5.2	3.6	3.3	2.5	2.7
sherman1	2.1	2.1	1.7	1.8	2.5	4.5	3.6	2.8	2.7	3.1
mcfe	7.2	5.4	4.2	3.7	4.8	10.5	7.0	5.3	4.6	5.2
jpwh991	6.1	4.1	3.1	1.9	3.7	9.8	6.7	4.7	3.9	4.4
orani678	7.9	6.6	6.1	6.0	7.5	18.2	12.9	11.2	9.4	9.4
orsreg1	20.4	18.1	10.7	9.4	10.4	29.2	23.0	15.4	11.4	11.9
watt1	21.8	11.8	10.5	8.6	9.5	32.8	17.7	15.6	11.0	10.9
watt2	16.3	13.4	12.3	9.1	9.6	28.5	18.8	16.5	11.1	11.1
sherman5	13.0	9.3	7.2	7.8	9.9	31.7	20.2	14.6	11.6	12.4
sherman2	16.0	12.0	7.7	6.2	7.0	39.7	23.2	14.6	9.1	8.8
sherman3	23.4	23.5	14.0	13.7	14.8	56.7	45.4	26.3	19.2	19.1
lns3937	38.6	25.7	18.5	14.5	16.0	65.3	37.9	28.0	19.3	18.3

Tabelle A.21: Laufzeit für die verteilte Lösung dünnbesetzter linearer Gleichungssysteme mit dünn- und dichtbesetzem Rechte-Seite-Vektor nach verteilter LU-Zerlegung in Abhängigkeit von der Anzahl der PEs.

Literaturverzeichnis

[1] G.M. Amdahl, *Validity of the single-processor approach to achieving large-scale computing capabilities*, AFIPS Conference Proceedings, 30 (1967), pp. 483-485

[2] N. Amenta and G.M. Ziegler, *Deformed products and maximal shadows of polytopes*, TU Berlin Preprint Reihe Mathematik, Report No. 502/1996

[3] E.D. Andersen and K.D. Andersen, *Presolving in linear programming*, Math. Prog. 71 (1995), pp. 221-245

[4] K.M. Anstreicher and T. Terlaky, *A monotonic build-up simplex algorithm for linear programming*, Op. Res. , 42, 3 (1991), pp. 556-561

[5] R. Asenjo and E.L. Zapata, *Sparse LU factorization on the Cray T3D*, Lecture Notes in Computer Science, 919 (1995), pp. 690, Springer Verlag

[6] O. Axelsson, *A survey of preconditioned iterative methods for linear systems of algebraic equations*, BIT 25 (1985), pp. 166-187

[7] D.A. Babayev and S.S. Mardanov, *Reducing the number of variables in integer and linear programming problems*, Comp. Opt. Appl. 3 (1994), pp. 99-109

[8] M.A. Baddourah and D.T. Nguyen, *Parallel-vector processing for linear programming*, Computers & Structures No. 38, Vol. 3 (1991), pp. 269-281

[9] R.S. Barr and B.L. Hickman, *Parallel simplex for large pure network problems: Computational testing and sources of speedup*, Op. Res. 42, 1 (1994), pp. 65-80

[10] R.H. Bartels and G.H. Golub, *The simplex method of linear programming using LU decomposition*, Comm. ACM 12 (1969), pp. 266-268

[11] R.K. Baryton, F.G. Gustavson, and R.A. Willoughby, *Some results on sparse matrices*, Math. Comp. 24, 122 (1970), pp. 937-954

[12] M. Benichou, J.M. Gauthier, G. Hentges and G. Ribieri, *The efficient solution of large-scale linear programming problems*, Math. Prog. 13 (1977), pp. 280-322

[13] R.E. Bixby and A. Martin, *Parallelizing the dual simplex algorithm*, ZIB Preprint SC 95-45 (1995)

[14] R.E. Bixby, J.W. Gregory, I.J. Lustig, *Very large-scale linear programming: A case study in combining interior point and simplex methods*, Report (1991), Department of Mathematical Sciences, Rice University, Houston Texas

