Der Buchberger-Algorithmus für torische Ideale und seine Anwendung in der ganzzahligen Optimierung

Christine Theis

Diplomarbeit
betreut von Prof. Dr. W. Decker

Fachbereich 9 – Mathematik

Universität des Saarlandes,
Saarbrücken

1999

Inhaltsverzeichnis

Einführung

Gröbnerbasen und ihre Berechnung mit Hilfe des Buchberger-Algorithmus bilden ein grundlegendes Konzept der Computeralgebra. Die elementaren Ideen haben schon viele Algorithmen hervorgebracht, die theoretische Konstruktionen der Algebra für konkrete Beispiele ausrechnen können. Doch damit scheint das Spektrum der Anwendungsmöglichkeiten noch lange nicht erfaßt zu sein; in zunehmendem Maße entdeckt man die Gröbnerbasen als Hilfsmittel für kombinatorische Probleme.

Beispielhaft dafür steht der 1991 veröffentlichte Artikel von Pasqualina Conti und Carlo Traverso ([7]), in dem beschrieben wird, wie Probleme der linearen ganzzahligen Optimierung mit dem Buchberger-Algorithmus gelöst werden können. In der Folge entstanden mehrere Algorithmen ([25], [12], [16], [3]), die die Ideen von Conti und Traverso weiterentwickeln. Diese Algorithmen bilden die Grundlage für die vorliegende Arbeit.

Es sei aber auch auf weitere Forschungsgegenstände verwiesen, auf die an dieser Stelle nicht näher eingegeangen wird. Das Buch „Gröbner Bases and Convex Polytopes“ von Bernd Sturmfels ([30]) enthält abwechslungsreiches Material über die Rolle der Gröbnerbasen in der polyedrischen Geometrie. Die Veröffentlichungen [17], [18], [31], [32] und [33] erörtern weiterführende Ideen zum Themenkreis Gröbnerbasen und ganzahlige Optimierung.

Aufbau der Arbeit

Kapitel 1 gibt eine Einführung in die benötigten Begriffe aus der Computeralgebra. Die Darstellung folgt im wesentlichen der in [9] und [11], ist aber in vielen Punkten vereinfacht und stark verkürzt. Schwerpunkte bilden die gewichteten monomialen Ordnungen sowie praktische Verbesserungen des Buchberger-Algorithmus. Der letzte Abschnitt macht einen Ausflug in die Theorie der Gitter; Gitter werden später von den meisten Optimierungsalgorithmen benötigt.

Der erste Teil von Kapitel 2 stellt die Problematik der ganzzahligen Optimierung vor, unter Einschluß von Beispielen, die Anschauungs- und Testmaterial liefern. Danach erfolgt die Übersetzung des ganzzahligen Optimierungsproblems in die Sprache der Computeralgebra; dieser Teil erklärt die Grundidee aller folgenden Algorithmen und ist wesentlich für das Verständnis. Nach einen Einschub, der sich mit linearer Optimierung und dem Simplex-Algorithmus befaßt, entwickeln wir einen vollständigen generischen Algorithmus zur Lösung ganzzahliger Optimierungsprobleme.

Kapitel 3 beschreibt verschiedene Algorithmen, die auf der Übersetzung aus

Kapitel 2 basieren. Ein besonderes Anliegen besteht darin, die uneinheitlichen Darstellungen der Originalliteratur in eine gemeinsame Sprache zu fassen.

Kapitel 4 stellt Techniken vor, die den Buchberger-Algorithmus auf die speziellen Ideale zuschneiden, die man für die ganzzahlige Optimierung braucht.

Kapitel 5 beschreibt eine konkrete Implementierung. Es dient zusammen mit dem Anhang als Programmdokumentation, beleuchtet aber auch alternative Programmiermöglichkeiten.

Kapitel 6 liefert schließlich eine Auswahl an Testmaterial. Verschiedene Ausprägungen des Buchberger-Algorithmus werden miteinander verglichen, ebenso die Laufzeiten der implementierten Algorithmen. Der Vergleich mit dem Computeralgebrasystem SINGULAR demonstriert die Wirkung der Methoden aus Kapitel 4. Die Praxistauglichkeit der vorgestellten Verfahren wird anhand eines Vergleichs mit professioneller IP-Software erörtert.

Danksagung

Ich bedanke mich bei Prof. Dr. Decker für das schöne Thema und die Freiheit bei der Gestaltung der Arbeit, die meine Interessen gut getroffen hat.

Vielen Dank an Serkan Hoşten und Mike Stillman für ihre hilfreichen Kommentare und Anregungen, an Olaf Bachmann und Hans Schönemann für die Anbindung an das Computeralgebrasystem SINGULAR sowie an Maximal Software Inc. für die kostenlose Bereitstellung der Studentenversion von MPL.

Mein Dank geht auch an meine Eltern und Freunde für die moralische Unterstützung und ganz besonders an Klaus für seine Hilfe bei allen technischen Problemen.

Kapitel 1

Grundlagen aus der Algebra

1.1 Polynome

$\mathbb{Z}$ bezeichne wie üblich die Menge der ganzen Zahlen, $\mathbb{N}$ die Menge der nichtnegativen ganzen Zahlen (inklusive 0), $\mathbb{R}$ die Menge der reellen Zahlen. Mit $\mathbb{R}_{\geq 0}$ bzw. $\mathbb{R}_{>0}$ werden die nichtnegativen bzw. positiven reellen Zahlen bezeichnet.

Sind $x_1, \ldots, x_n$ Variablen und $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$, so schreibt man:

$$x^\alpha := x_1^{\alpha_1} \cdot \ldots \cdot x_n^{\alpha_n}$$

$$|\alpha| := \alpha_1 + \ldots + \alpha_n$$

Definition 1.1 Sei K ein Körper. Ein formaler Ausdruck der Form

$$f = \sum_{\alpha \in \mathbb{N}^n} a_\alpha x^\alpha \tag{1.1}$$

mit $a_\alpha \in K,\ a_\alpha \neq 0$ nur für endlich viele α, heißt ein **Polynom** in den Variablen $x_1, \ldots, x_n$ mit **Koeffizienten** $a_\alpha \in K$.
Der **Totalgrad** von f ist definiert als

$$\deg(f) := \begin{cases} \max\{|\alpha| : a_\alpha \neq 0\}, & \text{falls } f \neq 0, \\ -\infty, & \text{falls } f = 0. \end{cases}$$

Ein Polynom der Form

- $ax^\alpha,\ a \neq 0$, heißt **Term,**
- x^α heißt **Monom,**
- $ax^\alpha - bx^\beta,\ a, b \neq 0$, heißt **Binom.**

Die Menge aller Polynome in $x_1, \ldots, x_n$ mit Koeffizienten in K bildet mit den üblichen Verknüpfungen

$$\sum_\alpha a_\alpha x^\alpha + \sum_\alpha b_\alpha x^\alpha := \sum_\alpha (a_\alpha + b_\alpha) x^\alpha,$$

$$\left(\sum_\alpha a_\alpha x^\alpha\right) \cdot \left(\sum_\beta b_\beta x^\beta\right) := \sum_\gamma \left(\sum_{\alpha+\beta=\gamma} a_\alpha b_\beta\right) x^\gamma,$$

$$\lambda \cdot \left(\sum_\alpha a_\alpha x^\alpha\right) := \sum_\alpha (\lambda a_\alpha) x^\alpha, \ \lambda \in K,$$

eine kommutative Algebra mit $1 = 1_K$, den **Polynomring** $K[x_1, \ldots, x_n]$. □

In den Anwendungen der folgenden Kapitel werden stets Binome der Form $x^\alpha - x^\beta$ betrachtet. Insbesondere treten nur die Koeffizienten 1 und -1 auf; der Körper K ist dann bedeutungslos und wird auch nicht spezifiziert.

Bemerkung 1.2 $K[x_1, \ldots, x_n]$ besitzt keine Nullteiler, ist also **Integritätsring.** □

Satz 1.3 (Hilbertscher Basissatz) *$K[x_1, \ldots, x_n]$ ist* **noethersch,** *d.h. jedes Ideal in $K[x_1, \ldots, x_n]$ ist endlich erzeugt.*

Beweis: Kapitel 2, §5, Theorem 4 in [9]. □

Das Ideal I, das von den Polynomen $f_1, \ldots, f_s$ erzeugt wird, schreibt man als

$$I = \langle f_1, \ldots, f_s \rangle.$$

1.2 Monomiale Ordnungen

Definition 1.4 Sei $\mathcal{X}$ eine Menge. Eine Relation $\Gamma \subseteq \mathcal{X} \times \mathcal{X}$ heißt **teilweise** oder **partielle Ordnung** auf $\mathcal{X}$, wenn gilt:

1. $(x, x) \in \Gamma \ \ \forall\, x \in \mathcal{X}$ **(Reflexivität)**,
2. $(x, y) \in \Gamma \wedge (y, x) \in \Gamma \ \Rightarrow \ x = y$,
3. $(x, y) \in \Gamma \wedge (y, z) \in \Gamma \ \Rightarrow \ (x, z) \in \Gamma$ **(Transitivität)**,

Man schreibt dann $x \succeq y$ oder $y \preceq x$, falls $(x, y) \in \Gamma$, und $x \succ y$ oder $y \prec x$, falls $(x, y) \in \Gamma$ und $x \neq y$, und bezeichnet Γ mit $\succ$.
Eine teilweise Ordnung $\succ_1$ heißt **feiner** als eine teilweise Ordnung $\succ_2$, wenn gilt:

$$x \succ_2 y \ \Rightarrow \ x \succ_1 y$$

Eine teilweise Ordnung $\succ$ heißt **totale Ordnung,** wenn sich je zwei Elemente aus $\mathcal{X}$ mit $\succ$ vergleichen lassen. □

Beipiel 1.5 Teilbarkeit definiert eine partielle Ordnung auf der Menge der Monome in $K[x_1, \ldots, x_n]$, die der üblichen teilweisen Ordnung $<$ auf $\mathbb{N}^n$ entspricht:

$$x^\alpha \mid x^\beta \ \Leftrightarrow \ \alpha \leq \beta \ :\Leftrightarrow \ \alpha_i \leq \beta_i \ \ \forall\, i = 1, \ldots, n \qquad □$$

Definition 1.6 Eine **monomiale Ordnung** oder **Termordnung** auf dem Polynomring $K[x_1, \ldots, x_n]$ ist eine totale Ordnung $\succ$ auf der Menge der Monome in $K[x_1, \ldots, x_n]$ mit den Eigenschaften:

1. $\succ$ verfeinert die durch Teilbarkeit definierte teilweise Ordnung, d.h. für Monome $m_1 \neq m_2$ gilt:
$$m_1 \mid m_2 \;\Rightarrow\; m_2 \succ m_1$$

2. $\succ$ ist **multiplikativ,** d.h. für Monome $m_1, m_2, n \neq 0$ gilt:
$$m_1 \succ m_2 \;\Rightarrow\; nm_1 \succ nm_2 \qquad \square$$

Eine monomiale Ordnung $\succ$ auf $K[x_1, \ldots, x_n]$ kann analog zum vorangehenden Beispiel als Ordnung auf $\mathbb{N}^n$ aufgefaßt werden: Für $\alpha, \beta \in \mathbb{N}^n$ setze
$$\alpha \succ \beta \;:\Leftrightarrow\; x^\alpha \succ x^\beta.$$
Im folgenden wird zwischen diesen beiden Interpretationen nicht unterschieden.

Das Monom $1 = x^0$ ist das kleinste bezüglich jeder monomialen Ordnung, denn es teilt alle anderen Monome.

