Algorithmische Anwendungen

Lineare Programmierung – Simplex-Algorithmus

Studiengang: Allgemeine Informatik
7.Semester

Gruppe: A - blau
Sibel Cilek 11038325
Daniela Zielke 11036577

22.01.2006

Inhaltsverzeichnis

1 Einleitung

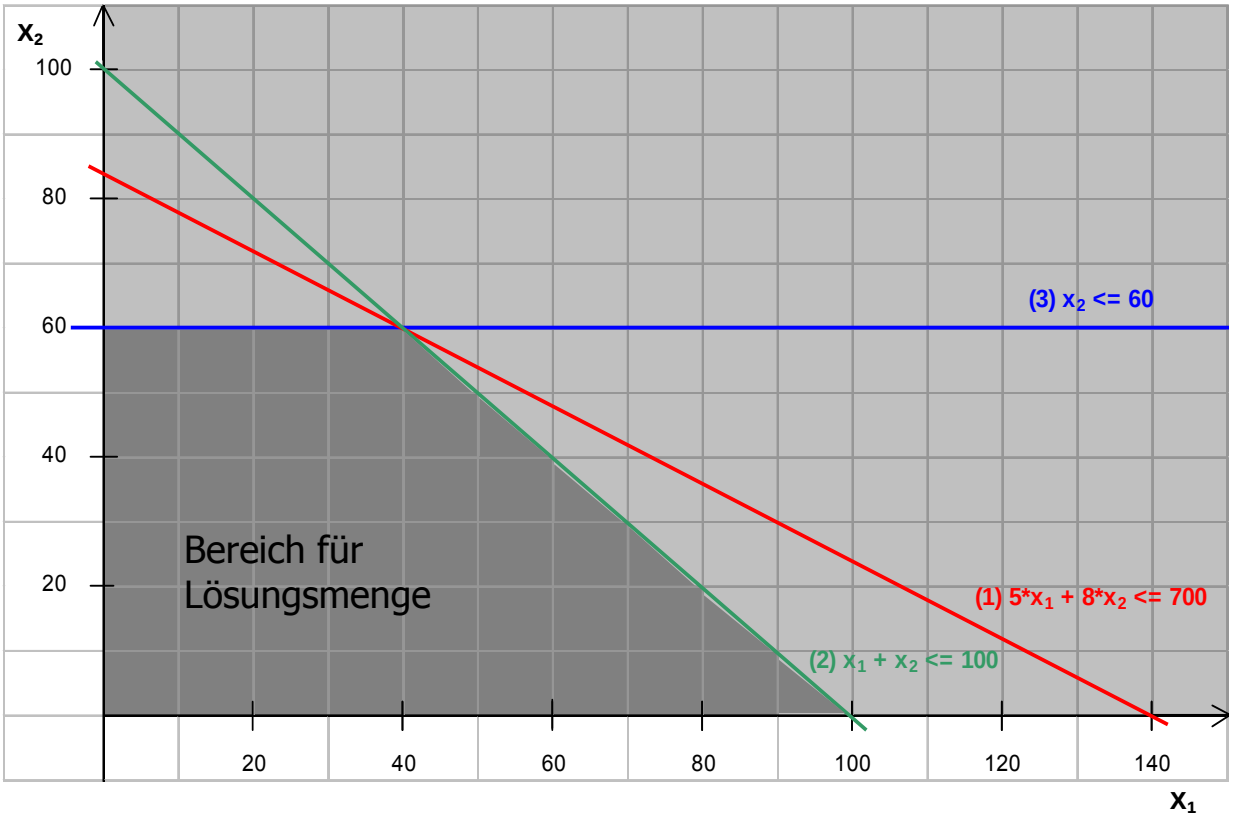
1.1 Was ist lineare Optimierung ?

Die lineare Optimierung ist ein Rechenverfahren, bei dem es bestimmte Bedingungen (in Form von Ungleichungen) gibt, die zur Berechnung einer Größe (Zielfunktion) eingehalten werden müssen. Von dieser Größe (Zielfunktion) wird entweder das Minimum oder das Maximum gesucht.
Die lineare Optimierung wird auch als lineare Programmierung bezeichnet. Im Operations Research, wo es um das Optimieren von Prozessen und Verfahren geht, ist dieser Bereich von zentraler Bedeutung.
Kleinere Probleme lassen sich relativ einfach berechnen. Je mehr Bedingungen und Variablen es gibt, desto höher der Rechenaufwand.

- **Beispielaufgabe aus der linearen Optimierung:**
  (von: http://www.matthias-priebe.de/start.php?template=projekte%2Fsimplex-index.php)

  Ein Hersteller kann kleine und große Packungen produzieren und will den maximalen Gewinn erzielen. An einer großen Packung verwendet er 2 €, an einer kleinen 1€.
  In der Produktion kosten die Packungen 8 bzw. 5 € und es steht ein Budget von 700 € für die Produktion bereit.
  Die Maschinen können bis zu 100 Packungen produzieren. Das Material für die kleine Packung steht unbegrenzt zur Verfügung, für die große genügt es für bis zu 60 Packungen.

Zielfunktion:	Maximiere $F = x_1 + 2*x_2$
Bedingungen:	(1) $5*x_1 + 8*x_2 <= 700$
	(2) $x_1 + x_2 <= 100$
	(3) $x_2 <= 60$



1.2 Anwendungsbeispiele

- **Transportproblem**
  - ⇒ Im Transportwesen geht es darum, dass umfangreich Transporte mit einem möglichst geringen Aufwand durchgeführt werden können.
  - ⇒ Bsp.: Transportpläne für Luftbrücke in Berlin 1948/49
  - ⇒ Bsp.: Amerikanische Telefongesellschaft AT&T (Bell Laboratories) → Kostengünstige Einrichtung der Telefonverbindungen zwischen den Städten

- **Produktionsplanung**
  - ⇒ In der Wirtschaft ist es für die Produktion wichtig, die vorhandenen Maschinen für die zu erstellenden Produkte optimal auszunutzen und den Gewinn für die Produkte zu maximieren.

- **Mischungsprobleme**
  - ⇒ In den Ernährungswissenschaften geht es darum, möglichst preiswerte Rationen mit einem vorgegebenen Gehalt von Nährstoffen herzustellen.
  - ⇒ Auch in der Pharmaindustrie sollen die Kosten für die Herstellung von Medikamenten kostengünstig sein und die dabei vorgegebenen Gehalte bestimmter Wirkstoffe müssen eingehalten werden.

- **Organisationsplanung**
  - ⇒ Schicht- und Flugbetriebsplanung bei American Airlines. Dabei ging es darum, den Einsatzpläne der Besatzungen der Flugzeuge so zu gestalten, dass die Effizienz und der Gewinn gesteigert werden und sich die Kosten reduzieren.

- **Graphentheoretische Probleme**
  - ⇒ In der Graphentheorie ist die Berechnung der kürzesten Pfade innerhalb eines Graphen von Bedeutung

- **Problem des Handlungsreisenden (Traveling Salesman Problem , TSP)**
  - ⇒ Möglichst schnell oder billig mehrere Orte hintereinander besuchen und wieder zum Ausgangsort zurückkehren
  - ⇒ Ziel: optimale Tour mit mininmalen Kosten

Abbildung 1: TSP-Problem aus „Lineare Programmierung“ von Frank Schönmann

1.3 Der Simplex-Algorithmus

Der Simplex-Algorithmus wurde 1947 von Georger B. Dantzig im Rahmen eines Forschungsauftrages der amerikanischen Luftwaffe erfunden. Dabei ging es um die Optimierung von militärischen Einsätzen.
Der Name dieses Algorithmus kommt daher, dass die Gleichungen des Problems ein Simplex (Polyeder) beschreiben, dessen Rand für das Auffinden einer Lösung beschritten wird. Jede Bedingung bildet bei n Variablen einen Halbraum des m-dimensionalen Raum. Der zulässige Bereich, der durch die Schnittmenge aller dieser Halbräume definiert wird, wird dann als Simplex bezeichnet.
Bis heute hat dieser Algorithmus eine große Bedeutung zum Lösen von Problemen in der linearen Programmierung.