[15] R.E. Bixby, *Das Implementieren des Simplex-Verfahrens: Die Startbasis*, ZIB Preprint SC 92-11 (1992)

[16] R.E. Bixby, *Progress in linear programming*, ORSA J. Comp. 6, 1 (1994), pp. 15-22

[17] R.G. Bland, *New finite pivoting rules for the simplex method*, Math. of Op. Res. 2 (1977), pp. 103-107

[18] A.L. Brearley, G. Mitra and H.P. Williams, *Analysis of mathematical programming problems prior to applying the simplex algorithm*, Math. Prog. 8 (1975), pp. 54-83

[19] R. Borndörfer, C. Ferreira and A. Martin, *Decomposing matrices into blocks*, ZIB preprint, to appear

[20] T.B. Boffey and R. Hay, *Implementing parallel simplex algorithms*

[21] K.H. Borgwardt, *The simplex method: A probabilistic analysis*, Algorithms and Combinatorics, Vol 1. Springer Verlag

[22] E.D. Brooks III, *Massively parallel computing*, in „The 1992 MPCI Yearly Report: Harnessing the Killer Micros“, E.D. Brooks et al eds., Massively Paralle Computing Initiative, Lawrence Livermore National Laboratory, Livermore, California 94550 (1992)

[23] R. Burkard, S. Karisch and F. Rendl, *QAPLIB — a quadratic assignment problem library*, Eur. J. Op. Res. 55 (1991), pp. 115-119

[24] A. Charnes, *Optimality and degeneracy in linear programming*, Econometrica, 20 (1952), pp. 160-170

[25] H.-D. Chen, P.M. Pardalos and M.A. Saunders, *The simplex algorithm with a new primal and dual pivot rule*, Op. Res. Let. 16 (1994), pp. 121-127

[26] V. Chvatal, *Linear Programming*, W.H. Freeman, New York (1983)

[27] *Using the CPLEX callable library*, CPLEX Optimization, Inc., Suite 279, 930 Tahoe Blvd., Bldg. 802, Incline Village, NV 89451, USA (1995) URL: `http://www.cplex.com/`.

[28] A.R. Curtis and J.K. Reid, *On the automatic scaling of matrices for Gaussian elimination*, J.o.t.Inst. of Math. a.i. Appl. 10 (1972), pp. 118-124

[29] G.B. Dantzig and W. Orchard-Hays, *The product form for the inverse in the simplex method*, Math. Comp. 8 (1954), pp. 64-67

[30] G.B. Dantzig, A. Orden and P. Wolfe, *The generalized simplex method for minimizing a linear form under linear inequality retraints*, Pacific Journal of Mathematics, 5 (1955), pp. 183-195

[31] G.B. Dantzig and P. Wolfe, *Decomposition principle for linear programs*, Op. Res. 8 (1960), pp. 101-111

[32] G.B. Dantzig, *Linear programming*, in „Histroy of Mathematical Programming“, J.K. Lenstra, A.H.G. Rinnooy Kan and A. Schrijver eds., North-Holland Amsterdam, 1991

[33] T.A. Davis and P. Yew, *A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization*, SIAM J. Matrix Anal. Appl. 11, 3 (1990), pp. 383-402

[34] P. Deuflhard and A. Hohmann, *Numerische Mathematik I*, de Gruyter (1993)

[35] E.W. Dijkstra, *GO TO statement considered harmful*, CACM V, 11 No. 3

[36] E.W. Dijkstra, *Structured programming*, Nato Science Comittee - Software Engineering Techniques (1970)

[37] A. Donescu, *The simplex algorithm with the aid of the Gauss-Seidel method*, Econ. Comput. and Econ. Cybernetics Studies and Research, 24 (1989), pp. 127-132

[38] I.S. Duff and J.K. Reid, *The multifrontal solution of unsymmetric sets of linear equations*, SIAM J. Sci. Stat. Comp. 5 (1984), pp. 633-641