Lemma 1.7 *Jede monomiale Ordnung auf* $K[x_1, \ldots, x_n]$ *ist* **artinsch,** *d.h. jede nichtleere Menge von Monomen aus* $K[x_1, \ldots, x_n]$ *hat ein kleinstes Element.*

Beweis: Kapitel 2, §4, Corollary 6 in [9]. $\square$

Definition 1.8 Seien $\succ$ eine monomiale Ordnung auf $K[x_1, \ldots, x_n]$, $0 \neq f = \sum_{\alpha \in \mathbb{N}^n} a_\alpha x^\alpha \in K[x_1, \ldots, x_n]$. Dann heißt

1. $\exp_\succ(f) := \max\{\, \alpha \in \mathbb{N}^n \mid a_\alpha \neq 0 \,\}$ der **Leitexponent,** wobei das Maximum bezüglich $\succ$ gebildet wird,

2. $\mathrm{in}_\succ(f) := \mathrm{lt}_\succ(f) := a_{\exp_\succ(f)} x^{\exp_\succ(f)}$ der **Initialterm** oder **Leitterm,**

3. $\mathrm{lm}_\succ(f) := x^{\exp_\succ(f)}$ das **Leitmonom,**

4. $\mathrm{lc}_\succ(f) := a_{\exp_\succ(f)}$ der **Leitkoeffizient**

von f (bezüglich $\succ$). Dabei wird $\mathrm{in}_\succ(0) := \mathrm{lt}_\succ(0) := 0$ gesetzt.
Für ein Ideal $I \subseteq K[x_1, \ldots, x_n]$ heißt
$$\exp_\succ(I) := \{\exp_\succ(f) \mid f \in I \backslash \{0\}\}$$
die **Menge der Exponenten** und
$$\mathrm{in}_\succ(I) := \langle\, \mathrm{in}_\succ(f) \mid f \in I \,\rangle$$
das **Initialideal** von I (bezüglich $\succ$). $\square$

Beipiel 1.9 Folgende Festlegungen definieren monomiale Ordnungen (mit $x_1 \succ x_2 \succ \ldots \succ x_n$ in den ersten drei Beispielen):

1. **Lexikographische Ordnung:**

$$\alpha \succ_{lex} \beta \;:\Leftrightarrow\; \exists\, i \in \{1,\dots,n\} \text{ mit } \alpha_j = \beta_j \;\; \forall\, j < i \text{ und } \alpha_i > \beta_i$$

2. **Graduierte lexikographische Ordnung:**

$$\begin{aligned} \alpha \succ_{glex} \beta \;:\Leftrightarrow\; & |\alpha| > |\beta| \text{ oder} \\ & (|\alpha| = |\beta| \text{ und } \alpha \succ_{lex} \beta) \end{aligned}$$

3. **Umgekehrte (graduierte) lexikographische Ordnung** oder **revers-lexikographische Ordnung:**

$$\begin{aligned} \alpha \succ_{rlex} \beta \;:\Leftrightarrow\; & |\alpha| > |\beta| \text{ oder} \\ & (|\alpha| = |\beta| \text{ und} \\ & \exists\, i \in \{1,\dots,n\} \text{ mit } \alpha_j = \beta_j \;\; \forall\, j > i \text{ und } \alpha_i < \beta_i) \end{aligned}$$

4. **Gewichtete lexikographische Ordnung:** Sei $\omega \in \mathbb{R}^n_{\geq 0}$.

$$\begin{aligned} \alpha \succ_{\omega-lex} \beta \;:\Leftrightarrow\; & \omega^\mathsf{T}\alpha > \omega^\mathsf{T}\beta \text{ oder} \\ & (\omega^\mathsf{T}\alpha = \omega^\mathsf{T}\beta \text{ und } \alpha \succ_{lex} \beta) \end{aligned}$$

5. **Gewichtete revers-lexikographische Ordnung:** Sei $\omega \in \mathbb{R}^n_{>0}$.

$$\begin{aligned} \alpha \succ_{\omega-rlex} \beta \;:\Leftrightarrow\; & \omega^\mathsf{T}\alpha > \omega^\mathsf{T}\beta \text{ oder} \\ & (\omega^\mathsf{T}\alpha = \omega^\mathsf{T}\beta \text{ und} \\ & \exists\, i \in \{1,\dots,n\} \text{ mit } \alpha_j = \beta_j \;\; \forall\, j > i \text{ und } \alpha_i < \beta_i) \end{aligned}$$

□

$\exp_\succ(f)$ usw. hängen im allgemeinen von der Wahl von $\succ$ ab. Zum Beispiel ist $\exp_{\succ_{lex}}(x+y^2) = x$, $\exp_{\succ_{rlex}}(x+y^2) = y^2$. Wenn keine Verwechslungsgefahr besteht bzw. wenn die konkrete Wahl der Termordnung unwichtig ist, schreibt man oft $\exp(f)$ an Stelle von $\exp_\succ(f)$ usw.

In den folgenden Kapiteln werden die gewichteten Ordnungen sowie Kombinationen aus den angegebenen Ordnungen (Block- oder Eliminationsordnungen, siehe 1.32, 1.33) eine Rolle spielen.

Bemerkung 1.10

1. Der Gewichtsvektor ω definiert eine Teilordnung $\succ_\omega$ auf $\mathbb{N}^n$ durch

$$\alpha \succ_\omega \beta \;:\Leftrightarrow\; \omega^\mathsf{T}\alpha > \omega^\mathsf{T}\beta\,.$$

Statt von $\succ_\omega$ reden wir auch von der „Teilordnung ω“.

2. ω darf keine negative Komponente besitzen: Sonst existiert ein $\alpha \in \mathbb{N}^n$ mit $\omega^\mathsf{T}\alpha < 0 = \omega^\mathsf{T}0$, und die Teilordnung $\succ_\omega$ ist nicht mit der durch Teilbarkeit gegebenen Teilordnung verträglich:

$$x^0 \mid x^\alpha, \text{ aber } 0 \succ_\omega \alpha.$$

3. Hat ω nur nichtnegative Komponenten, so gilt
$$x^\beta \mid x^\alpha \;\Rightarrow\; \alpha \succeq_\omega \beta,$$
und $\succ_\omega$ kann zu einer **gewichteten** monomialen Ordnung verfeinert werden.

4. Im Falle einer gewichteten revers-lexikographischen Ordnung müssen alle Komponenten von ω strikt positiv sein: Ist nämlich $\omega_i = 0$ für ein $i \in \{1, \dots, n\}$, so gilt für den i-ten Einheitsvektor $e_i \in \mathbb{N}^n$
$$\omega^\top e_i = 0 = \omega^\top 0,$$
und wegen $e_i > 0$ komponentenweise ergibt sich $0 \succ_{\omega-rlex} e_i$, d.h. $\succ_{\omega-rlex}$ ist nicht mit der Teilbarkeit verträglich.

5. Im Falle einer gewichteten lexikographischen Ordnung sind Nullkomponenten zulässig; denn wenn zwei Monome nicht anhand von $\succ_\omega$ verglichen werden können, werden sie durch die *monomiale Ordnung* $\succ_{lex}$ verglichen. □

1.3 Der Divisionsalgorithmus von Hironaka

Bemerkung 1.11 In diesem und den folgenden Abschnitten kommen häufiger Algorithmen vor. Diese werden in einem Pseudocode notiert, der für jeden verständlich sein dürfte, der eine Programmiersprache ähnlich PASCAL oder C kennt. Aus Platzgründen und der Übersichtlichkeit wegen werden logische Abhängigkeiten wie die Reichweite von Schleifen und if-Abfragen durch Einrücken gekennzeichnet. □

Der Divisionsalgorithmus von Hironaka verallgemeinert die Division mit Rest für Polynome in einer Variablen auf Polynome in mehreren Variablen.

Sind $f, f_1, \dots, f_s \in K[x_1, \dots, x_n]$ gegeben, so berechnet er eine Darstellung
$$f = (\sum_{i=1}^{s} g_i f_i) + h$$
mit $g_1, \dots, g_s, h \in K[x_1, \dots, x_n]$, so daß kein $\operatorname{in}(f_i)$, $i = 1, \dots, s$, ein Monom von h teilt. h heißt **Rest** der Division von f durch $f_1, \dots, f_s$ und wird mit $\overline{f}^{f_1,\dots,f_s}$ bezeichnet. Man sagt auch: f **reduziert zu** h **modulo** $f_1, \dots, f_s$.

Natürlich ist das Ergebnis der Division von der zugrundeliegenden monomialen Ordnung abhängig, da dies schon für die Initialterme gilt. Die monomiale Ordnung wird hier und in den folgenden Abschnitten oft nicht ausdrücklich erwähnt; es wird dann implizit angenommen, daß eine solche vorgegeben ist.

Algorithmus 1.12 (Divisionsalgorithmus von Hironaka)

```
(1)  for i := 1 to s do
(2)      g_i := 0;
(3)  h := 0;
(4)  f~ := f;
```

while $\tilde{f} \neq 0$ do

  if $\exists\ i \in \{1, \ldots, s\}$ mit $\mathrm{in}(f_i) \mid \mathrm{in}(\tilde{f})$

    $i_0 := \min\{\, i \ :\ \mathrm{in}(f_i) \mid \mathrm{in}(\tilde{f})\};$

    $t := \dfrac{\mathrm{in}(\tilde{f})}{\mathrm{in}(f_{i_0})};$

    $g_{i_0} := g_{i_0} + t;$

    $\tilde{f} := \tilde{f} - t f_{i_0};$

  else

    $h := h + \mathrm{in}(\tilde{f});$

    $\tilde{f} := \tilde{f} - \mathrm{in}(\tilde{f});$

Beweis der Terminierung: Kapitel 2, §5, Theorem 4 in [9]. □

Beipiel 1.13 Sei $f = xy + 1 \in K[x, y]$. Division mit Rest durch $f_1 = xy - 1$, $f_2 = y - 1$ liefert

$$f = 1 \cdot f_1 + 0 \cdot f_2 + h, \text{ wobei } h = 2;$$

Division mit Rest durch $\tilde{f}_1 = f_2$, $\tilde{f}_2 = f_1$ liefert

$$f = x \cdot \tilde{f}_1 + 0 \cdot \tilde{f}_2 + \tilde{h}, \text{ wobei } \tilde{h} = x + 1.$$

Das Ergebnis der Division mit Rest ist also abhängig von der Reihenfolge der $f_1, \ldots, f_s$. Ist diese festgelegt, so ist das Ergebnis aber eindeutig. □

1.4 Gröbnerbasen

Im folgenden seien $0 \neq I \subseteq K[x_1, \ldots, x_n]$ ein Ideal und $\succ$ eine monomiale Ordnung auf $K[x_1, \ldots, x_n]$.

Definition 1.14 Eine endliche Menge $\{f_1, \ldots, f_s\} \subseteq I$, $s \in \mathbb{N} \backslash \{0\}$, heißt **Gröbnerbasis** von I (bezüglich $\succ$), wenn gilt:

$$\mathrm{in}(I) = \langle\, \mathrm{in}(f_1), \ldots, \mathrm{in}(f_s)\, \rangle \qquad \square$$

Lemma 1.15

1. *I besitzt eine Gröbnerbasis bezüglich $\succ$.*

2. *Jede Gröbnerbasis von I erzeugt I als Ideal.*

Beweis: Kapitel 2, §5, Corollary 6 in [9]. □

Gröbnerbasen sind abhängig von der gewählten monomialen Ordnung und (im allgemeinen) keine Vektorraumbasen des Ideals. Ihre Bedeutung verrät das folgende Lemma:

Lemma 1.16

1. *Ist* $\{f_1, \dots, f_s\}$ *Gröbnerbasis von* I *und* $f \in K[x_1, \dots, x_n]$ *beliebig, so hängt der Rest der Division von* f *durch* $f_1, \dots, f_s$ *nur von* I *und* $\succ$ *ab (insbesondere nicht von der Wahl der Gröbnerbasis und der Reihenfolge ihrer Elemente).*

2. $\{f_1, \dots, f_s\}$ *ist Gröbnerbasis von* $I \Leftrightarrow$
$\forall f \in I$ *ist der Rest der Division von* f *durch* $f_1, \dots, f_s$ *gleich 0.*

Beweis: Kapitel 2, §6, Theorem 6 in [9]. □

Gröbnerbasen sind nicht eindeutig bestimmt: Durch Hinzunahme von Elementen des Ideals bleibt die Gröbnerbasiseigenschaft erhalten.

Definition 1.17 Eine Gröbnerbasis $\{f_1, \dots, f_s\}$ von I heißt **minimal,** wenn gilt:

1. $\mathrm{lc}(f_i) = 1 \;\; \forall\, i = 1, \dots, s$

2. $\mathrm{in}(f_i) \not| \,\mathrm{in}(f_j) \;\; \forall\, i, j = 1 \dots, s,\; i \neq j$ □

Lemma 1.18

1. *I besitzt eine minimale Gröbnerbasis bezüglich* $\succ$.

2. *Je zwei minimale Gröbnerbasen von* I *bezüglich* $\succ$ *haben die gleiche Anzahl von Elementen.*

Beweis: Im wesentlichen Kapitel 2, §4, Theorem 5 in [9]. □

Definition 1.19 Eine Gröbnerbasis $\{f_1, \dots, f_s\}$ von I heißt **reduziert,** wenn gilt:

1. $\mathrm{lc}(f_i) = 1 \;\; \forall\, i = 1, \dots, s$

2. $\mathrm{in}(f_i)$ teilt kein Monom von f_j für alle $i, j = 1 \dots, s,\; i \neq j$. □

Lemma 1.20 *I besitzt bezüglich einer festen monomialen Ordnung* $\succ$ *eine eindeutig bestimmte reduzierte Gröbnerbasis.*

Beweis: Kapitel 2, §7, Proposition 6 in [9]. □

Eine reduzierte Gröbnerbasis ist auch minimal. Minimale Gröbnerbasen sind aber im allgemeinen nicht eindeutig bestimmt: Sind f_1 und f_2 Elemente einer minimalen Gröbnerbasis $f_1, \dots, f_s$ von I bezüglich $\succ$, und ist $\mathrm{in}_\succ(f_1) \succ \mathrm{in}_\succ(f_2)$, so folgt $\mathrm{in}_\succ(f_1 + f_2) = \mathrm{in}_\succ(f_1)$; somit ist $f_1 + f_2, f_2, \dots, f_s$ ebenfalls eine minimale Gröbnerbasis.

Eine beliebige Gröbnerbasis kann man minimalisieren, indem man alle Elemente normiert und diejenigen daraus streicht, deren Initialterme nicht minimal bezüglich der Teilbarkeit sind. Eine minimale Gröbnerbasis kann man reduzieren, indem man jedes Element mit dem Algorithmus von Hironaka modulo den anderen Elementen reduziert.