- **Idee des Algorithmus:**

Abbildung 2: aus www.learn-line.nrw.de/ angebote/selma/foyer/projekte/hammproj1/

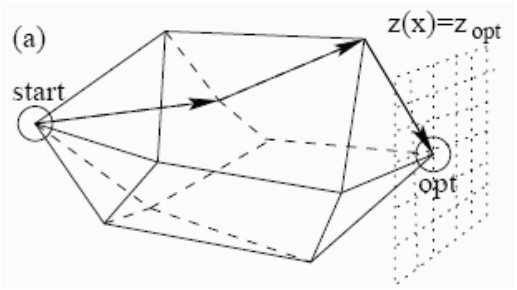


Abbildung 3: aus „Innere-Punkt-Methoden und automatische Ergebnisverifikation in der Linearen Programmierung“ von Matthias Hocks (TH Karlsruhe)

⇒ Der Algorithmus startet in einer zufällig ausgewählten Ecke .
⇒ In jeder Iteration bewegt sich der Algorithmus entlang einer Kante des Simplex von der aktuellen Ecke zu einer benachbarten Ecke. Der Wert dieser Ecke muss dann gleich oder in der Regel größer sein als der der aktuellen Ecke. Ist dies der Fall nähert man sich der optimalen Lösung.
⇒ Wenn es mehrere Ecken gibt, die in Frage kommen, wird aus diesen zufällig eine ausgewählt. Dafür gibt es Auswahlregeln.
⇒ Der Algorithmus wird solange wiederholt, bis die optimale Lösung gefunden wurde. Dies ist der Fall, wenn an einer Ecke alle benachbarten Ecken kleinere Werte haben und sich somit der Zielfunktionswert nicht mehr vergrößern kann.

1.4 Alternativen zum Simplex-Algorithmus

- **Problem des Simplex-Algorithmus**

  Je mehr Ecken zwischen der Startecke und der Zielecke zum Finden der Lösung ausgewählt werden müssen, desto langsamer wird der Algorithmus.
  Dazu tragen auch die Auswahlregeln bei. Wenn aus den zur Auswahl stehenden Ecken, die "falsche" ausgewählt wird, kann sich der Algorithmus verlängern.
  Dies führt im Worst-Case zu einer exponentiellen Laufzeit.

- **Verfahren von Karmarkar (Innere-Punkt-Methode)**

  1984 entwickelte der indische Mathematiker Narendar Karmarkar , während seiner Arbeit bei AT&T Bell Laboratories (amerikanische Telefongesellschaft), eine Methode, die schneller und effizienter arbeitet als der Simplex-Algorithmus.
  Das Verfahren wurde aber erst in den 90er Jahren populärer, als weitere Forscher auf Basis von Karmarkar Innere-Punkt-Algorithmen entwickelten.
  Das Verfahren baut auf die Ellipsoid-Methode (1979) von Leonid Khachiyan auf. Diese war ebenfalls effizienter, konnte jedoch praktisch nicht verwendet werden.

Dieser Algorithmus besitzt eine polynomiale Laufzeit. Das bedeutet, die Lösung des Problems hängt maximal polynomiell von der Problemgröße (Eingabelänge) ab. Dieser Algorithmus "wandert" durch das Innere des Zulässigkeitsbereichs, um eine optimale Lösung zu finden. Der Simplex-Algorithmus hingegen "wandert" am Rand des Polyeders von einer Ecke zu nächsten.
Dieser Algorithmus gilt als komplex, so dass in der Praxis oft bei der Lösung von kleineren Problemen noch der Simplex-Algorithmus eingesetzt wird.
Eine genauere Beschreibung dieses Verfahrens befindet sich im Kapitel 3.

- **Cutting-Plane-Methoden**

  Die Cutting-Plane-Methoden werden in Kombination mit den bekannten Verfahren wie dem Simplex-Algorithmus angewendet.

  Dabei wird das Gleichungssystem um ein cutting plane, eine Ungleichung erweitert, durch die Ecken aus dem möglichen Lösungsbereich abgeschnitten werden.

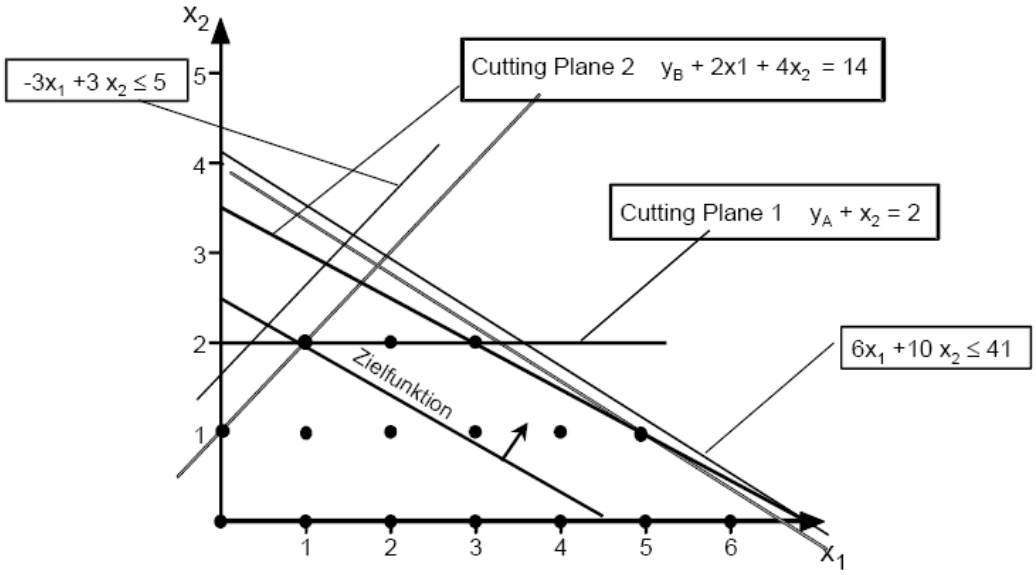


Abbildung 4: Beispiel für cutting plane aus http://www.bior.de/bior/lehre/vorles/or_winter/20042005/Folien_Download.pdf

1.5 Unsere Aufgabenstellung

Wir wollen in unserem Projekt folgende Dinge umsetzen:

- Laufzeitanalyse des Simplex-Algorithmus
  - ⇒ Vergleich mit der Laufzeit des Karmarkar-Verfahrens
- Simplex-Algorithmus am Beispiel einer kompletten Aufgabe
- Java-Programm zur Simulation des Simplex-Algorithmus
  - ⇒ GUI zur Eingabe der Daten
  - ⇒ Auswahl, ob Minimum oder Maximum gefunden werden soll
  - ⇒ Anzeigen der einzelnen Iterationsschritte bis zur optimalen Lösung

2 Rechnung mit Simplex-Algorithmus

In diesem Kapitel stellen wir das Rechnen mit dem Simplex-Algorithmus am in Kapitel 1 bereits vorgestellten Beispiel vor.

2.1 Gleichungssystem

Da man mit Gleichungen einfacher rechnen kann, werden die Ungleichungen der Bedingungen nun in Gleichungen überführt.
Dazu werden so genannte Schlupfvariablen x_3, x_4 und x_5 eingeführt, die z.B. ungenutzte Kapazitäten einer Maschine darstellen. Die Zielfunktion wird ebenfalls umgestellt.
Daraus ergibt sich für das Beispiel folgendes Gleichungssystem:

Zielfunktion: $F - 1*x_1 - 2*x_2 = 0$

Bedingungen: (1) $5*x_1 + 8*x_2 + \mathbf{1*x_3} = 700$
(2) $1*x_1 + 1*x_2 + \mathbf{1*x_4} = 100$
(3) $1*x_2 + \mathbf{1*x_5} = 60$

Für die Variablen x_1 und x_2 gibt es eine Grundbedingung, die **Nichtnegativitäts-bedingung**. Diese besagt, dass x_1 weder x_2 noch negativ sein dürfen. Wäre dies der Fall würden die Produkte „beseitigt" anstatt produziert werden.
Beim Simplex-Algorithmus geht man anfangs immer von einer Basislösung aus. Bei dieser Lösung sind x_1 und x_2 gleich 0 und es würde nichts produziert werden. Diese Basis-Lösung ist immer zulässig.

2.2 Simplex-Tableau

Als nächstes werden die Gleichungen nun in ein so genanntes Simplex-Tableau überführt.

- Die Variablen in der Kopfzeile(x_1, x_2) heißen Nichtbasisvariablen oder auch Problem-/Strukturvariablen
- Die Schlupfvariablen in der 1.Spalte werden als Basisvariablen bezeichnet.
- Die Zahlen in der Zeile der Zielfunktion heißen Zielfunktionskoeffizienten.
- Die Variablen b_1, b_2, b_3 bezeichnen die Werte der rechten Seite

	x_1	x_2	x_3	x_4	x_5	b
x_3	5	8	1			700
x_4	1	1		1		100
x_5	0	1			1	60
F	-1	-2	0	0	0	0

- Ziel ist es nun, in jedem Iterationsschritt eine Nichtbasisvariable gegen eine Basisvariable auszutauschen. Dieser Austauschschritt wird auch als Basisaustausch (**Pivotieren**) bezeichnet.