[39] I.S. Duff, A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford (1986)

[40] I.S. Duff, R.G. Grimes, and J.G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Soft. 15, 1 (1989), pp. 1-14

[41] I.S. Duff, *Parallel implementation of multifrontal schemes*, Parallel Computing, 3 (1986), pp. 193-204

[42] S.K. Eldersveld and M.A. Saunders, *A block-LU update for large-scale linear programming*, SIAM J. Matrix Anal. Appl. 12, 1 (1992), pp. 191-201

[43] D.J. Evans and G.M. Megson, *A systolic simplex algorithm*, Int. J. Comp. Math. 38 (1991), pp. 1-38

[44] A.V. Fiacco and G.P. Mc Cormick, *Nonlinear programming: Sequential unconstraint minimization techniques*, John Wiley & Sons, New York, 1968

[45] D.G. Firesmith, *Inheritance diagrams: which way is up*, JOOP 7, 1 (1994)

[46] J.J.H. Forest and J.A. Tomlin, *Updated triangular factors of the basis to maintain sparsity in the product form simplex method*, Math. Prog. 2 (1972, pp. 263-278

[47] J.J. Forest and D. Goldfarb, *Steepest-edge simplex algorithms for linear programming*, Math. Prog. , 57 (1992), pp. 341-374

[48] D.R. Fulkerson and P. Wolfe, *An algorithm for scaling matrices*, SIAM Review, 4 (1962), pp. 142-146

[49] A. George, M.T. Heath, J. Liu, and E. Ng, *Sparse cholesky factorization on a local-memory multiprocessor*, SIAM J. Sci. Stat. Comp. 9, 2 (1988), pp. 327-340

[50] P.E. Gill, W. Murray, M.A. Saunders and M.H. Wright, *A practical anti-cycling procedure for linear constraint optimization*, Math. Prog. 45 (1989), pp. 437-474

[51] S.K. Granendran and J.K. Ho, *Load balancing in the parallel optimization of block-angular linear programs*, Math. Prog. 62 (1993), pp. 41-67

[52] D. Goldfarb and J.K. Reid, *A practicable steepest-edge simplex algorithm*, Math. Prog. 12 (1977), pp. 361-371

[53] M. Grötschel, A. Löbel and M. Völker, *Optimierung des Fahrzeugumlaufs im öffentlichen Nahverkehr*, ZIB Preprint SC 96-8 (1996)

[54] M. Grötschel, L. Lovasz, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Algorithms and Combinatorics 2, Springer-Verlag (1988)

[55] M. Grötschel, *Optimierungsmethoden I*, Vorlesungsskriptum, Universität Augsburg (1985)

[56] J.L. Gustafson, *Reevaluating Amdahl's law*, Comm. ACM, Vol.31 No.5 (1988), pp. 532-533

[57] J.A.J. Hall and K.I.M. Mc Kinnon, *A class of cycling counter-examples to the EXPAND anti-cycling procedure*, Dep. of Math. and Stat., University of Edinburgh EH9 3JZ, UK (1995)

[58] J.A.J. Hall and K.I.M. Mc Kinnon, *An asynchronous parallel revised simplex algorithm*, Department of Math. and Stat. University of Edinburgh, MS 95-050 (1995)

[59] J.A.J. Hall and K.I.M. Mc Kinnon, *PARSMI, a parallel revised simplex algorithm incorporating minor iterations and Devex pricing*, PARA96: Workshop on Applied Parallel Computing in Industrial Problems and Optimization Copenhagen (1996)

[60] P.M.J. Harris, *Pivot selection methods for the Devex LP code*, Math. Prog. 5 (1973), pp. 1-28

[61] M. Heath, E. Ng, and B. Peyton, *Parallel algorithms for sparse linear systems*, SIAM J. Comp. 21, 1 (1991), pp. 111-139

[62] R.V. Helgason and J.L. Kennington, *A parallelization of the simplex method*, An. Op. Res. 14 (1988), pp. 17-40