1.5 Der Buchberger-Algorithmus

Definition 1.21 Seien $f_i, f_j \in K[x_1, \ldots, x_n]$ und

$$m_{ij} := \frac{\mathrm{in}(f_i)}{\gcd(\mathrm{in}(f_i), \mathrm{in}(f_j))},$$

wobei gcd den größten gemeinsamen Teiler bezeichnet. Das Polynom

$$S(f_i, f_j) := m_{ji} f_i - m_{ij} f_j$$

heißt **S-Polynom** oder **S-Paar** von f_i und f_j. □

Das S-Polynom von f_i und f_j kombiniert die beiden Polynome derart, daß sich ihre Initialterme gegenseitig aufheben. Im folgenden bezeichne

$$h_{ij} := \overline{S(f_i, f_j)}^{f_1, \ldots, f_s}$$

den Rest von $S(f_i, f_j)$ bei Division durch $f_1 \ldots, f_s$ (in dieser Reihenfolge).

Satz 1.22 (Buchberger-Kriterium)
Für ein Ideal $I \subseteq K[x_1, \ldots, x_n]$ und $f_1, \ldots, f_s \in I$ sind äquivalent:

1. *$\{f_1, \ldots, f_s\}$ ist eine Gröbnerbasis von I.*
2. *$\langle f_1, \ldots, f_s \rangle = I$ und $h_{ij} = 0 \ \forall\ 1 \leq i, j \leq s$.* □

Beweis: Kapitel 2, §6, Theorem 6 in [9]. □

Ist I durch Erzeuger $f_1, \ldots, f_s$ gegeben, so liefert das folgende Verfahren eine Gröbnerbasis von I (vgl. Kapitel 2, §7, Theorem 2 in [9]):

Algorithmus 1.23 (Buchberger-Algorithmus)

(1) $\mathcal{GB} := (f_1, \ldots, f_s)$;
(2) $done := false$;
(3) while $done = false$ do
(4)   for all $\{p, q\} \subseteq \mathcal{GB}$ do
(5) $h := \overline{S(p,q)}^{\mathcal{GB}}$;
(6)   if $h \neq 0$
(7) $\mathcal{GB}$.append(h);
(8)   break; (∗ beendet for all-, weiter mit while-Schleife ∗)
(9) $done := true$; (∗ alle Reste =0 ∗)
(10) return $\mathcal{GB}$; □

Dabei ist $\mathcal{GB}$ eine Liste, die die aktuell berechneten Gröbnerbasiselemente enthält. $\overline{g}^{\mathcal{GB}}$ bezeichnet den Rest der Division von g durch die Elemente von $\mathcal{GB}$ in der gegebenen Reihenfolge. $\mathcal{GB}$.append(h) fügt ein Polynom h am Ende der Liste $\mathcal{GB}$ ein.

Da durch Hinzunahme eines *Restes* h das Ideal, das von den Initialtermen der Elemente von $\mathcal{GB}$ aufgespannt wird, echt vergrößert wird, terminiert der Algorithmus wegen 1.3 (in noetherschen Ringen wird jede aufsteigende Kette von Idealen stationär, Kapitel 2, §5, Theorem 7 in [9]).

Bemerkung 1.24 (Praktische Aspekte)
Obige Version des Buchberger-Algorithmus stellt nur eine von vielen Möglichkeiten dar – und zwar keine, die für die Praxis gut geeignet ist. Generell sollte man bei der Implementierung beachten, daß bereits berechnete S-Paare nicht ein zweites Mal berechnet werden. Die folgenden Kriterien 1.27 bis 1.29 erlauben sogar, im voraus überflüssige S-Paare zu erkennen.

Praktische Implementierungen berechnen meist die reduzierte Gröbnerbasis, um ein eindeutiges Ergebnis zu garantieren. Wird die Reduktion nicht erst für das Endergebnis durchgeführt, sondern auch gelegentlich für Zwischenergebnisse (**Zwischenreduktionen**), so kann man die Menge $\mathcal{GB}$ kleiner halten und den Algorithmus dadurch beschleunigen.

Wesentliche Freiheiten liegen in der Wahl der Reihenfolge, in der man die Operationen ausführt. In obiger Einfachstversion darf z.B. die Abarbeitungsreihenfolge der forall-Schleife frei gewählt werden. Aus 1.26 folgt, daß man die Reste h sogar an beliebiger Stelle in $\mathcal{GB}$ einfügen kann. Daraus zieht die in Kapitel 5 beschriebene Implementierung Nutzen: Diese Aussage erlaubt es nämlich, die Erzeugerliste $\mathcal{GB}$ in mehrere Listen aufzuspalten und neue Erzeuger in eine beliebige einzufügen.

Eine verbreitete Strategie ordnet die Liste der Erzeuger nach aufsteigenden Initialtermen. Da die Initialterme einer reduzierten Gröbnerbasis die minimalen Elemente von $\mathrm{in}(I)$ bezüglich der Teilbarkeit sind, hofft man, schneller zu einer Gröbnerbasis zu gelangen, indem man Polynome mit kleinem Initialterm immer zuerst betrachtet.

Solche Entscheidungen beeinflussen die Effizienz des Algorithmus wesentlich. Es ist aber schwierig, sie so zu treffen, daß sie für alle Eingabeideale zu einem guten Verhalten führen. □

Definition 1.25 Seien $\mathcal{G} = \{f_1, \ldots, f_s\} \subseteq K[x_1, \ldots, x_n]$ und $f \in K[x_1, \ldots, x_n]$. Kann f in der Form

$$f = a_1 f_1 + \ldots a_s f_s$$

mit $a_i \in K[x_1, \ldots, x_n]$ und $\deg(f) \geq \deg(a_i f_i) \quad \forall\ i = 1, \ldots, s$ geschrieben werden, so sagt man: f **reduziert zu 0 modulo $\mathcal{G}$,** in Zeichen:

$$f \longrightarrow_{\mathcal{G}} 0 \qquad \square$$

Satz 1.26 *Für ein Ideal $I \subseteq K[x1, \ldots, x_n]$ und $\mathcal{G} := \{f_1, \ldots, f_s\} \subseteq I$ sind äquivalent:*

1. *$\mathcal{G}$ ist eine Gröbnerbasis von I.*

2. *$\langle f_1, \ldots, f_s \rangle = I$ und*

$$S(f_i, f_j) \longrightarrow_{\mathcal{G}} 0 \quad \forall\ 1 \leq i < j \leq s.$$

Beweis: Kapitel 2, §9, Theorem 3 in [9]. □

Dieser Satz verallgemeinert Satz 1.22 in folgender Hinsicht: Um eine Menge $\mathcal{G}$ auf ihre Eigenschaft als Gröbnerbasis zu testen, muß man sich bei der Reduktion der S-Paare durch die Elemente von $\mathcal{G}$ nicht an eine feste Reihenfolge dieser Elemente halten.

Satz 1.27 (Teilerfremde Initialterme)
Seien $\mathcal{G} \subseteq K[x_1, \ldots, x_n]$ endlich und $f, g \in \mathcal{G}$, so daß $\mathrm{in}(f)$ und $\mathrm{in}(g)$ teilerfremd sind. Dann gilt:

$$S(f, g) \longrightarrow_{\mathcal{G}} 0$$

Beweis: Kapitel 2, §9, Proposition 4 in [9]. □

Satz 1.28 (Gebauer-Möller-Kriterien)
Sei $\mathcal{G} = \{f_1, \ldots, f_s\} \subseteq K[x_1, \ldots, x_n]$ ein Erzeugendensystem von I, auf das der Buchberger-Algorithmus angewendet werden soll. Das S-Polynom $S(f_i, f_j)$, $i < j$, braucht nicht betrachtet zu werden, wenn eines der folgenden Kriterien für i, j erfüllt ist:

1. **Kriterium M:** *$\exists\ k < j$ mit*

$$\mathrm{lcm}(\mathrm{in}(f_k), \mathrm{in}(f_j)) \mid \mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_j)) \text{ und}$$

$$\mathrm{lcm}(\mathrm{in}(f_k), \mathrm{in}(f_j)) \neq \mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_j))$$

2. **Kriterium F:** *$\exists\ k < i$ mit*

$$\mathrm{lcm}(\mathrm{in}(f_k), \mathrm{in}(f_j)) = \mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_j))$$

3. **Kriterium B:** *$\exists\ k > j$ mit*

$$\mathrm{in}(f_k) \mid \mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_j)) \text{ und}$$

$$\mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_k)) \neq \mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_j)) \neq \mathrm{lcm}(\mathrm{in}(f_j), \mathrm{in}(f_k))$$

Dabei bezeichnet lcm *das kleinste gemeinsame Vielfache.*

Beweis: Theorem 2.5 und Proposition 3.5 in [13]. □

Satz 1.29 (Zweites Buchberger-Kriterium) *Seien $\mathcal{G}$, I wie in 1.28. Das S-Polynom $S(f_i, f_j)$, $i < j$, braucht nicht betrachtet zu werden, wenn gilt:*

$$\exists\ k < i \text{ mit } \mathrm{in}(f_k) \mid \mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_j))$$

Beweis: Kapitel 2, §9, Theorem 9 und Proposition 10 in [9]. □

Die Bedingung $\mathrm{in}(f_k)\,|\,\mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_j))$ des zweiten Buchberger-Kriteriums ist äquivalent zu $\mathrm{lcm}(\mathrm{in}(f_k), \mathrm{in}(f_j))\,|\,\mathrm{lcm}(\mathrm{in}(f_i), \mathrm{in}(f_j))$, verallgemeinert also die Bedingung der Kriterien F und M.

Beipiel 1.30 Sei $I = \langle f_1, f_2\rangle \subset K[x, y, z]$, wobei $f_1 = xz - y$, $f_2 = xy - z$. Eine Gröbnerbasis von I bzgl. $\succ_{lex}$ erhält man wie folgt: Wegen $\mathrm{in}(f_1) = xz$ und $\mathrm{in}(f_2) = xy$ ist

$$S(f_1, f_2) = yf_1 - zf_2 = -y^2 + z^2;$$

da kein Term von $S(f_1, f_2)$ durch $\mathrm{in}(f_1)$ oder $\mathrm{in}(f_2)$ teilbar ist, folgt

$$f_3 := h_{12} = -y^2 + z^2 \neq 0.$$

Wegen 1.27 reduziert $S(f_1, f_3)$ zu 0. Weiter ist

$$S(f_2, f_3) = -yf_2 - xf_3 = -xz^2 + yz = -zf_1,$$

also ist $h_{23} = 0$ und somit $\{f_1, f_2, f_3\}$ eine Gröbnerbasis von I; normiert man f_3 durch Multiplikation mit -1, so ist sie sogar reduziert. □

Allgemein sind Schranken für die Zahl der Schritte, die der Buchberger-Algorithmus bei einer bestimmten Eingabe ausführt, nicht leicht zu finden. Die Schrittzahl hängt von den verschiedensten Faktoren wie der zugrundeliegenden monomialen Ordnung, der Größe der Koeffizienten und dem Grad der Polynome ab. Die bekannten theoretischen Schranken (vgl. [23]) sind sehr schlecht und wenig aussagekräftig, was das Verhalten des Buchberger-Algorithmus in der Praxis anbelangt.

Im speziellen Fall torischer Ideale, der der Gegenstand der nachfolgenden Kapitel ist, bleiben uns einige dieser Probleme erspart; z.B. sind die Koeffizienten der Polynome immer ± 1. Dann gibt es bessere Schranken (vgl. [30], Kapitel 4), deren Eignung als praktischer Anhaltspunkt allerdings noch immer zweifelhaft ist.

1.6 Elimination

Definition 1.31 Sei $\mathcal{X}$ eine Menge und $\succ_1, \succ_2, \ldots$ eine Folge teilweiser Ordnungen auf $\mathcal{X}$. Die zu $\succ_1, \succ_2, \ldots$ gehörige **lexikographische Produktordnung** $\succ$ ist definiert durch

$$x \succ y \;:\Leftrightarrow\; x \succ_i y \text{ für } i \text{ minimal, so daß } x \text{ und } y \text{ vergleichbar sind bzgl. } \succ_i .$$

Man sagt auch: $\succ$ ist $\succ_1$, **verfeinert durch** $\succ_2$, verfeinert durch ... □

Die lexikographische Produktordnung einer Folge von Teilordnungen kann sowohl eine teilweise als auch eine totale Ordnung sein.

Im folgenden sei $T = K[y_1, \ldots, y_m, x_1, \ldots, x_n]$, $R = K[x_1, \ldots, x_n]$ und I ein Ideal in T. Oft möchte man das Ideal $I \cap R$ in R berechnen, d.h. die Variablen $y_1, \ldots, y_m$ **eliminieren.**

Definition 1.32 Eine monomiale Ordnung $\succ$ auf T hat die **Eliminationseigenschaft** bzgl. $y_1, \ldots, y_m$, falls gilt:

$$f \in T,\ \mathrm{in}_{\succ}(f) \in R \ \Rightarrow\ f \in R$$

$\succ$ heißt dann auch **Eliminationsordnung.** □

Beipiel 1.33

1. $\succ_{lex}$ auf $K[x_1, \ldots, x_n]$ hat die Eliminationseigenschaft bzgl. $x_1, \ldots, x_i$ für alle $i = 1, \ldots, n$.