2.3 Varianten und Rechenregeln

Bevor nun ein Iteration beginnt, wird ein sogenanntes Pivot-Element gesucht.
Dies wird folgendermaßen bestimmt:

- Bei der Auswahl der Spalte ist es wichtig, dass sich der Wert der Zielfunktion auf jeden Fall vergrößert. Man sucht deshalb die Spalte, die den größtmöglichen Zuwachs ermöglicht.

- Für die Auswahl einer Spalte gibt es 2 Möglichkeiten:
  Greatest-Change-Methode:
  Spalte mit dem absolut größtem Produkt von negativem Zielfunktions-koeffizienten und kleinstem Quotienten aus rechter Seite und dem Element der entsprechenden Spalte.
  ⇒ Formel:

$$MAX = \frac{a_{0s} * b_j}{a_{rs}}$$

  Dabei entspricht a_{0s} dem negativen Zielfunktionskoeffizienten, b_j der rechten Seite und a_{rs} ist das Element der entsprechenden Spalte
  Steepest-Unit-Ascent-Methode:
  Spalte mit dem absolut größtem negativen Zielfunktionskoeffizienten
  ⇒ Wenn es mehrere Spalten gibt, die auf die Bedingungen zutreffen, muss eine ausgewählt werden (Auswahlregeln).
- Die ausgewählte Spalte wird dann als **Pivotspalte** bezeichnet.
- Innerhalb der ausgewählten Spalte wird dann die Zeile gesucht, wo der Koeffizient dieser Spalte größer Null ist.
  ⇒ Der Wert in der Zeile muss größer Null sein, da man sonst einen negativen Verbrauch für ein Produkt hätte, was wirtschaftlich sinnlos ist.
  ⇒ Wenn es mehrere Zeilen gibt, wird diejenige ausgewählt, bei der der Quotient aus rechter Seite und Koeffizient ein Minimum ergibt. Da alle Bedingungen für ein Produkt gelten, wäre dies die insgesamt größtmögliche Stückzahl, die vom Produkt produziert werden kann.
  ⇒ Die ausgewählte Zeile wird dann als **Pivotzeile** bezeichnet.

Durch Auswahl von Pivotspalte und –zeile erhält man das **Pivotelement**.

	x_1	x_2	x_3	x_4	x_5	b	
x_3	5	8	1			700	700/8 = 87,5
x_4	1	1		1		100	100/1 = 100
x_5	0	**1**			1	60	60/1 = 60
F	-1	-2	0	0	0	0	

Um einen Iterationsschritt durchzuführen, gibt es nun folgende Rechenregeln für die Elemente des Simplex-Tableaus:

Pivotelement	$a^*_{rs} = \frac{1}{a_{rs}}$
Pivotzeile	$a^*_{rj} = \frac{a_{rj}}{a_{rs}}$
Pivotspalte	$a^*_{is} = -\frac{a_{is}}{a_{rs}}$
restliche Elemente	$a^*_{ij} = a_{ij} - \frac{a_{rj} * a_{is}}{a_{rs}}$

r = Pivotzeile
s = Pivotspalte

Eine weitere Möglichkeit ist, im Simplex-Tableau wie in einem Gleichungssystem zu rechnen. Dabei wird versucht in allen Zeilen der Pivotspalte bis auf Pivotzeile, Nullen zu schaffen. Die Rechenregeln für das Pivotelement und die Pivotzeile bleiben bestehen.

2.4 Iterationen

Durch Anwenden der Regeln erhalten wir nun folgendes Simplex-Tableau:

	x_1	x_2	x_3	x_4	x_5	b	
x_3	5	0	1		-8	220	- 8*Pivotzeile
x_4	1	0		1		40	- Pivotzeile
x_2	0	**1**			1	60	
F	-1	0	0	0	0	120	+ 2*Pivotzeile

Da es noch einen weiteren negativen Zielkoeffizienten gibt, muss eine weitere Iteration durchgeführt werden. Erst wenn alle Zielkoeffizienten positiv sind, ist die optimale Lösung für das Problem gefunden worden. Ein Produkt mit negativem Gewinn würde wirtschaftlich keinen Sinn ergeben und nicht produziert werden.
Um die negativen Zielkoeffizienten wurde Zielfunktion anfangs umgeschrieben.

Am Ende hat man folgende optimale Lösung für die Rechnung:

	x_1	x_2	x_3	x_4	x_5	b
x_3	0	0	1	-5	-8	20
x_1	1	0		1		40
x_2	0	1			1	60
F	0	0	0	1	2	160

Mit x_1=40 und x_2=60 ergibt sich ein maximaler Gewinn von 160.

2.5 Minimierungs- statt Maximierungsproblem

Die Struktur eines Minimierungsproblems ist sehr ähnlich wie die eines Maximierungsproblems.
Es gibt Bedingungen, die als Ungleichungen dargestellt werden und eine Zielfunktion, die diesmal allerdings minimiert werden soll.
Dieses Problem ist auch mit dem Simplex-Algorithmus lösbar. Dazu muss das Minimierungsproblem in ein Maximierungsproblem transformiert werden.
Dies ist durch die sogenannte Dualkonversion möglich, die besagt, dass zu jedem Maximierungsproblem genau ein einziges Minimierungsproblem gehört.
Um aus der Minimierungsaufgabe eine Maximierungsaufgabe zu machen, müssen Zeilen und Spalten des Simplex-Tableaus vertauscht werden.

- **Beispielaufgabe:**
  (von: http://www.zingel.de/pdf/08sim.pdf)

  Um eine gegebene Anzahl von Hühnern zu füttern stehen einem Landwirt zwei verschiedene Arten von Hühnerfutter zur Verfügung, Sorte 1. und .Sorte 2.. Jede Sorte enthält Eiweiß, Fett und Kohlehydrate in unterschiedlichen Mengen pro Kilo, sowie eine bestimmte Menge Ballaststoffe. Um die Tiere gesund zu halten, sind pro Tag bestimmte Mindestmengen an Fett, Eiweiß und Kohlehydrate erforderlich:

	Sorte 1	Sorte 2	Mindest
Eiweiß:	100 g	200 g	1,0 kg
Fett:	200 g	100 g	0,8 kg
Kohlehydrate:	100 g	600 g	1,8 kg
Preis pro Kilo:	8,0 €	12,0 €	→ Min!

Abbildung 5: Tabelle aus http://www.zingel.de/pdf/08sim.pdf

⇒ Ziel: Minimierung der Kosten

Zielfunktion:	Minimiere F = $8*x_1 + 12*x_2$	x_1 →Sorte 1 x_2 → Sorte 2
Bedingungen:	(1) $0,1*x_1 + 0,2*x_2 >= 1,0$ (2) $0,2*x_1 + 0,1*x_2 >= 0,8$ (3) $0,1*x_1 + 0,6*x_2 >= 1,8$	Bedingungen 1 bis 3: Nährstoff-Anteil pro Produkt in einer Mindestmenge

⇒ Das Simplex-Tableau würde anfangs folgendermaßen aussehen:

Minimierung

Maximierung

	x_1	x_2	x_3	x_4	x_5	b
x_3	0,1	0,2	1			1,0
x_1	0,2	0,1		1		0,8
x_2	0,1	0,6			1	1,8
F	-8	-12	0	0	0	0

Im Simplex-Tableau versucht man in der Zielfunktionszeile immer die Werte bei einem Maximierungsproblem zu minimieren (Null-Werte für Nichtbasisvariablen), während man in der Spalte b versucht den Wert der Zielfunktion zu maximieren. Um nun ein Minimierungsproblem zu lösen, müssen Zeilen und Spalten getauscht werden.

⇒ Nach der Transformation würde das Simplex-Tableau folgendermaßen aussehen:

	x_3	x_4	x_5	x_1	x_2	b
x_1	0,1	0,2	0,1	1		8
x_2	0,2	0,1	0,6		1	12
F	-1	-0,8	-1,8	0	0	0

⇒ Nun kann das Simplex-Tableau genau wie beim Maximierungsproblem gelöst werden. Am Ende ergibt sich für das Beispiel dann folgende Lösung:

	x_3	x_4	x_5	x_1	x_2	b
x_1	0	1	-4/3	20/3	-10/3	40/3
x_2	1	0	11/3	-10/3	20/3	160/3
F	0	0	4/5	2	4	64

⇒ Durch die Transformation ist das Ergebnis diesmal unten abzulesen. $x_1=2$ und $x_2=4$ ergeben minimale Kosten von 64 €.