[63] E. Hellerman and D. Rarick, *Reinversion with the preassigned pivot procedure*, Math. Prog. 1, 2 (1971), pp. 195-215

[64] J.K. Ho and R.P. Sundarraj, *A timing model for the revised simplex method*, Op. Res. Letters, 13 (1993), pp. 67-73

[65] A.J. Hoffman, *Linear programming at the National Bureau of Standards*, in „History of Mathematical Programming“, J.K. Lenstra, A.H.G. Rinnooy Kan and A. Schrijver eds., North-Holland Amsterdam, 1991

[66] M.T. Jones and P.E. Plassmann, *Scalable iterative solution of sparse linear systems*, Par. Comp. 20 (1994), pp. 753-773

[67] N.K. Karmarkar, *A new polynomial-time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373-395

[68] M. Katevenis, *RISC Architectures*, in „Parallel & Distributed Computing Handbook“, A.Y. Zomaya ed., McGraw-Hill (1996)

[69] L.G. Khachian, *A polynomial algorithm in linear programming*, Soviet Math. Doklady 20 (1979), pp. 191-194

[70] V. Klee and P. Kleinschmidt *The d-step conjecture and its relatives*, Math. of Op. Res. 12, 4 (1987), pp. 718-755

[71] H.W. Kuhn and R.E. Quandt, *An experimental study of the simplex method*, Proceedings of Symposia in Applied Mathematics, 15 (1963), pp. 107-124

[72] E.V. Krishnamurthy, *Complexity issues in parallel and distributed computing*, in „Parallel & Distributed Computing Handbook“, A.Y. Zomaya ed., McGraw-Hill (1996)

[73] D.W. Krumme, *Fast gossiping for the hypercube*, SIAM J. Comput. 21, 2 (1992), pp. 365-380

[74] A. Löbel, *Optimal vehicle scheduling in public transit*, PhD thesis, Technical University of Berlin, in preparation

[75] J. Luo, A.N.M. Hulsbosch and G.L. Reijns, *An MIMD work-station for large LP problems*, Par. Proc. and Appl., Proc. Int. Conf. , L'Aquila (1987)

[76] I.J. Lustig, R.E. Marsten and D.F. Shanno, *Interior point methods for linear programming: computational state of the art*, ORSA J. Comp. 6, 1 (1994), pp. 1-14

[77] H.M. Markowitz, *The elimination form of the inverse and its application to linear programming*, Management Sci. 3 (1957), pp. 255-269

[78] K. Margaritis and D.J. Evans, *Parallel systolic LU factorization for simplex updates*, in „Lecture Notes in Computer Science“, G. Goos and J. Hartmanis eds., Springer Verlag 1988

[79] I. Nassi and B. Schneiderman, *Flowchart techniques for structured programming*, SIGPLAN Notices (1973)

[80] M. Padberg, *Linear Programming and Extensions*, Algorithms and Combinatorics, Vol 12. Springer Verlag (1995)

[81] M. Padberg and M.P. Rijal, *Location, Scheduling, Design and Integer Programming*, Kluwers's International Series (1996)

[82] P.Q. Pan, *A Simplex-like method with bisection for linear programming*, Optimization, 22 (1991), pp. 717-743

[83] D.A. Reed and M.L. Patrick, *Parallel iterative solution of sparse linear systems: Models and architectures*, Par. Comp. 2 (1985), pp. 45-67

[84] J.K. Reid, *A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases*, Math. Prog. 24 (1982), pp. 55-69

[85] P. Sadayappan and S.K. Rao, *Communication reduction for distributed sparse matrix factorization on a processor mesh*, Supercomputing '89, ACM Press New York (1989), pp. 371-379

[86] M.A. Saunders, *A fast, stable implementation of the simplex method using Bartels-Golub updating*, in „Sparse Matrix Computations“, J. Bunch and D. Rose, eds., Academic Press, New York (1976), pp. 213-226