2. Ist $\succ_1$ eine monomiale Ordnung auf $K[y_1, \ldots, y_m]$ und $\succ_2$ eine monomiale Ordnung auf $K[x_1, \ldots, x_n] = R$, so erhält man eine Eliminationsordnung $\succ$ für $y_1, \ldots, y_m$ durch die Festsetzung:

   $$y^\alpha x^\beta \succ y^\gamma x^\delta \ :\Leftrightarrow\ y^\alpha \succ_1 y^\gamma \text{ oder } (y^\alpha = y^\gamma \text{ und } x^\beta \succ_2 x^\delta)$$

   Eine solche Ordnung heißt **Blockordnung** – die Variablen werden blockweise (in Gruppen) betrachtet. Natürlich sind mehr als zwei Blöcke möglich, aber im folgenden werden stets nur zwei benötigt. Blockordnungen kann man als lexikographische Produktordnungen schreiben, daher werden sie auch gelegentlich **Produktordnungen** genannt. □

Satz 1.34 *Sei $\mathcal{GB} := \{f_1, \ldots, f_s\}$ eine Gröbnerbasis von I bzgl. einer monomialen Ordnung $\succ$, die die Eliminationseigenschaft bzgl. $y_1, \ldots, y_m$ erfüllt. Seien $f_1, \ldots, f_t$, $t \leq s$, die Elemente von $\mathcal{GB}$, in denen $y_1, \ldots, y_m$ nicht vorkommen. Dann ist $\{f_1, \ldots, f_t\}$ eine Gröbnerbasis von $I \cap R$ bzgl. der Einschränkung von $\succ$ auf R.*

Beweis: Kapitel 3, §1, Theorem 2 in [9]. Der Beweis ist dort nur für die lexikographische Ordnung aufgeführt, läßt sich aber problemlos übertragen. □

1.7 Graduierung und Homogenität

Definition 1.35

1. Ein **graduierter Ring** S ist ein Ring zusammen mit einer Zerlegung

   $$S = \bigoplus_{d \in \mathbb{N}} S_d$$

   in eine direkte Summe von Untergruppen S_d, so daß gilt:

   $$S_d \cdot S_e \subseteq S_{d+e} \ \ \forall\, d, e \in \mathbb{N}.$$

   Die Elemente von S_d heißen **homogen** vom **Grad** d.

2. Jedes $f \in S$ hat eine eindeutige Darstellung

   $$f = \sum_{d \in \mathbb{N}} f_d,\ f_d \in S_d.$$

   Die f_d heißen die **homogenen Komponenten** von f.

3. Ein Ideal $I \subseteq S$ heißt **homogen,** wenn
$$I = \bigoplus_{d \in \mathbb{N}} (I \cap S_d),$$
also wenn mit $f \in I$ auch jede homogene Komponente von f in I liegt. □

Lemma 1.36 *Für ein Ideal I in einem graduierten Ring S gilt:*

I ist homogen $\Leftrightarrow$ I wird von homogenen Elementen von S erzeugt.

Beweis: (Skizze)

„$\Rightarrow$" folgt unmittelbar aus der Definition: Man zerlege die Erzeuger von I in ihre homogenen Komponenten.

„$\Leftarrow$" Für eine S-Linearkombination f von homogenen Polynomen $f_1, \dots, f_s$ gilt: Alle homogenen Komponenten von f sind Linearkombinationen derselben Polynome $f_1, \dots, f_s$. □

Bemerkung 1.37 Ist $\omega \in \mathbb{N}^n$, so ist $S = K[x_1, \dots, x_n]$ mit der Zerlegung
$$S_d := \{f \in S \mid \text{Jeder Term von } f \text{ hat Gewicht } d \text{ bzgl. } \omega.\}$$
ein graduierter Ring; dabei ist das **Gewicht** von ax^α als $\omega^\top \alpha$ definiert. Die Elemente von S_d heißen **homogen** bzgl. der durch ω definierten **Graduierung** (kurz: der Graduierung ω). Sind alle Komponenten von ω positiv, so spricht man von einer **positiven Graduierung.**

Mit $\omega \in \mathbb{R}^n_{\geq 0}$ funktioniert diese Konstruktion immer noch. Die direkte Summe ist dann nicht mehr mit Elementen aus $\mathbb{N}$, sondern mit Elementen aus $\omega^\top \mathbb{N}^n = \{\omega^\top \alpha \mid \alpha \in \mathbb{N}^n\}$ indiziert – was aber keine tiefgreifende Änderung der Definition verlangt, da diese Menge immer noch abzählbar ist. □

Für den Rest des Abschnitts betrachten wir den graduierten Polynomring $S = K[x_1, \dots, x_n]$ bezüglich einer Graduierung wie in 1.37.

Lemma 1.38

1. *Sind $f \in S$ und $f_1, \dots, f_s \in S$ homogen bzgl. einer Graduierung wie in 1.37, so sind auch $g_1, \dots, g_s$ und h aus 1.12 (Division mit Rest) homogen.*
2. *Das S-Polynom zweier homogener Polynome ist wieder homogen.*

Beweis: (Skizze)

1. Widerspruchsbeweis: Vor jeder Ausführung der if-Abfrage gilt die Darstellung
$$f = (\sum_{i=1}^{s} g_i f_i) + h + \tilde{f},$$
und h, $\tilde{f}$ und die g_i sind zu Beginn homogen. Ferner folgt induktiv, daß $\tilde{f}$ stets homogen bleibt. In jedem Schritt wird nur eines der Polynome $h, g_1, \dots, g_s$ verändert. Nach dem ersten Schritt, durch den eines dieser Polynome inhomogen würde, würde das Ausmultiplizieren obiger Darstellung einen Widerspruch zur Homogenität von f liefern.

2. Das S-Polynom zweier Polynome p und q ist eine S-Linearkombination $t_1p + t_2q$ von p und q mit *Termen* t_1, t_2 als Koeffizienten und der Eigenschaft, daß sich die Vielfache der beiden Initialterme gerade wegheben, also insbesondere gleichen Grad haben. Aus der Homogenität von p und q folgt, daß auch die Vielfache der restlichen Monome von p und q, also die Monome des S-Polynoms, alle gleichen Grad haben.

Satz 1.39 *Sei $I \subseteq S$ ein homogenes Ideal bezüglich einer Graduierung ω wie in 1.37. Dann besteht die reduzierte Gröbnerbasis von I (bzgl. einer beliebigen monomialen Ordnung) aus Elementen, die ebenfalls homogen bezüglich ω sind.*

Beweis: Da S noethersch ist, wird I von endlich vielen Polynomen erzeugt, also auch von endlich vielen homogenen Polynomen (siehe Beweis von „$\Rightarrow$“ in 1.36). Wendet man auf letztere den Buchberger-Algorithmus an, so folgt die Behauptung aus dem vorangehenden Lemma. $\square$

1.8 Idealquotient und Saturierung

In diesem und im folgenden Abschnitt sei R ein kommutativer Ring.

Definition 1.40 Sind I, J Ideale in R, so heißt

$$(I : J) := \{f \in R \mid f \cdot g \in I \;\; \forall\, g \in J\}$$

der **Idealquotient** von I durch J. Ist dabei $J = \langle g \rangle,\; g \in R$, ein Hauptideal, so schreibt man

$$(I : g) := (I : J) = \{f \in R \mid f \cdot g \in I\}. \qquad \square$$

Definition 1.41 Sind I, J Ideale in R, so heißt

$$(I : J^\infty) := \{f \in R \mid \;\forall\, g \in J\; \exists\, d \in \mathbb{N} \text{ mit } f \cdot g^d \in I\}$$

die **Saturierung** von I bezüglich J. Ist dabei $J = \langle g \rangle,\; g \in R$, ein Hauptideal, so schreibt man

$$(I : g^\infty) := (I : J^\infty) = \{f \in R \mid \exists\, d \in \mathbb{N} \text{ mit } f \cdot g^d \in I\}.$$

I heißt **saturiert** bezüglich $g \in R$, falls $(I : g^\infty) = I$. $\square$

Die Klammern um $I : J$ bzw. $I : J^\infty$ werden häufig weggelassen.

Bemerkung 1.42 Unmittelbar aus den Definitionen folgt:

1. $I : J$ und $I : J^\infty$ sind ebenfalls Ideale in R mit $I \subseteq (I : J) \subseteq (I : J^\infty)$.

2. Für $f, g \in R$ gilt: $I : (f \cdot g)^\infty = (I : f^\infty) : g^\infty$

3. Ist $J \subseteq I$, so ist auch $(J : f) \subseteq (I : f)$ und $(J : f^\infty) \subseteq (I : f^\infty)$. $\square$

Lemma 1.43 *Ist $g \in R$ invertierbar modulo dem Ideal I, so ist*

$$(I : g^\infty) = (I : g) = I.$$

Beweis: „g ist invertierbar modulo I“ bedeutet: Es existiert ein $h \in R$ mit $hg \equiv 1 \bmod I$, d.h. $hg - 1 \in I$. Die Inklusionen „$\supseteq$“ sind klar. Die Inklusionen „$\subseteq$“ ergeben sich aus

$$gf \in I \;\Rightarrow\; hgf \in I \;\Rightarrow\; hgf - \underbrace{(hg-1)}_{\in I} f = f \in I\,. \qquad \square$$

1.9 Ergänzungen

1.9.1 Primideale

Definition 1.44 Ein echtes Ideal $I \subsetneqq R$ heißt **Primideal,** wenn für $f, g \in R$ gilt:

$$f \cdot g \in I \;\Rightarrow\; f \in I \;\vee\; g \in I \qquad \square$$

Bemerkung 1.45 Ist S ein Integritätsring und $\varphi : R \to S$ ein Ringhomomorphismus, so ist $\ker(\varphi)$ ein Primideal in R. $\square$

1.9.2 Laurentpolynome

Die folgende Verallgemeinerung von Polynomringen wird in dieser Arbeit nur für formale Rechnungen gebraucht; die Idee der Inversion von Variablen ist jedoch grundlegender Bestandteil der Algorithmen aus Kapitel 3.

Definition 1.46 Ersetzt man in Formel (1.1) die Forderung $\alpha \in \mathbb{N}^n$ durch $\alpha \in \mathbb{Z}^n$, so erhält man ein **Laurentpolynom.** Mit den Verknüpfungen aus Definition 1.1 wird die Menge der Laurentpolynome zu einer kommutativen Algebra mit $1 = 1_K$, dem **Laurentpolynomring** $K[x_1, x_1^{-1}, \ldots, x_n, x_n^{-1}]$. $\square$

Dabei handelt es sich nicht um einen Polynomring in $2n$ Variablen; denn nach Definition kommen in keinem Monom x_j und x_j^{-1} gleichzeitig vor.

Lemma 1.47 $K[x_1, x_1^{-1}, \ldots, x_n, x_n^{-1}]$ *ist Integritätsring.*

Beweis: (Skizze) Induktion nach n:
Für $n = 0$ ist die Behauptung klar, da K Integritätsring. Ist $n \geq 1$, so gilt:

$$\begin{aligned} K[x_1, x_1^{-1}, \ldots, x_n, x_n^{-1}] &\cong K[x_1, x_1^{-1}, \ldots, x_{n-1}, x_{n-1}^{-1}][x_n, x_n^{-1}] \\ &\cong K[x_1, x_1^{-1}, \ldots, x_{n-1}, x_{n-1}^{-1}][x_n, y_n] \big/ \langle x_n y_n - 1 \rangle \end{aligned}$$

Die erste Isomorphie ergibt sich durch Zusammenfassung der Terme mit gleicher Potenz in x_n, die zweite durch die Abbildung $x_n \mapsto \overline{x_n},\; x_n^{-1} \mapsto \overline{y_n}$.
Ist nun $R := K[x_1, x_1^{-1}, \ldots, x_{n-1}, x_{n-1}^{-1}]$ nullteilerfrei, so auch $R[x_n, y_n]$. Wegen der Irreduzibilität von $x_n y_n - 1$ in $R[x_n, y_n]$ ist das Hauptideal $\langle x_n y_n - 1 \rangle$ prim und somit $R[x_n, y_n]/\langle x_n y_n - 1 \rangle \cong K[x_1, x_1^{-1}, \ldots, x_n, x_n^{-1}]$ nullteilerfrei. $\square$

1.10 Gitter

Definition 1.48

1. Ein $\mathbb{Z}$-Modul heißt **frei,** wenn er eine Basis besitzt.

2. Ein endlich erzeugter, freier $\mathbb{Z}$-Modul heißt **Gitter (lattice).** □

Bemerkung 1.49 Das Standardwerk [6] verlangt für ein Gitter L zusätzlich die Angabe einer positiv definiten quadratischen Form auf $L \otimes \mathbb{R}$. Hier werden wir immer die quadratische Form $q(x) := x^\intercal x$ zugrundelegen, die durch das Standardskalarprodukt auf $\mathbb{R}^n$ gegeben ist. □

Die Gitter, die uns interessieren, sind spezieller Natur: Ist

$$A \in \mathbb{Z}^{m \times n} := \{m \times n - \text{Matrizen mit Koeffizienten in } \mathbb{Z}\},$$

so ist die Menge $\ker(A) \cap \mathbb{Z}^n$ ein Untermodul von $\mathbb{Z}^n$. Aus dem nächsten Satz folgt, daß sie sogar ein Untergitter, d.h. endlich erzeugt und frei ist; dies ist sehr wichtig, da wir später Erzeugendensysteme bzw. Basen für $\ker(A)$ als Teilmenge von $\mathbb{Z}^n$ berechnen müssen.