3 Verfahren von Karmarkar (Innere-Punkt-Methode)

3.1 Beschreibung

- Das 1984 von Karmakar entwickelte projektionsverfahren auch als Innere – Punkt Verfahren bekannt unterscheidet sich vom Simplex – Verfahren in der Hinsicht, daß sie sich nicht nur auf der Oberfläche des zulässigen Bereiches bewegt sondern Sprünge im Innern des zulässigen Bereiches macht.
- Es basiert auf einem linearen Optimierungsproblem der Form
  Minimiere/Maximiere $f(x) = c^{(T)} * x$
  unter den Nebenbedingungen $A * x = 0$,
  $e^{(T)} * x = 1$,
  $x \geq 0$,
  wobei A eine reelle (m*n) Matrix ist,
  c und $x \in R^n$,
  e ein n – dimensionaler Vektor mit 1 in allen Komponenten ist.
- Da das Verfahren von Karmarkar nicht der Standardform eines linearen Optimierungsproblems entspricht, zeigte Karmarkar, daß jedes lineare Optimierungsproblems in die Form des Karmarkar – Problem transformatiert werden kann.
  Der Algorithmus setzt für die erfolgreiche Verwendung des Verfahrens die Existenz eines strickt zulässigen Startpunktes x^0 voraus, d.h. eines Punktes, der echt im Inneren des zulässigen Bereiches vom Lösungsproblems liegt.
- Der Karmarkar – Algorithmus startet an einer Anfangsannäherung x^0 der Lösung von dem Karmarkar – Problem, für die
  $A * x^0 = 0$,
  $e^{(T)} * x^0 = 1$,
  und $x^0 > 0$ gilt.

- Der Algorithmus bewegt sich mit Hilfe von projektiven Transformationen zu einer neuen Näherung x^k und von x^k zu x^{k+1} bis zur optimalen Lösung ŷ.

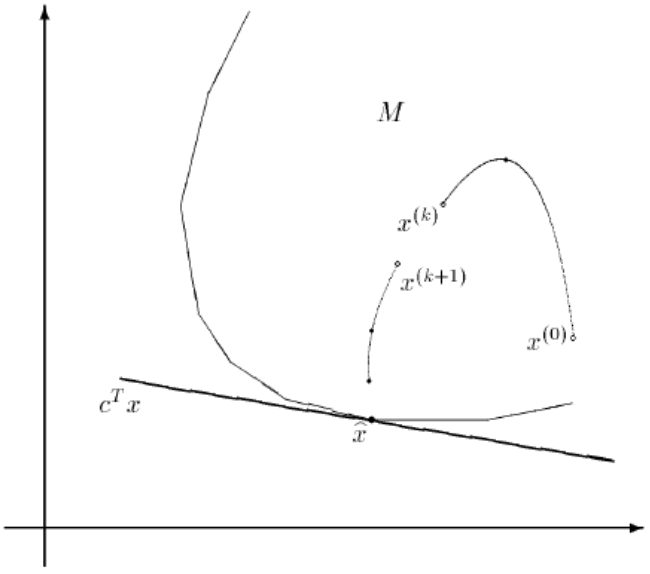


Abbildung 6: Graphische Darstellung des Karmarkar-Algorithmus aus „Innere-Punkt-Methoden und automatische Ergebnisverifikation in der Linearen Programmierung“

- **Geometrische Interpretation der Rechenschritte:**
  Karmarkar beruht auf der folgenden Idee:
  Angenommen man befinde sich an einem zulässigen Punkt, sagen wir x^k, dann sucht man eine Richtung (also einen Vektor $d \in \mathbb{R}^n$),bezüglich der man die Zielfunktion verbessert werden kann. Daraufhin bestimmt man eine Schrittlänge ρ, so dass man von x^k zum nächsten Punkt x^{k+1} := x^k + ρdgelangt. Der nächste Punkt x^{k+1}soll natürlich auch zulässig sein und einen "wesentlich" besseren Zielfunktionswert haben.
  Die Essenz eines jeden solchen Verfahrens steckt natürlich in der Wahl der Richtung und der Schrittlänge. Bei derartigen Verfahren tritt häufig die folgende Situation ein. Man ist in der Lage, eine sehr gute Richtung zu bestimmen (d. h. die Zielfunktion wird in Richtung d stark verbessert), aber man kann in Richtung d nur einen sehr kleinen Schritt ausführen, wenn man die zulässige Menge nicht verlassen will. Trotz guter Richtung kommt man also im Bezug auf eine tatsächliche Verbesserung kaum vorwärts und erhält unter Umständen global schlechtes Konvergenzverhalten.
  Man muss sich also bemühen, einen guten Kompromiss zwischen "Qualität der Richtung" und "mögliche Schrittlänge" zu finden, um insgesamt gute Fortschritte zu machen.
  Ist man — wie im vorliegenden Fall — im relativen Inneren der Menge P, aber nah am Rand und geht man z. B. in Richtung des Normalenvektors der Zielfunktion, so kann man sehr schnell an den Rand von P gelangen, ohne wirklich weiter gekommen zu sein, siehe Abbildung:

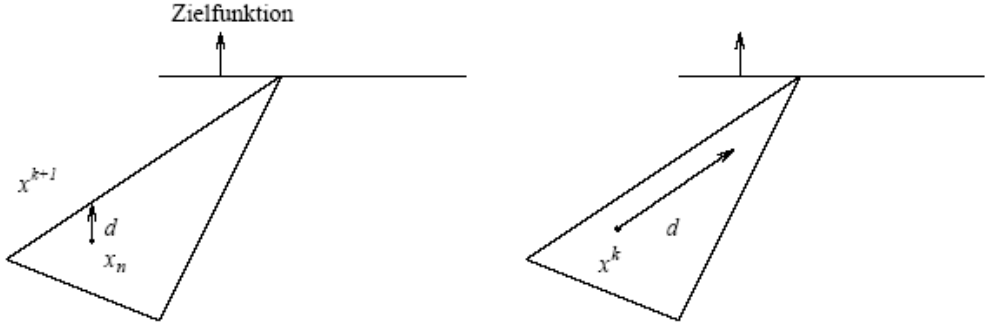


Abbildung 7: aus www.zib.de/groetschel/teaching/skriptADMII.pdf

Karmarkars Idee zur Lösung bzw. Umgehung dieser Schwierigkeit ist die folgende. Er führt eine projektive Transformation aus, die den Simplex Σ auf sich selbst, den affinen Teilraum Ω auf einen anderen affinen Teilraum Ω' abbildet und den relativ inneren Punkt x^k auf das Zentrum $\frac{1}{n}\mathbb{1}$ von Σ wirft. P wird dabei auf ein neues Polyeder P_k abgebildet. Offenbar kann man von $\frac{1}{n}\mathbb{1}$ aus recht große Schritte in alle zulässigen Richtungen machen, ohne sofort den zulässigen Bereich P_k zu verlassen.

Man bestimmt so durch Festlegung einer Richtung und einer Schrittlänge von $\frac{1}{n}\mathbb{1}$ ausgehend einen Punkt $y^{k+1} \in P_k$ und transformiert diesen zurück, um den nächsten (zulässigen) Iterationspunkt $x^{k+1} \in P$ zu erhalten.

3.2 Rechenschritte

Vorausgesetzt ist, daß ein zulässiger innerer Anfangspunkt $x = x^0 > 0$, welcher die Bedingung $A*x^{(0)} = b$ erfüllt, gegeben ist.

Die Schritte des Algorithmus sieht wie folgend aus:

1. Initialisiere den Zähler: Setze $t = 0$.
2. Erzeuge D: Setze D = Diagonale Matrix von $(x^{\wedge(t)})$.
3. Errechne zentrale Transformation: $A' = A*D$, $c' = D*c$.
4. Bestimme die Projekt – Matrix: Errechne $P' = I - A'^{\wedge T} * (A'*A'^{\wedge T})^{-1} * A'$.
5. Errechne die steilst – absteigende Richtung: Setze $p^{\wedge T} = -P' * c'$.
6. Setze $\Theta = - \min * p^{\wedge T}$
7. Teste für unbegrenztes Objekt: Wenn $\Theta \leq 0.0$ ist, melde daß das Objekt unbegrenzt ist und beende.
8. Erreiche $x'^{\wedge(t+1)}$: Errechne $x'^{\wedge(t+1)} = e + (\alpha/\Theta)*p^{\wedge t}$, wobei $e = (1,1,...,1)^{\wedge T}$ and α ist streng zwischen 0 und 1 liegt.
9. Errechne : $x^{\wedge(t+1)} = D* x'^{\wedge(t+1)}$.
10. Schlußuntersuchung: Wenn $x^{\wedge(t+1)}$ $x^{\wedge t}$ sehr nah ist, wird $x^{\wedge(t+1)}$ als optimale Beendigung markiert.
11. Setze $t = t+1$ an springe zum Schritt 2.