[87] D.F. Shanno, *Computational methods for linear programming*, Rutcor Research Report RRR 19-93 (1993)

[88] R. Sharda, *Linear programming solver software for personal computers: 1995 Report*, OR/MS Today, October 1995, pp. 49-57

[89] A.F. v.d.Stappen, R.H. Bisseling and J.G.G. v.d.Forst, *Parallel sparse LU decomposition on a mesh network of transputers*, SIAM J. Matrix Anal. Appl , Vol.14 No.3 (1993), pp. 853-879

[90] C.B. Stunkel and D.A. Reed, *Hypercube implementation of the simplex algorithm*, 3rd Conf. on Hypercube Concurrent Programming, Pasadena (1988), pp. 1473-1482

[91] C.B. Stunkel, *Linear optimization via message-based parallel processing*, Proc. Int. Conf. Par. Processing III (1988), pp. 264-271

[92] U.H. Suhl and L.M. Suhl, *Computing sparse LU factoriztions for large-scale linear programming bases*, ORSA J. Comp. 2, 4 (1990), pp. 325-335

[93] U.H. Suhl and L.M. Suhl, *A fast LU update for linear programming*, An. Oper. Res. 43, 1-4 (1993), pp. 33-47

[94] T. Terlaky and S. Zhang, *A survey on pivot rules for linear programming*, ISSN 0922-5641, Reports of the Faculty of Technical Mathematics and Informatics, Delft University of Technology, 19-99 (1991)

[95] R.P. Tewarson, *The product form of inverses of sparse matrices and graph theory*, SIAM Review, 9, 1 (1667), pp. 91-99

[96] J.A. Tomlin, *On scaling linear programming problems*, Math. Prog. Study, 4 (1975), pp. 146-166

[97] P. Tseng, *Distributed computation for linear programming problems satisfying a certain diagonal dominance condition*, Math. Op. Res. 15, 1 (1990), pp. 33-48

[98] R.A. Wagner, *Parallel solution of arbitrarily sparse linear systems*, Par. Comp. 9 (1988/89), pp. 313-331

[99] P. Wolfe, *A technique for resolving degeneracy in linear programming*, SIAM J. Appl. Math. 11 (1993), pp. 205-211

[100] P. Wolfe, *The composite simplex algorithm*, SIAM Review, 7, 1 (1965), pp. 42-54

[101] R. Wunderling, H.C. Hege, and M. Grammel, *On the impact of communication latencies on distributed sparse LU factorization*, ZIB Preprint SC 93-28, (1993)

[102] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Disc. Meth. 1 (1981), pp. 77-79

[103] S. Zhang, *On anti-cycling pivoting rules for the simplex method*, Op. Res. Letters 10 (1991), pp. 189-192

[104] G.M. Ziegler, *Lectures on Polytopes*, Graduate Texts in Mathematics 152, Springer Verlag (1995)

[105] X. Zhu, S. Zhang and A.G. Constantinides, *Lagrange neural networks for linear programming*, J. Par. Distr. Compt. 14 (1992), pp. 354-360

[106] *ZIB Jahresbericht 1995*

[107] anonymous ftp: `research.att.com:/dist/lpdata`

Index

Lebenslauf

Persönliche Angaben

Name:	Roland Wunderling
Geburtsdatum:	27. Oktober 1965
Geburtsort:	Berlin
Familienstand:	verheiratet
Staatsangehörigkeit:	deutsch

1971 - 1975	Grundschule an der Europäischen Schule in Varese (Italien)
1976 - 1979	Oberschule an der Europäischen Schule in Varese (Italien)
1979 - 1984	Droste-Hülshoff-Gymnasium in Berlin
SS 1985	Beginn des Diplomstudiengangs an der FU-Berlin in den Fächern Physik und Mathematik
Nov. 1987	Vordiplom in Mathematik
Nov. 1987	Vordiplom in Physik
Dez. 1991	Diplom in Physik
Jan. 1992	Wechsel zur Mathematik mit Studienziel Promotion an der TU-Berlin