Bemerkung 1.50 In einem freien Modul ist die Anzahl der Elemente einer Basis eindeutig bestimmt. Diese Anzahl heißt **Rang** oder **Dimension** des Moduls. □

Satz 1.51 *Sei F ein freier, endlich erzeugter Modul über $\mathbb{Z}$ und M ein Untermodul von F. Dann ist M frei und der Rang von M ist kleiner oder gleich dem von F.*

Beweis: Theorem 7.2 in [20]. Es handelt sich um einen recht elementaren Induktionsbeweis. □

Bemerkung 1.52 Nicht jede ganzzahlige Vektorraumbasis von $\mathbb{R}^n$ ist eine Gitterbasis von $\mathbb{Z}^n$: Zum Beispiel erzeugen die Vektoren $(1,2)^\intercal$ und $(2,1)^\intercal$ $\mathbb{Z}^2$ als Vektorraum, nicht aber als Gitter; denn der Vektor $(1,1)^\intercal = \frac{1}{3}(1,2)^\intercal + \frac{1}{3}(2,1)^\intercal$ läßt sich nicht als ganzzahlige Linearkombination von $(1,2)^\intercal$ und $(2,1)^\intercal$ darstellen. Insbesondere kann man zur Berechnung einer Gitterbasis keine gewöhnliche Gaußelimination verwenden. □

Gitterbasen für den Kern von Matrizen kann man z.B. mit Hilfe der sogenannten **Hermite-Normalform** berechnen (vgl. [26], Kapitel 4-6 für eine umfangreiche Theorie dazu oder Algorithmus 2.4.1 in [6]). Wir stellen hier den **LLL-Algorithmus** vor. Beide Ansätze können eine Gitterbasis in polynomieller Zeit berechnen; der erstere allerdings nur, wenn die Hermite-Normalform auf geschickte Weise bestimmt wird (vgl. Corollary 5.3c in [26]).

Bemerkung 1.53

1. Die meisten der Algorithmen aus Kapitel 3 führen solche Gitterbasisberechnungen durch. Es stellt sich heraus, daß diese nur einen winzigen Bruchteil der Laufzeit ausmachen; in dieser Beziehung ist es für unseren Zweck nicht notwendig, beide Algorithmen zu testen.

2. Wichtig ist dagegen, wie die berechnete Gitterbasis aussieht; konkret sollten die Koeffizienten der Basisvektoren möglichst klein bleiben. Daher erweitert man beide Algorithmen um eine Reduktionsphase, wobei der Begriff einer reduzierten Basis recht unterschiedlich ausfällt (vgl. Theorem 6.4 in [26] und Definition 2.6.1 in [6]). □

Definition 1.54 Sei L ein Gitter mit Basis $b_1, \ldots, b_r$. Definiere induktiv

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^* \text{ für } i = 1, \ldots, r,$$

wobei

$$\mu_{i,j} = \frac{b_i^\top b_j^*}{|b_j^*|^2} \text{ für } 1 \le j < i \le r$$

und $|x| := \sqrt{x^\top x}$ für $x \in L$. $b_1, \ldots, b_r$ heißen **LLL-reduziert,** wenn gilt:

1. $\mu_{i,j} \le \frac{1}{2}$ für $1 \le j < i \le r$ und
2. $|b_i^* + \mu_{i,i-1} b_{i-1}^*|^2 \ge \frac{3}{4} |b_{i-1}^*|^2$ für $1 \le i \le r$. □

Bemerkung 1.55 Die b_i^* aus 1.54 bilden die orthogonale Basis, die das Gram-Schmidt-Verfahren aus den b_i konstruiert. □

Die folgenden Seiten führen zwei Versionen des LLL-Algorithmus auf. Der ganzzahlige LLL-Algorithmus 1.56 erhält als Eingabe r Vektoren $b_1, \ldots, b_r$ mit ganzzahligen Koeffizienten, die eine Basis des Gitters L bilden, und transformiert sie in eine LLL-reduzierte Basis. Er entspricht Algorithmus 2.6.7 in [6]; dort wird zusätzlich eine Umformungsmatrix berechnet. Alle Operationen bleiben im Bereich der ganzen Zahlen. Der Algorithmus benutzt einige Hilfsvariablen, auf deren Funktion auch in [6] nicht genau eingegangen wird.

Die Variante 1.57 zur Gitterbasisberechnung (Algorithmus 2.7.2 in [6]) erhält als Eingabe eine Matrix $A \in \mathbb{Z}^{m \times n}$, deren Spalten mit $b_1, \ldots, b_n$ bezeichnet werden, und liefert eine Matrix $H \in \mathbb{Z}^{n \times n}$ und eine Zahl $r \in \mathbb{N}$ zurück, so daß die ersten r Spalten von H eine Gitterbasis von $\ker(A)$ bilden. Diese kann dann mit Algorithmus 1.56 LLL-reduziert werden. In der angegebenen Version muß $n \ge 2$ gelten; für $n = 1$ ist die Berechnung einer Gitterbasis jedoch trivial.

Die Algorithmen aus [6] sind hier optisch stark verändert, die aus der Mode gekommenen Sprunganweisungen meistens durch Schleifen ersetzt. Zur Orientierung haben wir die einzelnen Schritte des Originals im Kommentar vermerkt. Die Variablennamen wurden beibehalten.

Algorithmus 1.56 (Ganzzahliger LLL-Algorithmus)

(1) $k := 2$; $k_{max} := 1$; $d_0 := 1$; $d_1 := b_1^\top b_1$;

```
do
    if $k > k_{max}$
        (* „inkrementelles Gram-Schmidt-Verfahren“ *)
        $k_{max} := k$;
        for $j := 1$ to $k$ do
            $u = b_k^\intercal b_j$;
            for $i := 1$ to $j - 1$ do
                $u := \frac{d_i u - \lambda_{k,i} \lambda j, i}{d_{i-1}}$;
            if $j < k$
                $\lambda_{k,j} := u$;
            else (* $j = k$ *)
                $d_k := u$ ;
        if $d_k = 0$
            beende mit Fehler: Eingabevektoren linear abhängig;
    do
        (* teste „LLL-Bedingung“ *)
        Reduce-Integral($k, k-1$); (* Subroutinen: siehe unten *)
        if $d_k d_{k-2} < \frac{3}{4} d_{k-1}^2 - \lambda_{k,k-1}^2$
            Swap-Integral($k$);
            $k := \max(2, k-1)$;
        else
            break (* verlasse innere do-Schleife *);
    while $true$ (* solange nicht durch break beendet *);
    for $l := k - 2$ downto 1 do
        Reduce-Integral($k, l$);
    $k := k + 1$;
while $k \leq r$;
```

Algorithmus 1.57 (Gitterbasis des Kerns mit LLL)

```
$k := 2$; $k_{max} := 1$; $d_0 := 1$; $t := b_1^\intercal b_1$;
$H := I_n$; (* $n \times n$-Einheitsmatrix *)
if $t \neq 0$
    $d_1 := t$; $f_1 := 1$;
else
    $d_1 := 1$; $f_1 := 0$;
```

```
do
    if $k > k_{max}$
        (∗ „inkrementelles Gram-Schmidt-Verfahren“ ∗)
        $k_{max} := k$;
        for $j := 1$ to $k$ do
            if $j < k$ and $f_j = 0$
                $\lambda_{k,j} := 0$;
            else
                $u := b_k^\top b_j$;
                for $i := 1$ to $j - 1$ do
                    if $f_i \neq 0$
                        $u := \dfrac{d_i u - \lambda_{k,i}\lambda_{j,i}}{d_{i-1}}$;
                if $j < k$
                    $\lambda_{k,j} := u$;
                else (∗ $j = k$ ∗)
                    if $u \neq 0$
                        $d_k := u$; $f_k := 1$;
                    else (∗ $u = 0$ ∗)
                        $d_k := d_{k-1}$; $f_k := 0$;
    do
        (∗ teste, ob $f_k = 0$ und $f_{k-1} \neq 0$ ∗)
        if $f_{k-1} \neq 0$
            Reduce-Kernel($k, k - 1$);
        if $f_{k-1} \neq 0$ and $f_k = 0$
            Swap-Kernel($k$); $k := \max(2, k - 1)$;
        else
            break; (∗ verlasse innere do-Schleife ∗)
    while $true$;  (∗ solange nicht durch break beendet ∗)
    for $l := k - 2$ downto 1 do
        if $f_l \neq 0$
            Reduce-Kernel($k, l$);
    $k := k + 1$;
while $k \leq n$;
$f_{n+1} := 1$;  (∗ wegen nächster Zeile, falls alle $f_i = 0$ ∗)
$r := \min\{\, i \mid f_i \neq 0 \,\} - 1$;  (∗ Dimension der Basis ∗)
```

Die eigentliche Reduktion erfolgt in den Subroutinen:

Reduce-Integral(k, l)

(1) if $|2\lambda_{k,l}| > d_l$

(2) $\quad q := \left\lfloor \frac{2\lambda_{k,l} + d_l}{2d_l} \right\rfloor$; (∗ nächste ganze Zahl zu $\frac{\lambda_{k,l}}{d_l}$ ∗)

(3) $\quad b_k := b_k - qb_l$; $\lambda_{k,l} := \lambda_{k,l} - qd_l$;

(4) $\quad$ for $i := 1$ to $l - 1$ do

(5) $\quad\quad \lambda_{k,i} := \lambda_{k,i} - q\lambda_{l,i}$;

Swap-Integral(k)

(1) $b_k :=: b_{k-1}$; (∗ vertausche ∗)

(2) if $k > 2$

(3) $\quad$ for $j := 1$ to $k - 2$ do

(4) $\quad\quad \lambda_{k,j} :=: \lambda_{k-1,j}$;

(5) $\lambda := \lambda_{k,l}$; $B := \frac{d_{k-2}d_k + \lambda^2}{d_{k-1}}$;

(6) for $i := k + 1$ to k_{max} do

(7) $\quad t := \lambda_{i,k}$;

(8) $\quad \lambda_{i,k} := \frac{d_k\lambda_{i,k-1} - \lambda t}{d_{k-1}}$; $\lambda_{i,k-1} := \frac{Bt + \lambda\lambda_{i,k}}{d_k}$;

(9) $d_{k-1} := B$; □

Swap-Kernel(k)

(1) $H_k :=: H_{k-1}$;

(2) $b_k :=: b_{k-1}$; (∗ fehlt in [6] ∗)

(3) if $k > 2$

(4) $\quad$ for $j := 1$ to $k - 2$ do

(5) $\quad\quad \lambda_{k,j} :=: \lambda_{k-1,j}$;

(6) $\lambda := \lambda_{k,k-1}$;

(7) if $\lambda = 0$

(8) $\quad d_{k-1} := d_{k-2}$;

(9) $\quad f_k :=: f_{k-1}$;

(10) $\quad$ for $i := k + 1$ to k_{max} do

(11) $\quad\quad \lambda_{i,k} := \lambda_{i,k-1}$;

(12) $\quad\quad \lambda_{i,k-1} := 0$;

(13) else (* $\lambda \neq 0$ *)

(14) for $i := k+1$ to k_{max} do

(15) $\lambda_{i,k-1} := \dfrac{\lambda\lambda_{i,k-1}}{d_{k-1}}$;

(16) $t := d_k$;

(17) $d_{k-1} := \dfrac{\lambda^2}{d_{k-1}}$;

(18) $d_k := d_{k-1}$;

(19) for $j := k+1$ to $k_{max} - 1$ do

(20) for $i := j+1$ to k_{max} do

(21) $\lambda_{i,j} := \dfrac{\lambda i, j d_{k-1}}{t}$;

(22) for $j := k+1$ to k_{max} do

(23) $d_j := \dfrac{d_j d_{k-1}}{t}$; □

Die Prozedur Reduce-Kernel erhält man aus Reduce-Integral, indem man Zeile (3) durch eine entsprechende Umformung der Matrix H ergänzt:

$$H_k := H_k - qH_l;$$

Beide Versionen des LLL-Algorithmus führen an den Spalten von A elementare Umformungen durch; die zweite merkt sich diese in der Matrix H. Die Umformungen sind zweierlei Art:

- „Addiere ein ganzzahliges Vielfaches einer Spalte zu einer anderen Spalte“: Dies wird in Reduce-Integral bzw. Reduce-Kernel getan und läßt die Determinante von H unverändert.