3.3 Teile eines Beispiels

Minimiere	$-2* x_1 - x_2$	$= z$
mit folgenden Nebenbedingungen:	$x_1 + x_2$	≤ 5
	$2* x_1 + 3* x_2$	≤ 12
wobei $x_1 \geq 0, x_2 \geq 0$.		

Ein zulässiger innerer Anfangspunkt mit folgenden Werten $x_1^o = 1$, $x_2^o = 2$ ist gegeben.

Man startet damit die (Un-)Gleichungen in die Standardform zu transformieren, in dem man die Schlupfvariablen hinzufügt:

Minimiere	$-2* x_1 - x_2$			$= z$
mit folgenden Nebenbedingungen:	$x_1 + x_2$	$+ x_3$		$= 5$
	$2* x_1 + 3* x_2$		$+x_4$	$= 12$
wobei $x \geq 0$ für $j = 1,...,4$.				

Der dazugehörige Startpunkt ist folgender: $x^0 = (1 \;\; 2 \;\; 2 \;\; 4)^{\wedge T}$
Der Wert von x^0 für $z = -4$.

Wir starten die 1.Iteration indem wir den Abgleichmatrix D erstellen:

$$\begin{pmatrix} 1 & & & \\ & 2 & & \\ & & 2 & \\ & & & 4 \end{pmatrix}$$

A' und c' werden berechnet:

$$A' = A * D = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 2 & 3 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & & & \\ & 2 & & \\ & & 2 & \\ & & & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 2 & 0 \\ 2 & 6 & 0 & 4 \end{pmatrix}$$

$$c' = c * D = \begin{pmatrix} -2 \\ -1 \\ 0 \\ 0 \end{pmatrix} * \begin{pmatrix} 1 & & & \\ & 2 & & \\ & & 2 & \\ & & & 4 \end{pmatrix} = \begin{pmatrix} -2 \\ -2 \\ 0 \\ 0 \end{pmatrix}$$

Als nächstes berechne die Projektionsmatrix P':

$$P' = I - A'^{T} * (A' * A'^{T}) * A'$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} 1 & 2 \\ 2 & 6 \\ 2 & 0 \\ 0 & 4 \end{pmatrix} * \left(\begin{pmatrix} 1 & 2 & 2 & 0 \\ 2 & 6 & 0 & 4 \end{pmatrix} * \begin{pmatrix} 1 & 2 \\ 2 & 6 \\ 2 & 0 \\ 0 & 4 \end{pmatrix} \right)^{-1} * \begin{pmatrix} 1 & 2 & 2 & 0 \\ 2 & 6 & 0 & 6 \end{pmatrix}$$

$$= \begin{pmatrix} 0.8831 & -0.2597 & -0.1818 & -0.0519 \\ -0.2597 & 0.3117 & -0.1818 & -0.3377 \\ -0.1818 & -0.1818 & 0.2727 & 0.3636 \\ -0.0519 & -0.3777 & 0.3636 & 0.5325 \end{pmatrix}$$

Im weiteren wird die Projektionsmatrix berechnet:

$$p^0 = -P' * c' = \begin{pmatrix} 0.8831 & -0.2597 & -0.1818 & -0.0519 \\ -0.2597 & 0.3117 & -0.1818 & -0.3377 \\ -0.1818 & -0.1818 & 0.2727 & 0.3636 \\ -0.0519 & -0.3777 & 0.3636 & 0.5325 \end{pmatrix} * \begin{pmatrix} -2 \\ -2 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1.2468 \\ 0.1039 \\ -0.7273 \\ -0.7792 \end{pmatrix}$$

Als nächstes bestimmen wir Θ:

$\Theta = -\min p^o = .7792.$

Hier fehlen noch aufgrund ihrer mathematischen Komplexität die Schritte 7-11.

4 Implementierung des Simplex-Algorithmus

In diesem Kapitel stellen wir die Implementierung des Simplex-Algorithmus anhand von Pseudocodes für die einzelnen Teile der Rechnung genauer vor. Vor der Ausführung des Algorithmus werden die entsprechenden Daten für die Berechnung in die folgenden Oberflächen eingegeben.