- „Vertausche zwei Spalten“: Dies wird in Swap-Integral bzw. Swap-Kernel getan; die Determinante von H ändert sich dadurch um einen Faktor -1.

Die Umformungsmatrix H bleibt also stets invertierbar über $\mathbb{Z}$.

Leider ist der Begriff der LLL-reduzierten Basis nicht sehr anschaulich. Die Wirkungsweise der Algorithmen macht man sich am besten an Beispielen klar. Genauere Erläuterungen findet man in [6], Abschnitt 2.6.1. ff.

Beipiel 1.58 Betrachte die 1×3-Matrix $A = (4\ 2\ 3)$. Folgende Tabelle zeigt anhand der Veränderungen der Eingabevektoren $b_1 = 4$, $b_2 = 2$, $b_3 = 3$ und der Transformationsmatrix H den Ablauf des LLL-Algorithmus zur Berechnung einer Gitterbasis von $\ker(A)$. Die Tabelle führt nur diejenigen Aufrufe von Reduce-Kernel auf, die auch zu einer Veränderung führen.

	(b_1, b_2, b_3)	H
am Anfang	$(4,\ 2,\ 3)$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
nach Swap-Kernel(2)	$(2,\ 4,\ 3)$	$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
nach Reduce-Kernel(2,1)	$(2,\ 0,\ 3)$	$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
nach Swap-Kernel(2)	$(0,\ 2,\ 3)$	$\begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
nach Reduce-Kernel(3,2)	$(0,\ 2, -1)$	$\begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}$
nach Swap-Kernel(3)	$(0, -1,\ 2)$	$\begin{pmatrix} 1 & 0 & 0 \\ -2 & -2 & 1 \\ 0 & 1 & 0 \end{pmatrix}$
nach Reduce-Kernel(3,2)	$(0, -1,\ 0)$	$\begin{pmatrix} 1 & 0 & 0 \\ -2 & -2 & -3 \\ 0 & 1 & 2 \end{pmatrix}$
nach Swap-Kernel(3,2)	$(0,\ 0, -1)$	$\begin{pmatrix} 1 & 0 & 0 \\ -2 & -3 & -2 \\ 0 & 2 & 1 \end{pmatrix}$

Die Eingabevektoren werden also in die Vektorenfolge $\tilde{b}_1 = 0$, $\tilde{b}_2 = 0$, $\tilde{b}_3 = -1$ mit zwei Nullvektoren und dem betragsmäßig kleinen Vektor -1 transformiert. Die ersten zwei Spalten von H liefern die gesuchte Gitterbasis $\{(1, -2, 0)^\mathsf{T}, (0, -3, 2)^\mathsf{T}\}$. Wendet man Algorithmus 1.56 auf diese an, so wird nur ein einziger Prozeduraufruf getätigt, nämlich Reduce-Integral(2,1); dabei werden die Vektoren in die reduzierte Gitterbasis $\{(1, -2, 0)^\mathsf{T}, (1, -1, 2)^\mathsf{T}\}$ transformiert. □

Wie man sieht, muß man schon im Falle einer 1×3-Matrix ein bißchen rechnen. In [6] wird folgende theoretische Laufzeitschranke für eine nicht ganzzahlige Variante des LLL-Algorithmus angegeben:

Ist $b_1, \ldots, b_r$ eine Gitterbasis und $B = \max_{1 \le i \le r} |b_i|^2$, so ist die Laufzeit des LLL-Algorithmus auf der Eingabe $b_1, \ldots, b_r$ beschränkt durch

$$\mathcal{O}(r^6 \cdot \ln^3(B));$$

d.h. es gibt eine Konstante $c \in \mathbb{R}$, so daß für genügend große r der Algorithmus $\le c \cdot r^6 \cdot \ln^3(B)$ Zeiteinheiten benötigt.

Diese Formulierung macht allerdings weder eine Aussage über die praktisch zu erwartende Laufzeit noch sagt sie etwas über die Existenz von Instanzen aus, für die die Schranke scharf ist.

Kapitel 2

Ganzzahlige Optimierung

2.1 Die Problemstellung

Definition 2.1 Eine Aufgabe der Form

$$\begin{aligned} \text{minimiere bzw. maximiere } & \sum_{j=1}^{n} c_j X_j \\ \text{mit } X = (X_1, \ldots, X_n)^\top & \in \mathbb{Z}^n, \\ \sum_{j=1}^{n} a_{ij} X_j \bowtie b_i & \text{ für } i = 1, \ldots, m, \\ X_j \geq 0 & \text{ für } j \in J_1, \\ X_j \leq 0 & \text{ für } j \in J_2, \end{aligned} \tag{2.1}$$

wobei

$$\begin{gathered} \bowtie \in \{=, \leq, \geq\}, \\ J_1, J_2 \subseteq \{1, \ldots, n\}, \\ c_j, a_{ij}, b_i \in \mathbb{Z} \text{ für } i = 1, \ldots, m, \ j = 1, \ldots, n, \end{gathered}$$

heißt **Instanz** des **ganzzahligen (linearen) Optimierungsproblems** oder **Integer (Linear) Programming Problems (IP-Problems).**
Der Vektor $c = (c_1, \ldots, c_n)^\top$ wird als lineare Funktion

$$c : \mathbb{Z}^n \to \mathbb{Z}; \ X \mapsto c^\top X = \sum_{j=1}^{n} c_j X_j$$

aufgefaßt und heißt **Zielfunktion, Kostenfunktion** oder **Kostenvektor.** Die (Un-)Gleichungen

$$\sum_{j=1}^{n} a_{ij} X_j \bowtie b_i$$

heißen **Nebenbedingungen** oder **Restriktionen.**
Ein Vektor $X^0 \in \mathbb{Z}^n$, der die Nebenbedingungen erfüllt (d.h. $\sum_{j=1}^{n} a_{ij} X_j^0 \bowtie b_i$ für $i = 1, \ldots, m$), heißt **zulässig** oder **zulässige Lösung** für die Instanz; ein zulässiger Vektor $X^0 \in \mathbb{Z}^n$ mit

$$c^\mathsf{T} X^0 = \min \text{ bzw. } \max\{c^\mathsf{T} X \mid X \in \mathbb{Z}^n \text{ zulässig}\}$$

heißt **optimal** oder **optimale Lösung.** Die Menge aller zulässigen Lösungen heißt **Zulässigkeitsbereich.** Eine Instanz **lösen** heißt, eine optimale Lösung für die Instanz zu finden.

Eine Instanz der Form

$$\left.\begin{array}{c} \text{minimiere } \sum_{j=1}^{n} c_j X_j \\ \text{mit } X \in \mathbb{Z}^n, \\ \sum_{j=1}^{n} a_{ij} X_j = b_i \text{ für } i = 1, \ldots, m, \\ X_j \geq 0 \text{ für } j = 1, \ldots, n, \end{array}\right\} \tag{2.2}$$

hat **Standardform.** Man schreibt (2.2) dann auch in Matrixform

$$\min\{c^\mathsf{T} X \mid X \in \mathbb{Z}^n,\ AX = b,\ X \geq 0\}, \tag{2.3}$$

wobei

$$c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \in \mathbb{Z}^n,\ A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix} \in \mathbb{Z}^{m\times n},\ b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \in \mathbb{Z}^m$$

und $X \geq 0$ komponentenweise zu verstehen ist. A heißt die **Koeffizientenmatrix** und b der **Begrenzungsvektor („rechte Seite“).** □

Bemerkung 2.2

1. Man sagt oft „IP-Problem“ und meint damit eine konkrete Instanz eines IP-Problems. Strenggenommen ist ein IP-Problem eine Menge von Instanzen mit bestimmten charakteristischen Eigenschaften (vgl. Abschnitt 2.2).

2. Die Vorzeichenbedingung $X_j \geq 0$ kann auch geschrieben werden als

$$\sum_{k=1}^{n} a_{ik} X_k \geq b_i \text{ mit } b_i = 0 \text{ und } a_{ik} = \begin{cases} 0, & \text{falls } k \neq j, \\ 1, & \text{falls } k = j. \end{cases}$$

   Rein formal sind die Vorzeichenbedingungen in der Formulierung (2.1) also überflüssig. Sie spielen jedoch in den meisten Verfahren zur Lösung von IP-Problemen eine große Rolle und werden dort nicht wie andere Nebenbedingungen behandelt.

3. Jede Instanz der Form (2.1) kann in eine äquivalente Instanz in Standardform überführt werden; äquivalent bedeutet hier, daß man die zulässigen bzw. optimalen Lösungen von (2.2) durch einfache Transformationen aus den zulässigen bzw. optimalen Lösungen von (2.1) erhält und umgekehrt. Konkret funktioniert das folgendermaßen:

(a) Soll $c^\intercal X$ maximiert werden, minimiere stattdessen $(-c)^\intercal X$. Die Menge der zulässigen bzw. optimalen Lösungen verändert sich hierdurch überhaupt nicht.

(b) Ist $X_j \in J_1 \cap J_2$, so ist die j-te Komponente aller zulässigen Lösungen $= 0$; streiche die Variable X_j und entsprechend c_j und a_{ij} für $i = 1, \ldots, m$.
Ist $X_j \in J_2 \backslash J_1$, so ersetze X_j durch $Y_j := -X_j$ und entsprechend c_j durch $-c_j$ und a_{ij} durch $-a_{ij}$ für $i = 1, \ldots, m$.
Ist $X_j \notin J_1 \cup J_2$, also ohne Vorzeichenbeschränkungen, so ersetze X_j durch die Differenz $Y_{j_1} - Y_{j_2}$ zweier Variablen $Y_{j_1}, Y_{j_2} \geq 0$.

(c) Lautet eine Nebenbedingung $\sum_{j=1}^n a_{ij} X_j \leq b_i$, so führe eine **Schlupfvariable** $Y_i \geq 0$ ein, die für die Differenz $b_i - \sum_{j=1}^n a_{ij} X_j \in \mathbb{Z}$ steht. Ersetze die Nebenbedingung durch

$$\sum_{j=1}^n a_{ij} X_j + Y_i = b_i.$$

Eine Nebenbedingung der Form $\sum_{j=1}^n a_{ij} X_j \geq b_i$ wird durch Multiplikation mit -1 äquivalent in $\sum_{j=1}^n (-a_{ij}) X_j \leq -b_i$ umgeformt und dann wie oben behandelt. □

Lemma 2.3 *Für jede Instanz des IP-Problems trifft genau einer der folgenden drei Fälle zu:*

1. *Sie ist unlösbar, d.h. der Zulässigkeitsbereich ist leer.*

2. *Sie besitzt eine optimale Lösung.*

3. *Sie ist unbeschränkt, d.h. sie besitzt beliebig „gute“ zulässige Lösungen.*

Beweis: Wir betrachten den Fall eines Minimierungsproblems; der eines Maximierungsproblems läßt sich mit Hilfe der Transformationen aus Bemerkung 2.2 darauf zurückführen.

Sei $r := \inf\{c^\intercal X \mid X \text{ zulässige Lösung}\} \in \mathbb{R} \cup \{\pm\infty\}$ (r existiert!). Ist $r = +\infty$, so ist die Menge der zulässigen Lösungen leer, und es gilt Fall 1. Ist $r = -\infty$, befinden wir uns in Fall 3. Ist r endlich, so ist die Menge $\{c^\intercal X | X \text{ zulässige Lösung}\}$ nach unten beschränkt und eine Teilmenge von $\mathbb{Z}$, da $c \in \mathbb{Z}^n$ und $X \in \mathbb{Z}^n$ für alle zulässigen X gilt. Also existiert $\min\{c^\intercal X \mid X \text{ zulässige Lösung}\}$, und wir befinden uns in Fall 2. □

Beipiel 2.4 (Transformation auf Standardform)
Die Instanz

$$\max X_1 + 2X_2$$

$$\text{mit } X_1 + X_2 \leq 3,\ X_1 \geq 0,\ X_1, X_2 \in \mathbb{Z} \tag{2.4}$$

wird zunächst zum Minimierungsproblem

$$\begin{aligned} &\min -X_1 - 2X_2 \\ &\text{mit } X_1 + X_2 \le 3,\ X_1 \ge 0,\ X_1, X_2 \in \mathbb{Z}; \end{aligned} \tag{2.5}$$

durch die Ersetzung $X_2 = Y_{2_1} - Y_{2_2}$, $Y_{2_1}, Y_{2_2} \ge 0$, erhält man

$$\begin{aligned} &\min -X_1 - 2Y_{2_1} + 2Y_{2_2} \\ &\text{mit } X_1 + Y_{2_1} - Y_{2_2} \le 3,\ X_1, Y_{2_1}, Y_{2_2} \ge 0,\ X_1, Y_{2_1}, Y_{2_2} \in \mathbb{Z}, \end{aligned} \tag{2.6}$$

und schließlich durch Einführung der Schlupfvariablen Y_1

$$\begin{aligned} &\min -X_1 - 2Y_{2_1} + 2Y_{2_2} \\ &\text{mit } X_1 + Y_{2_1} - Y_{2_2} + Y_1 = 3,\ X_1, Y_{2_1}, Y_{2_2}, Y_1 \ge 0,\ X_1, Y_{2_1}, Y_{2_2}, Y_1 \in \mathbb{Z}. \end{aligned} \tag{2.7}$$

Der zulässigen Lösung

$$X_1 = 1,\ Y_{2_1} = 1,\ Y_{2_2} = 2,\ Y_1 = 3$$

von (2.7) entsprechen dann die zulässige Lösung

$$X_1 = 1,\ Y_{2_1} = 1,\ Y_{2_2} = 2$$

von (2.6) und die zulässige Lösung

$$X_1 = 1,\ X_2 = 1 - 2 = -1$$

von (2.5) bzw. (2.4). Beim Übergang von (2.5) auf (2.6) ist diese Entsprechung nicht eineindeutig, weil sich der Wert von X_2 auf viele Arten als Differenz nichtnegativer Werte schreiben läßt. □

Das Beispiel zeigt, daß sich durch Transformation auf Standardform die Zahl der Variablen erheblich vergrößern kann.

2.2 Beispiele ganzzahliger Optimierungsprobleme

Beipiel 2.5 (Produktionsplanung) Eine Fabrik stellt n Produkte her. Zur Herstellung stehen m Ressourcen (z.B. Arbeitszeit, Material) zur Verfügung. Eine Einheit von Produkt j bringt einen Gewinn von c_j und braucht a_{ij} Einheiten von Ressource i. Von Ressource i sind b_i Einheiten vorhanden. Man möchte nun einen Produktionsplan erstellen, der den Gewinn maximiert.

Bezeichnet X_j die Anzahl der hergestellten Einheiten von Produkt j, so kann man das Problem wie folgt modellieren:

$$\begin{aligned} &\max \sum_{j=1}^{n} c_j X_j \\ &\text{mit } \sum_{j=1}^{n} a_{ij} X_j \le b_i \text{ für } i = 1, \ldots, m, \\ &\qquad X_j \ge 0 \text{ für } j = 1, \ldots, n. \end{aligned}$$

Wann definiert dies ein ganzzahliges Optimierungsproblem? Zuerst einmal sollten die Koeffizienten c_j, a_{ij}, b_i ganzzahlig sein; aber auch rationale Koeffizienten stellen kein Problem dar, da sich durch Multiplikation aller Koeffizienten mit ihrem Hauptnenner an der Menge der zulässigen bzw. optimalen Lösungen nichts ändert.

Außerdem muß $X_j \in \mathbb{Z}$ für alle j gefordert werden. Diese Forderung ist nur dann sinnvoll, wenn eine Einheit jedes Produktes als unteilbar anzusehen ist. Für einen Autohersteller trifft dies sicherlich zu; eine Brauerei hingegen kann ihre Einheiten sehr flexibel wählen. □

Probleme der Produktionsplanung treten in der Praxis häufig auf. Dennoch sind sie für die ganzzahlige Optimierung selten interessant: Die hergestellten Mengen sind in der Regel so groß, daß es auf eine halbe Einheit nicht ankommt und man die Forderung der Ganzzahligkeit einfach fallen läßt. Dann kann man nämlich auf effiziente Verfahren der (kontinuierlichen) linearen Optimierung, wie z.B. den Simplex-Algorithmus, zurückgreifen; siehe dazu Abschnitt 2.4.

Ohnehin wären viele dieser Probleme als ganzzahlige Optimierungsprobleme praktisch nicht lösbar : Die Zahl der Variablen und Nebenbedingungen geht nicht selten in die Hunderte oder gar Tausende. Nun gehört das ganzzahlige Optimierungsproblem in seiner allgemeinen Form (2.1) zu den sogenannten **NP-vollständigen** Problemen – eine Klasse, die sich dadurch auszeichnet, daß selbst die Lösung kleiner Probleme oft sehr hohe Laufzeiten beansprucht. Wir verzichten auf die genaue Definition dieses Begriffes; sie ist nicht sehr erhellend. Der interessierte Leser kann sich in jedem Informatikbuch zur Komplexitätstheorie schlau machen (z.B. in [22]); speziell den Beweis obiger Aussage findet man in [26], Kapitel 18.

Ein breiteres Anwendungsgebiet findet die ganzzahlige lineare Optimierung bei Modellen, bei denen nur ganzzahlige Lösungen interpretiert werden können. Die zugehörigen Probleme sind oft erst auf den zweiten Blick als ganzzahlige Optimierungsprobleme zu erkennen; in vielen Fällen handelt es sich um **Entscheidungsprobleme,** die mit Hilfe von **0-1-Variablen** modelliert werden. Die im folgenden aufgeführten Probleme mit spezieller Struktur werden in dieser Arbeit als **kombinatorische Probleme** bezeichnet; der Begriff ist nicht unbedingt Literaturstandard.

Beipiel 2.6 (Rucksackprobleme, Knapsack)

1. Ein Einbrecher dringt in eine Wohnung ein und findet dort n Wertgegenstände. Gegenstand j hat einen bestimmten Wert v_j und ein Gewicht w_j. Da der Einbrecher auf der Flucht nicht schlapp machen möchte (im Original: da sein Rucksack nur eine gewisse Tragkraft hat), beschließt er, Sachen mit einem Gesamtgewicht von höchstens W mitgehen zu lassen. Nun steht er vor der Entscheidung, welche der n Sachen er mitnehmen soll, um eine möglichst wertvolle Beute zu machen.

   Als Spezialist für ganzzahlige Optimierung modelliert er das Problem so: Er führt Entscheidungsvariablen X_j mit Werten aus $\{0, 1\}$ ein, die wie folgt interpretiert werden:

$$X_j = \begin{cases} 1, & \text{falls Gegenstand } j \text{ mitgenommen wird,} \\ 0, & \text{sonst.} \end{cases}$$

Damit lautet sein Optimierungsproblem:

$$\max \sum_{j=0}^{n} v_j X_j$$

$$\text{mit } \sum_{j=1}^{n} w_j X_j \leq W,$$

$$X_j \in \{0,1\} \text{ für } j = 1, \ldots, n.$$

Die letzte Zeile läßt sich äquivalent formulieren als

$$0 \leq X_j \leq 1, \; X_j \in \mathbb{Z} \text{ für } j = 1, \ldots, n.$$

2. Unser Einbrecher steht diesmal nicht in einer fremden Wohnung, sondern in der Lagerhalle einer Fabrik. Nun sieht er sich mit der Tatsache konfrontiert, daß jede Art Gegenstand gleich in so vielen Exemplaren vorliegt, daß er gar nicht alle mitnehmen kann.

   Daraufhin ändert er die Interpretation seiner Variablen: X_j bezeichnet nun die *Anzahl* der Exemplare der Art j, die er mitnimmt; gibt es insgesamt n Arten von Gegenständen, so erhält er die Aufgabenstellung

$$\max \sum_{j=0}^{n} v_j X_j$$

$$\text{mit } \sum_{j=1}^{n} w_j X_j \leq W,$$

$$X_j \geq 0, \; X_j \in \mathbb{Z} \text{ für } j = 1, \ldots, n,$$

   mit nach oben unbeschränkten Variablen.

3. Der Einbrecher sieht ein, daß Raubzüge sich nur lohnen, wenn ein gewisser Mindestwert V erbeutet wird. Da er dennoch nicht unnötig viel schleppen möchte, betrachtet er in Zukunft die Aufgabe

$$\min \sum_{j=0}^{n} w_j X_j$$

$$\text{mit } \sum_{j=1}^{n} v_j X_j \geq V,$$

$$(1 \geq) X_j \geq 0, \; X_j \in \mathbb{Z} \text{ für } j = 1, \ldots, n.$$

Als Rucksackprobleme bezeichnet man allgemein IP-Probleme mit nur einer Nebenbedingung. □

Beipiel 2.7 (Problem des Handlungsreisenden oder **Traveling Salesman Problem, TSP)** Ein Geschäftsmann muß Kunden in n verschiedenen Städten aufsuchen. Um von Stadt i nach Stadt j zu gelangen, braucht er Zeit t_{ij}. Er sucht nun eine schnellste Rundreise (Tour), bei der er alle Städte genau einmal aufsucht und an den Anfangsort zurückkehrt.

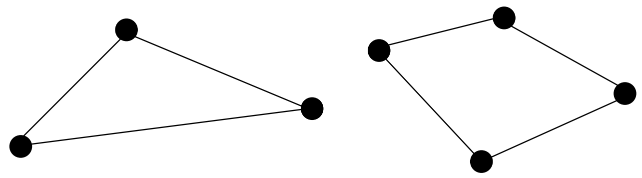
Um eine Formulierung als IP-Problem zu erhalten, führt man Entscheidungsvariablen X_{ij} ein mit

$$X_{ij} = \begin{cases} 1, & \text{falls die Tour von Stadt } i \text{ direkt nach Stadt } j \text{ führt,} \\ 0, & \text{sonst.} \end{cases}$$

Wie kann man damit eine Tour modellieren? Zunächst hat eine Tour die Eigenschaft, daß jede Stadt genau einmal Ausgangsort und genau einmal Zielort ist; dies liefert folgende Nebenbedingungen für die Stadt k:

$$\sum_{j=1}^{n} X_{kj} = 1 \qquad (* \ 1\times \text{ Ausgangsort } *)$$
$$\sum_{i=1}^{n} X_{ik} = 1 \qquad (* \ 1\times \text{ Zielort } *)$$

Aber damit hat man den im Bild dargestellten Fall noch nicht ausgeschlossen; eine Rundreise muß nämlich zusammenhängend sein.



Dazu genügt es zu fordern, daß von jeder nichttrivialen Teilmenge S der Städte eine Verbindung in die komplementäre Teilmenge führt; dies ergibt die Nebenbedingung

$$\sum_{i\in S,\, j\notin S} X_{ij} \geq 1.$$

Damit lautet die Modellierung insgesamt:

$$\min \sum_{i,j=1}^{n} t_{ij} X_{ij}$$
$$\text{mit } \sum_{j=1}^{n} X_{ij} = 1 \ \text{ für } i = 1, \ldots, n,$$
$$\sum_{i=1}^{n} X_{ij} = 1 \ \text{ für } j = 1, \ldots, n,$$
$$\sum_{i\in S,\, j\notin S} X_{ij} \geq 1 \ \ \forall\, \emptyset \subsetneqq S \subsetneqq \{1, \ldots, n\},$$
$$X_{ij} \in \{0, 1\} \text{ für } i, j = 1, \ldots, n.$$

Die letzte Zeile kann durch die Bedingung

$$X_{ij} \geq 0,\ X_{ij} \in \mathbb{Z} \text{ für } i,j = 1,\ldots,n$$

ersetzt werden (ohne Beschränkung von X_{ij} nach oben), da die Gleichheitsrestriktionen garantieren, daß X_{ij} in einer zulässigen Lösung höchstens den Wert 1 hat. □

Beipiel 2.8 (Transportproblem) Ein Bauer hat s Kartoffelfelder und t Lagerräume. Auf Feld i wachsen f_i Traktorladungen Kartoffeln, Lagerraum j kann l_j Ladungen aufnehmen. Der Weg von Feld i zu Lagerraum j hat Länge w_{ij}. Der Bauer möchte nun wissen, wie er die Kartoffeln von den Feldern auf die Lager verteilen soll, so daß er einen möglichst kurzen Gesamtweg fahren muß. Der Einfachheit halber wird angenommen, daß die Lagerkapazität genau für die Ernte ausreicht, d.h.

$$\sum_{i=1}^{s} f_i = \sum_{j=1}^{t} l_j.$$

Bezeichnet X_{ij} die Anzahl der Traktorladungen, die von Feld i zu Lagerraum j gebracht werden, so läßt sich das Problem modellieren als

$$\min \sum_{i=1}^{s} \sum_{j=1}^{t} w_{ij} X_{ij}$$

$$\text{mit } \sum_{j=1}^{t} X_{ij} = f_i \quad \text{für } i = 1,\ldots,s, \ (* \text{ zu verteilende Ernte von Feld } i \, *)$$

$$\sum_{i=1}^{s} X_{ij} = l_j \quad \text{für } j = 1,\ldots,t, \ (* \text{ einzuhaltende Kapazität von Lager } j \, *)$$

$$X_{ij} \geq 0,\ X_{ij} \in \mathbb{Z} \text{ für } i = 1,\ldots,s,\ j = 1,\ldots,t.$$

Diese Modellierung hat Standardform (2.2). In der Matrixform (2.3) geschrieben, erhält man eine $(s+t) \times (s \cdot t)$-Matrix.

Überschreitet die Lagerkapazität die Erntemenge, muß man das „=“ in den Restriktionen für die Lager durch „≤“ ersetzen; bei einer Unterschreitung geht man analog bei den Restriktionen für die Felder vor. □

Beipiel 2.9 (Zuweisungsproblem oder **Assignment Problem)** Oma Erna hat n Tafeln Schokolade für ihre n Enkelkinder gekauft, dummerweise verschiedene Sorten. Kind j ist zu w_{ij} Prozent zufrieden, wenn es Tafel i bekommt. Wie muß Oma Erna die Schokolade verteilen, um möglichst glückliche Gesichter zu sehen?