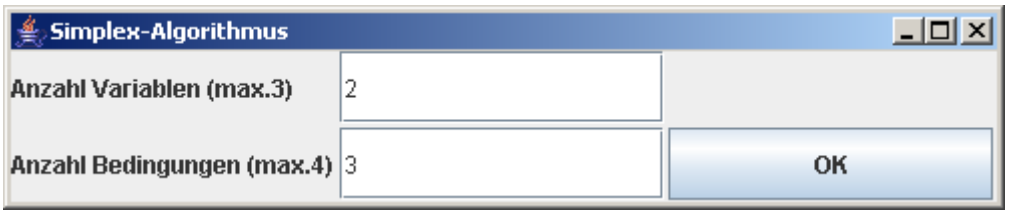


Abbildung 8: Eingabe von Anzahl Variablen und Bedingungen

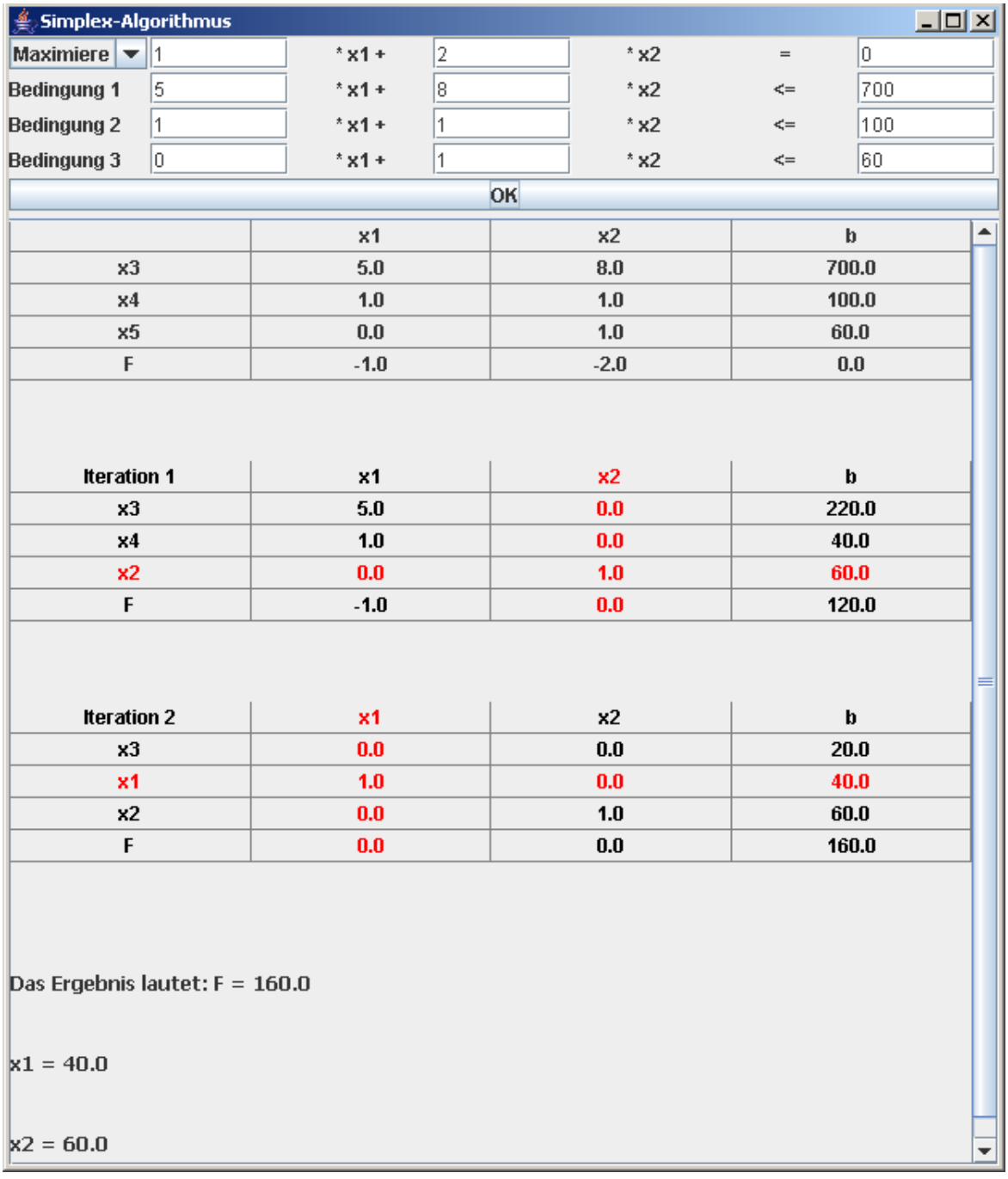


Abbildung 9: Eingabe der Gleichungen und Ausgabe der Tableaus

4.1 Pseudo-Code des gesamten Algorithmus

In diesem Kapitel haben wir den Pseudocode des gesamten Algorithmus. Die hervorgehobenen Funktionen werden dann im Weiteren genauer erläutert.

```
1   Programm simplex()
2      negativ ← true
3      if minProblem = 1 then
4         transformTableau()
5      while negativ = true do
6         findPivotElement()
7         berechneElemente()
8         AusgabeTableau()
9         negativ ← false
10        for j ← 0 to anzSpalten-1 do
11           if tableau[anzZeilen-1][j] < 0 then
12              negativ ← true
13  EndeProgramm
```

- Vor der Ausführung von `simplex()` werden die Daten zur Berechnung in der GUI eingegeben.
  Dabei einmal die Variable `minProblem` gesetzt. Sie erhält den Wert 1, wenn es sich bei der Berechnung um ein Minimierungsproblem handelt.
  Die Variable `anzSpalten` wird auf die Anzahl der Variablen, die angegeben wurde, plus 1 für die Ergebnisspalte b gesetzt.
  Die Variable `anzZeilen` wird auf die Anzahl der Bedingungen, die angegeben wurde, plus 1 für die Zielfunktion F gesetzt.
  In das Array `tableau[][]` werden die Werte der Gleichungen geschrieben, die in der GUI eingegeben wurden.
- In Zeile 3-4 wird die Variable `minProblem` überprüft. Wenn diese den Wert 1 hat, muss das Simplex-Tableau transformiert werden, so dass das Problem als Minimierungsproblem auszurechnen ist. Dies geschieht in der Funktion `transformTableau()`.
- Die Variable `negativ`, die in Zeile 2 mit true initialisiert wird, dient zur Überprüfung, ob es noch negative Zielfunktionskoeffizienten gibt. Dies geschieht in der for-Schleife in Zeile 10-12. Vor dieser Schleife wird `negativ` auf false gesetzt. Dann findet die Überprüfung statt. Wenn es noch mindestens einen negativen Zielfunktionskoeffizienten gibt, wird `negativ` wieder auf true gesetzt.
  Die Variable wird dann auch in der while-Schleife ab Zeile 5 abgefragt. Solange `negativ` true ist, wird eine weitere Iteration durchgeführt.
- Innerhalb der while-Schleife wird dann zuerst die Funktion `findPivotElement()`, die das Pivotelement sucht, aufgerufen, danach die Funktion `berechneElement()`, die die Berechnung der weiteren Element durchführt.
  Danach wird das neu ermittelte Tableau der Iteration dann auf der GUI wieder ausgegeben werden. Dafür werden das Tableau in eine Tabelle geschrieben, die dann ausgegeben wird. Die Werte der Pivot-Zeile und –Spalte der jeweiligen Iterationen werden in der Tabelle rot gefärbt.

4.2 Pseudo-Code „Transformation Minimierung“

```
1   transformTableau()
2      temp[][] ← tableau
3      tempAnz ← anzSpalten
4      anzSpalten ← anzZeilen
5      anzZeilen ← tempAnz
```

```
6      tableau ← new double[anzZeilen][anzSpalten+(tempAnz-1)]
7      for i ← 0 to anzZeilen-1 do
8         for j ← 0 to anzSpalten-1 do
9            if j >= anzZeilen then
10              tableau[i][j] ← 0
11           if i = anzZeilen-1 or j = anzSpalten-1 then
12              tableau[i][j] ← -temp[j][i]
13           else
14              tableau[i][j] ← temp[j][i]
15              tempZeilen ← anzZeilen
16              if anzZeilen = anzSpalten then
17                 tempZeilen--
18                 gleich ← true
19              tableau[i][anzSpalten+1]← tableau[i][tempZeilen]
20              tableau[i][tempZeilen] ← 0
21              if tempZeilen+i != anzSpalten+1 then
22                 tableau[i][tempZeilen+i] ← 1
23     anzSpalten ← anzSpalten + (tempAnz-1)
24  Ende transformTableau()
```

- In Zeile 2 wird zuerst ein neues zweidimensionales Array initialisiert, was als Hilfsvariable genötigt wird. In diesem Array wird das bisherige Tableau abgespeichert.
- Danach müssen in Zeile 3-5 die Werte für die Anzahl der Spalten und Zeilen vertauscht werden, da das Simplex-Tableau für die Berechnung eines Minimierungsproblem gedreht wird.
- Die Variable `tableau` wird dann auf die neuen Werte für `anzZeilen` und `anzSpalten` angepasst.
- In den verschachtelten for-Schleifen ab Zeile 7 findet dann das eigentliche Vertauschen von Spalten und Zeilen statt.
  Dabei muss noch beachtet werden, dass die unterste Zeile wieder auf negative Werte gesetzt wird, da dies bei den weiteren Berechnungen überprüft wird. Diese Überprüfung geschieht in der if-Anweisung in Zeile 11.
- Bei einem Maximierungsproblem haben wir die Schlupfvariablen bisher nicht beachtet. Dies muss jetzt aber beim Minimierungsproblem gemacht werden. Dafür gibt es die if-Anweisung in Zeile 9 sowie die Zuweisungen in Zeile 16-22,um die die Werte für die Schlupfvariablen zu setzen.

4.3 Pseudo-Code „Bestimme Pivot-Element“

```
1   findPivotElement()
2      maxWert ← 0
3      minWert ← 0
4      Div ← 0
5      for j ← 0 to anzSpalten-2 do
6         if tableau[anzZeilen-1][j] < maxWert then
7            maxWert ← tableau[anzZeilen-1][j]
8            pivotSpalte ← j
9      for i ← 0 to anzZeilen-2 do
10        Div ← tableau[i][anzSpalten-1]/tableau[i][pivotSpalte]
11        if Div < minWert or minWert = 0 then
12           minWert ← Div
13           pivotZeile ← i
14     pivotElement ← tableau[pivotZeile][pivotSpalte]
15  Ende findPivotElement()
```

- In Zeile 5-8 wird zuerst die Pivotspalte gesucht. Innerhalb der for-Schleife werden alle Spalten der Zielfunktion überprüft. Dabei wird der kleinste negative Wert gesucht. Da die Zielfunktion vor Beginn der Berechnung auf negative Werte umgestellt wurde, ist dies der größte Wert der Zielfunktion.
  Dieser wird dann in die Variable `maxWert` geschrieben. Die Variable wurde anfangs mit 0 initialisiert. In der if-Anweisung wird dann überprüft, ob es einen Wert in der Zielfunktionszeile gibt, der kleiner ist als `maxWert`. Dann erhält maxWert diesen Wert und die Spalte wird zur Pivotspalte.
- In der for-Schleife Zeile 9-13 wird dann die Pivotzeile gesucht. Dabei wird der Wert der rechten Seite der aktuellen geprüften Zeile durch den Wert in der Pivotspalte geteilt.
  In der if-Anweisung wird geprüft, ob es der Wert dieser Division kleiner ist als die Variable `minWert`. Die Variable wurde anfangs mit 0 initialisiert. Ist dies der Fall erhält `minWert` für die weiteren Durchläufe den Wert der Division und die Pivotzeile wird die aktuelle Zeile.
- Nach den beiden for-Schleifen wurden dann die Pivotspalte und –zeile gefunden und die Variable `pivotElement` kann auf das entsprechende Element des Tableaus gesetzt werden.

4.4 Pseudo-Code „Bestimme restliche Elemente“

```
1   berechneElemente()
2      for i ← 0 to anzZeilen-1 do
3         for j ← 0 to anzSpalten-1 do
4            if i != pivotZeile and j != pivotSpalte then
5               tableau[i][j] ← tableau[i][j]-
      ((tableau[i][pivotSpalte]*tableau[pivotZeile][j])/pivotElement)
6      for j ← 0 to anzSpalten-1 do
7            tableau[pivotZeile][j] ←
                     tableau[pivotZeile][j]/pivotElement
8      for i ← 0 to anzZeilen-1 do
9         if i != pivotZeile then
10           tableau[i][pivotSpalte] = 0
11  Ende berechneElemente()
```

- In der verschachtelten for-Schleife 2-5 werden zuerst alle Elemente außer denen in der in Pivotspalte und –zeile umgerechnet. Dies wird mit der in Kapitel 2.3 genannten Formel für die restlichen Elemente gemacht.
- In der for-Schleife Zeile 6-7 werden dann alle Elemente in der Pivotzeile umgerechnet. Dies geschieht mit der Formel aus Kapitel 2.3 für die Elemente der Pivotzeile.
- In der letzten for-Schleife Zeile 8-10 werden dann die Elemente aus der Pivotspalte, bis auf der Pivotelement selber, alle auf 0 gesetzt. Damit ist die Umrechnung des Simplex-Tableaus fertig.

5 Laufzeitanalyse des Simplex-Algorithmus

In diesem Kapitel haben wir eine Laufzeitanalyse am Pseudocode der verschiedenen Funktionen durchgeführt. Dazu werden wir eine asymptotische Laufzeitanalyse sowie anhand des Ratio Test eine experimentelle Laufzeitanalyse erstellen.
Für größere Werte bei der Eingabe gibt es einen Vergleich mit dem Karmarkar-Verfahren.

5.1 Vorbereitung

Für die Darstellung des Simplex-Tableaus wird ein m,n-elementiges Array benötigt.

- m→ Anzahl der Zeilen
  - ⇒ abhängig von der Anzahl der Bedingungen plus der Zielfunktion
- n → Anzahl der Spalten
  - ⇒ abhängig von der Anzahl der Basisvariablen plus der rechten Seite b
  - ⇒ Beim Minimierungsproblem gibt es zusätzlich noch für jeden Bedingung eine Schlupfvariable, so dass sich die Anzahl der Spalten auf n+m erhöht.
    Durch dieses Problem muss bei der Laufzeitanalyse zwischen Minimierung und Maximierungsproblem unterschieden werden.

5.2 Asymptotische Laufzeitanalyse

- **`starteRechnung()`**
  - ⇒ Der entscheidene Teil für die Laufzeit ist in dieser Funktion die while-Schleife für die Anzahl der Iterationen. Gleichzeitig ist es auch am schwersten von der Laufzeit her zu schätzen. Die Schleife wird durchlaufen, solange es noch negative Zielfunktionskoeffizienten in den Spalten der Basisvariablen gibt.
    Dies bedeutet bei einem Maximierungsproblem, dass die Durchläufe der Schleife abhängig ist von der Anzahl der Basisvariablen und somit wäre die Laufzeit der Schleife O(n).
    Bei einem Minimierungsproblem erhöht sich durch die Einbeziehung der Schlupfvariablen die Laufzeit der Schleife auf O(n+m).
  - ⇒ In unseren Quellen gibt es eine Reihe von Angaben zur möglichen Laufzeit der Iterationen. Laut Cormen beträgt zum Beispiel die Laufzeit $O\binom{n+m}{n}$.
  - ⇒ Desweiteren gibt es innerhalb der while-Schleife in Zeile 10-12 noch eine for-Schleife, die abhängig ist von der Anzahl der Spalten. Dadurch ergibt sich eine Laufzeit von O(n) bei einem Maximierungsproblem und O(n+m) bei einem Minmierungsproblem.
  - ⇒ Insgesamt hat die Funktion in Falle eines Maximierungsproblem eine Laufzeit von **$O(n^2)$** und bei einem Minimierungsproblem **$O((n+m)^2)$**.
- **`transformTableau()`**
  - ⇒ Diese Funktion ist nur für ein Minimierungsproblem zu beachten, da sie nur in dem Fall ausgeführt wird.
  - ⇒ Innerhalb dieser Funktion gibt es zwei verschachtelte for-Schleifen. Die eine for-Schleife ist abhängig von der Anzahl der Zeilen, die andere von der Anzahl der Spalten.
  - ⇒ Dadurch ergibt sich für diese Funktion eine Laufzeit von **$O(n*m)$**.

- **`findePivotElement()`**
  - ⇒ In dieser Funktion gibt es in Zeile 5-8 einmal eine for-Schleife, die abhängig ist von der Anzahl der Spalten. Allerdings wird die letzte Spalte (rechte Seite b der Gleichungen) nicht berücksichtigt.
  - ⇒ In Zeile 9-13 gibt es eine weitere for-Schleife, die diesmal von der Anzahl der Zeilen abhängig ist. Die letzte Zeile (Zielfunktion) wird in der Schleife nicht berücksichtigt.
  - ⇒ Für diese Funktion ergibt sich bei einem Maximierungsproblem dadurch eine Laufzeit von **O((n-1)+(m-1))** und bei einem Minimierungsproblem von **O((n+m-1)+(m-1))**.
- **`berechneElemente()`**
  - ⇒ Im ersten Teil dieser Funktion gibt es zwei verschachtelte for-Schleifen. Die äußere Schleife ist abhängig von der Anzahl der Zeilen während die innere Schleife von der Anzahl der Spalten abhängig ist.
  - ⇒ Desweiteren gibt es in Zeile 6-7 eine for-Schleife, die für alle Spalten durchlaufen wird.
  - ⇒ In Zeile 8-10 gibt es noch eine weitere for-Schleife, die abhängig ist von der Anzahl der Zeilen.
  - ⇒ Bei einem Maximierungsproblem haben wir bei dieser Funktion so eine Laufzeit von **O(n*m+n+m)** und bei einem Minimierungsproblem eine Laufzeit von **O((n+m)*m+(n+m)+m)**.

- **Gesamt-Laufzeit**
  Für den gesamten Algorithmus ergeben sich für die jeweiligen Probleme folgende Laufzeiten:
  - ⇒ Maximierungsproblem:

n * (	n +	(n-1)+(m-1) +	(n*m+n+m))
while-Schleife	for-Schleife	findePivotElement	berechneElemente

  Da die Funktionen findePivotElement() und berechneElemente() innerhalb der while-Schleife in starteRechnung() aufgerufen werden, muss die Laufzeit für die while-Schleife noch mit den Laufzeiten der Funktionen multipliziert werden. Nach dem Ausklammern haben wir somit folgende Laufzeit für ein Maximierungsproblem:
  $O(2n^2 + n^2 - 2n + 2nm + n^2m)$

  - ⇒ Minimierungsproblem

(n*m) +	(n+m) * (	(n+m) +	((n+m)-1)+(m-1))+	((n+m)*m+(n+m)+m)
transformTableau	while-Schleife	for-Schleife	findePivotElement	berechneElemente

  Genau wie beim Maximierungsproblem werden die Funktionen findePivotElement() und berechneElemente() innerhalb der while-Schleife in starteRechnung() aufgerufen, so dass die Laufzeit für die while-Schleife noch mit den Laufzeiten der Funktionen multipliziert werden müssen. Die Transformation des Tableaus findet ausserhalb der while-Schleife statt.
  Bei einem Minimierungsproblem ergibt sich somit eine Laufzeit von:
  $O(nm + 3(n+m)^2 - 2(n+m) + 2(nm+m^2) + (n+m)^2m)$

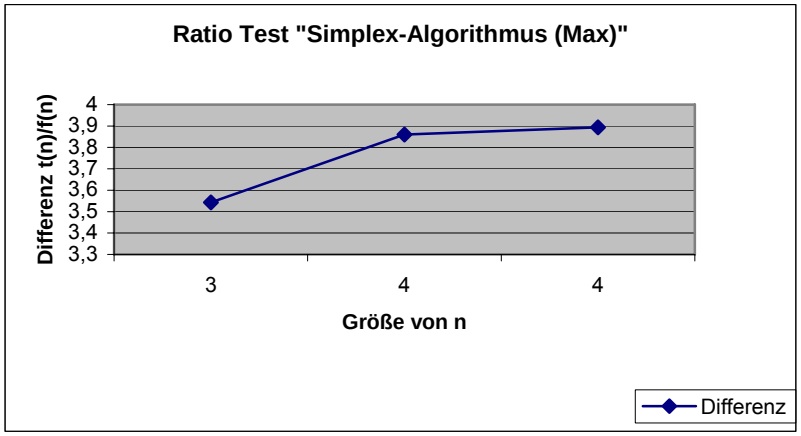
5.3 Experimentelle Laufzeitanalyse

Für die experimentelle Laufzeitanalyse wird ein Ratio Test durchgeführt.

- Zum Zählen der Elementaroperationen wurde in den Algorithmus ein Counter eingefügt (siehe Quellcode der Klasse „SimplexRatio" in den Anlagen).
- Da die Aufgaben, die wir zum Testen gefunden haben, von der Anzahl der Variablen und Bedingungen sehr gering ist, haben wir nur einen relativ kleinen Bereich für den Ratio Test.
- Der Power Test liefert aufgrund der Ergebnisse kein repräsentatives Ergebnis.
- **Ergebnis für Ratio Test bei einem Maximierungsproblem:**

Größe von n	3	4	4
Größe von m	4	4	5
Anzahl der Operationen	287	525	623
Schätzung	81	136	160
Differenz	3,5432	3,86029	3,89375

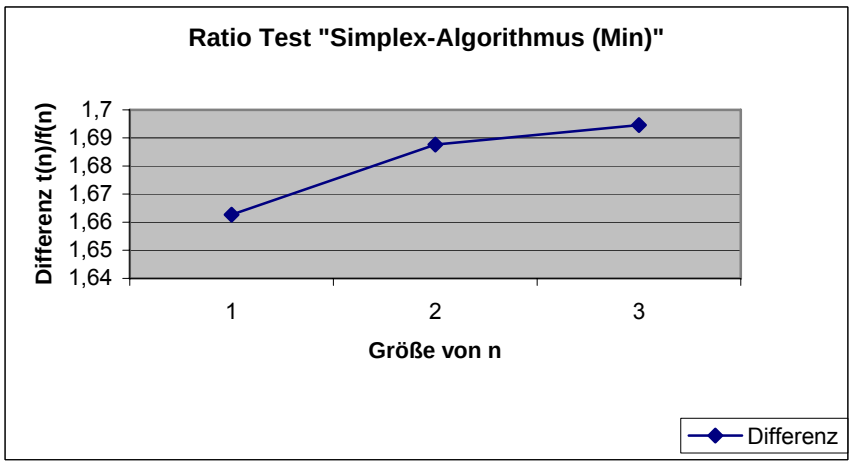
Schätzung der Laufzeit **n * (n + (n-1)+(m-1) + (n*m+n+m))**



- **Ergebnis für Ratio Test bei einem Minimierungsproblem:**

Größe von n	3	3	4
Größe von m	3	4	5
Anzahl der Operationen	414	670	1254
Schätzung	249	397	740
Differenz	1,6627	1,68766	1,69459

Schätzung der Laufzeit **(n*m) + (n+m) * ((n+m) + ((n+m)-1)+(m-1) + ((n+m)*m+(n+m)+m))**



- Um Verhalten des Simplex-Algorithmus bei größeren Eingabemengen im Vergleich zum Karmarkar-Verfahren haben wir folgende Tabelle:

Vergleichkriterium	Simplex	Karmarkar
Autor	G. Dantzig (1947)	N. Karmarkar (1984)
Aufwand bzgl. Problem-dimension n	Durchschnitt	Durchschnitt
$n < 100$	$O(3n)$	$O(K)$
$100 \leq n < 10.000$	$O(n^2)$	$O(K)$
$n \geq 10.000$	$O(n^4)$	$O(n^3 K)$
worst case	$O(2^n)$	$O(n^{3.5} K)$
Konvergenz	i.a. garantiert	asymptotisch
behandelbare Problemklassen	linear	linear, (nicht-linear) konvex, kombinatorisch

Abbildung 10: Laufzeitvergleich aus „Innere-Punkt-Methoden und automatische Ergebnisverifikation in der Linearen Programmierung“

⇒ Das K in der Laufzeitfunktion von Karmarkar entspricht der Anzahl der Iterationen, die benötigt werden, um das Problem zu lösen.

⇒ An dieser Tabelle sieht man, dass der Simplex-Algorithmus für kleinere Probleme (n < 100) einen größeren Vorteil. Laut „Innere-Punkt-Methoden und automatische Ergebnisverifikation in der Linearen Programmierung“ fällt die Größe K für die Anzahl der Iterationen bei Karmarkar in diesen Fällen meistens grösser aus.

⇒ Erst bei den größeren Problemdimensionen hat das Karmarkar-Verfahren seine Vorteile in der Laufzeit.

⇒ Im worst-case-Fall ist die Laufzeit des Simplex-Algorithmus exponentiell abhängig von der Problemgröße.

6 Aufgabensammlung

Zum Testen des jar-Files haben wir eine Aufgabensammlung zusammengestellt, mit Aufgaben, die wir auch zum Testen verwendet haben.

- **Aufgabe 1:**

  Maximiere $F = 1*x_1 + 2*x_2$
  Bedingungen: (1) $5*x_1 + 8*x_2 <= 700$
  (2) $1*x_1 + 1*x_2 <= 100$
  (3) $0*x_1 + 1*x_2 <= 60$

  ⇒ Lösung: $x_1 = 40$, $x_2 = 60$ und $F = 160$

- **Aufgabe 2:**

  Minimiere $F = 8*x_1 + 12*x_2$
  Bedingungen: (1) $0{,}1*x_1 + 0{,}2*x_2 >= 1{,}0$
  (2) $0{,}2*x_1 + 0{,}1*x_2 >= 0{,}8$
  (3) $0{,}1*x_1 + 0{,}6*x_2 >= 1{,}8$

  ⇒ Lösung: $x_1 = 2$, $x_2 = 4$ und $F = 64$

- **Aufgabe 3:**

  Minimiere $F = 1*x_1 + 1*x_2 = 2$
  Bedingungen: (1) $1*x_1 + 1*x_2 <= 5$
  (2) $2*x_1 + 3*x_2 <= 12$

  ⇒ Lösung: $x_1 = 3$, $x_2 = 2$ und $F = 7$

- **Aufgabe 4:**

  Maximiere $F = 3*x_1 + 1*x_2 + 2*x_3$
  Bedingungen: (1) $1*x_1 + 1*x_2 + 3*x_3 <= 30$
  (2) $2*x_1 + 2*x_2 + 5*x_3 <= 24$
  (3) $4*x_1 + 1*x_2 + 2*x_3 <= 36$

  ⇒ Lösung: $x_1 = 8$, $x_2 = 4$, $x_3 = 0$ und $F = 28$

- **Aufgabe 5:**

  Maximiere $F = 15*x_1 + 20*x_2 + 130*x_3$
  Bedingungen: (1) $5*x_1 + 0*x_2 + 0*x_3 <= 2$
  (2) $10*x_1 + 10*x_2 + 0*x_3 <= 3$
  (3) $0*x_1 + 20*x_2 + 20*x_3 <= 4$
  (4) $0*x_1 + 0*x_2 + 50*x_3 <= 5$

  ⇒ Lösung: $x_1 = 0.2$, $x_2 = 0.1$, $x_3 = 0.1$ und $F = 18$

7 Literatur

- George B. Dantzig, Mukund N. Thapa: Linear Programming (Springer Series in Operations Research)
- www.learn-line.nrw.de/angebote/selma/foyer/projekte/hammproj1/rund.htm
- de.wikipedia.org/wiki/Simplex-Verfahren
- Ausarbeitung „Der Simplex-Algorithmus" von Sandip Sar-Dessai (RWTH Aachen)
- „Lineare Optimierung – Ergänzungen" von Prof. Dr. Huhn (TU Clausthal)
- „Der Algorithmus von Karmarkar - Idee, Realisation, Beispiel und numerische Erfahrungen" von Alfred Schönlein
- "How I implemented the Karmarkar Algorithm in One Evening" von E.R. Swart
- „Lineare Programmierung" von Frank Schönmann
- Dissertation „Innere-Punkt-Methoden und automatische Ergebnisverifikation in der Linearen Programmierung" von Matthias Hocks (TH Karlsruhe)
- http://www.zib.de/groetschel/teaching/skriptADMII.pdf
- http://www.zingel.de/pdf/08sim.pdf
- http://www.bior.de/bior/lehre/vorles/or_winter/20042005/Folien_Download.pdf
- Algorithmen – Eine Einführung von Cormen, Leiserson, Rivest, Stein