Mit den Entscheidungsvariablen

$$X_{ij} = \begin{cases} 1, & \text{falls Kind } j \text{ Tafel } i \text{ bekommt,} \\ 0, & \text{sonst,} \end{cases}$$

erhält man die Formulierung als IP-Problem

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} X_{ij}$$

$$\text{mit } \sum_{i=1}^{n} X_{ij} = 1 \quad \text{für } j = 1, \ldots, n, \ (* \text{ eine Tafel für jedes Kind } *)$$

$$\sum_{j=1}^{n} X_{ij} = 1 \quad \text{für } i = 1, \ldots, n, \ (* \text{ jede Tafel geht an genau ein Kind } *)$$

$$X_{ij} \in \{0,1\} \text{ für } i, j = 1, \ldots, n.$$

Wegen der Gleichheitsrestriktionen kann man wiederum die letzte Zeile ersetzen durch

$$X_{ij} \geq 0, \ X_{ij} \in \mathbb{Z} \text{ für } i, j = 1, \ldots, n.$$

Daran erkennt man, daß das Assignment-Problem ein Spezialfall des Transportproblems ist; im Beispiel mit dem Bauer entspricht es dem Fall, daß auf jedem Feld genau eine Traktorladung Kartoffeln wächst und jedes Lager genau eine solche Ladung faßt. □

Weitere Beispiele für besonders strukturierte IP-Probleme findet man in [26] und [27].

2.3 Eine Übersetzung des IP-Problems in die Sprache der Computeralgebra

Im folgenden sei stets $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$, $c \in \mathbb{Z}^n$. Für $j = 1, \ldots, n$ bezeichne $a_j \in \mathbb{Z}^m$ die j-te Spalte von A. Ist $u \in \mathbb{Z}^n$, so seien $u^+, u^- \in \mathbb{Z}^n$ die eindeutig bestimmten Vektoren mit $u^+ \geq 0$, $u^- \geq 0$ (komponentenweise) und $u = u^+ - u^-$, so daß gilt: Für $j = 1, \ldots, n$ ist $u_j^+ \cdot u_j^- = 0$. K sei ein beliebiger Körper (mit $\mathrm{char}(K) \neq 2$, damit $1 \neq -1$ ist).

2.3.1 Torische Ideale

Definition 2.10 Das Ideal

$$I_A := \langle\, x^{u^+} - x^{u^-} \mid u \in \ker(A) \cap \mathbb{Z}^n \,\rangle \subseteq K[x_1, \ldots, x_n]$$

heißt das **torische Ideal** zu A. □

Bemerkung 2.11 Es gilt auch

$$I_A = \langle\, x^v - x^w \mid v - w \in \ker(A),\ v, w \in \mathbb{Z}^n \,\rangle;$$

denn $v - w \in \ker(A) \ \Rightarrow\ x^v - x^w = \gcd(x^v, x^w)\, \underbrace{(x^{(v-w)^+} - x^{(v-w)^-})}_{\in I_A} \in I_A.$ □

Eine weitere Möglichkeit der Definition erhält man über den durch

$$\Pi: \ K[x_1, \ldots, x_n] \longrightarrow K[t_1, t_1^{-1}, \ldots, t_m, t_m^{-1}]; \ x_j \longmapsto t^{a_j}$$

definierten Ringhomomorphismus.

Lemma 2.12 *Es gilt* $I_A = \ker(\Pi)$.

Beweis:

„$\subseteq$“ $u \in \ker(A) \;\Rightarrow\; 0 = \sum_{j=1}^{n} u_j a_j = \sum_{j=1}^{n} u_j^+ a_j - \sum_{j=1}^{n} u_j^- a_j$

$$\Rightarrow\; \Pi(x^{u^+} - x^{u^-}) = t^{\sum_{j=1}^{n} u_j^+ a_j} - t^{\sum_{j=1}^{n} u_j^- a_j} = 0$$

„$\supseteq$“ Wir zeigen, daß $\ker(\Pi)$ von den Binomen $x^v - x^w,\ v - w \in \ker(A),\ v, w \in \mathbb{Z}$, sogar als K-Vektorraum erzeugt wird. Sei dazu $\succ$ eine beliebige, aber feste monomiale Ordnung auf $K[x_1, \ldots, x_n]$.
Annahme: Die Behauptung ist falsch. Dann existiert ein normiertes Polynom $f \in \ker(\Pi)$ mit minimalem Initialterm $\mathrm{in}(f) = x^u$ bzgl. $\succ$, das nicht als K-Linearkombination der $x^v - x^w,\ v - w \in \ker(A),\ v, w \in \mathbb{Z}$, geschrieben werden kann. Da $\Pi(f) = 0$ gilt und Π Monome auf Monome abbildet, muß in f ein *Monom* $x^{\tilde{u}}$ vorkommen mit $\Pi(x^u) = \Pi(x^{\tilde{u}})$. Dann gilt $u - \tilde{u} \in \ker(A)$ (Beweis wie in „$\subseteq$“ mit „$\Leftarrow$“ statt „$\Rightarrow$“); somit kann auch das Polynom $\tilde{f} := f - x^u + x^{\tilde{u}} \in \ker(\Pi)$ nicht als K-Linearkombination der $x^v - x^w,\ v - w \in \ker(A),\ v, w \in \mathbb{Z}$, geschrieben werden. Nach Konstruktion gilt $\mathrm{in}(\tilde{f}) \prec \mathrm{in}(f)$ im Widerspruch zur Minimalität von $\mathrm{in}(f)$.
$\square$

Diese Charakterisierung ist für praktische Zwecke nicht besonders handlich, liefert aber ohne weiteren Aufwand interessante theoretische Ergebnisse:

Satz 2.13 *I_A ist Primideal.*

Beweis: Folgt direkt aus 1.45, 1.47 und 2.12. $\square$

Lemma 2.14 *I_A ist saturiert bezüglich jeder Variablen x_j, $j = 1, \ldots, n$.*

Beweis: Nach Definition von Π gilt $x_j \notin \ker(\Pi) = I_A$ für $j = 1, \ldots, n$. Da I_A prim ist, liegt mit $x_j \cdot f$ also auch f in I_A. Durch Induktion nach k folgt für alle $k \in \mathbb{N}$:

$$x_j^k \cdot f \in I_A \;\Rightarrow\; f \in I_A \qquad \square$$

Definition 2.15 Ein **reines Binom** ist ein Binom mit teilerfremden Monomen und Koeffizienten $+1$ und -1. $\square$

Bemerkung 2.16

1. Für $u \in \mathbb{Z}^n$ ist das Binom $x^{u^+} - x^{u^-}$ ein reines Binom; torische Ideale werden also von reinen Binomen erzeugt.

2. Jedes reine Binom in n Variablen kann in der Form $x^{u^+} - x^{u^-},\ u \in \mathbb{Z}^n$, geschrieben werden. $\square$

Lemma 2.17 *Für $u \in \mathbb{Z}^n$ gilt:*

$$u \in \ker(A) \;\Leftrightarrow\; x^{u^+} - x^{u^-} \in I_A$$

Beweis:

„$\Rightarrow$“ gilt nach Definition von I_A.

„$\Leftarrow$“ Nach 2.12 gilt für $x^{u^+} - x^{u^-} \in I_A$:

$$0 = \Pi(x^{u^+} - x^{u^-}) = t^{\sum_{j=1}^n u_j^+ a_j} - t^{\sum_{j=1}^n u_j^- a_j}$$

$$\Rightarrow \sum_{j=1}^n u_j^+ a_j = \sum_{j=1}^n u_j^- a_j \Rightarrow \sum_{j=1}^n u_j a_j = 0 \Rightarrow u \in \ker(A) \qquad \square$$

Wenn man mit torischen Idealen rechnen will, besteht das erste Problem in der Bestimmung eines endlichen Erzeugendensystems für I_A. Der folgende Satz liefert eine Idee:

Satz 2.18 *Sei $v_1, \ldots, v_r$ ein Erzeugendensystem des Gitters* $\ker(A) \subseteq \mathbb{Z}^n$ und

$$J := \langle\, x^{v_k^+} - x^{v_k^-} \mid k = 1, \ldots . r \rangle.$$

Dann gilt:

$$I_A = J : (x_1 \cdot \ldots \cdot x_n)^\infty = ((\ldots (J : x_1^\infty) : \ldots) : x_n^\infty)$$

Beweis: Die zweite Gleichheit folgt aus 1.42. Beweis der ersten:

„$\supseteq$“ $J \subseteq I_A$ gilt nach Definition von I_A. Somit folgt die Behauptung aus 2.14 und 1.42.

„$\subseteq$“ Gezeigt wird: Ist $v \in \ker(A)$ beliebig, so ist $x^{v^+} - x^{v^-} \in J : (x_1 \cdot \ldots \cdot x_n)^\infty$. v hat eine Darstellung der Form

$$v = \sum_{k=1}^r \lambda_k v_k \text{ mit } \lambda_k \in \mathbb{Z} \text{ für } k = 1, \ldots, n.$$

Also gilt im Laurentpolynomring $K[x_1, x_1^{-1}, \ldots, x_n, x_n^{-1}]$:

$$x^{v^+ - v^-} - 1 = x^{\sum_{k=1}^r ((\lambda_k v_k)^+ - (\lambda_k v_k)^-)} - 1$$

Sei m das *Monom* $x^{(\lambda_1 v_1)^-} \cdot \ldots \cdot x^{(\lambda_r v_r)^-}$. Durch Multiplikation mit $x^{v^-} \cdot m$ erhält man eine Gleichung in $K[x_1, \ldots, x_n]$:

$$m \cdot (x^{v^+} - x^{v^-}) = x^{v^-} \cdot \underbrace{(\prod_{k=1}^r x^{(\lambda_k v_k)^+} - \prod_{k=1}^r x^{(\lambda_k v_k)^-})}_{=:\, b_r}$$

Wir zeigen $b_r \in J$; daraus folgt $m \cdot (x^{v^+} - x^{v^-}) \in J$ und somit die Behauptung $(x^{v^+} - x^{v^-}) \in J : (x_1 \cdot \ldots \cdot x_n)^\infty$. Dazu definieren wir für $t = 1, \ldots, r$

$$b_t := \prod_{k=1}^t x^{(\lambda_k v_k)^+} - \prod_{k=1}^t x^{(\lambda_k v_k)^-}$$

und zeigen $b_t \in J$ durch Induktion nach t. Für $t = 1$ gilt: Im Falle $\lambda_1 = 0$ ist $b_1 = 0$ und die Behauptung trivial. Ist $\lambda_1 \neq 0$, so gelten wegen

$$(\lambda_k v_k)^+ = \begin{cases} |\lambda_k| v_k^+, & \text{falls } \lambda_k > 0, \\ |\lambda_k| v_k^-, & \text{falls } \lambda_k < 0, \end{cases}$$

die Gleichungen:

$$b_1 = x^{(\lambda_1 v_1)^+} - x^{(\lambda_1 v_1)^-} = \pm(x^{(|\lambda_1| v_1^+)} - x^{(|\lambda_1| v_1^-)}) = \pm((x^{v_1^+})^{|\lambda_1|} - (x^{v_1^-})^{|\lambda_1|})$$

$$= \pm \underbrace{(x^{v_1^+} - x^{v_1^-})}_{\in J} \cdot \sum_{l=0}^{|\lambda_1|-1} (x^{v_1^+})^l (x^{v_1^-})^{|\lambda_1|-1-l}$$

Also ist $b_1 \in J$. Ist $t > 1$ und $b_{t-1} \in J$, so erhält man die Darstellung

$$b_t = (\prod_{k=1}^{t-1} x^{(\lambda_k v_k)^+}) \underbrace{(x^{(\lambda_t v_t)^+} - x^{(\lambda_t v_t)^-})}_{\in J (\text{Induktionsanfang})} + x^{(\lambda_t v_t)^-} \underbrace{(\prod_{k=1}^{t-1} x^{(\lambda_k v_k)^+} - \prod_{k=1}^{t-1} x^{(\lambda_k v_k)^-})}_{= b_{t-1} \in J}$$

von b_t als Summe von Polynomen in J. Somit ist auch $b_t \in J$. □

Der folgende Spezialfall ist besonders einfach:

Lemma 2.19 *Mit den Bezeichnungen aus 2.18 gilt: Existiert ein $k \in \{1, \ldots, r\}$, so daß alle Komponenten von v_k strikt positiv sind, so ist $I_A = J$.*

Beweis: Es ist dann $x^{v_k^+} - x^{v_k^-} = x^{v_k} - x^0 = x^{v_k} - 1 \in J$. Da alle Variablen im Monom x^{v_k} involviert sind, sind also alle Variablen invertierbar modulo J. Nach 1.43 ist J bezüglich allen Variablen saturiert, und die Behauptung ergibt sich aus 2.18. □

Im allgemeinen ist das Ideal J aus 2.18 jedoch nicht saturiert:

Beipiel 2.20 Betrachte die Matrix

$$A = \begin{pmatrix} 5 & 1 & 2 & 0 \\ 2 & 1 & 1 & 1 \end{pmatrix} \in \mathbb{Z}^{2\times 4}.$$

Die Vektoren $(1, -1, -2, 1)^\intercal$ und $(0, 2, -1, -1)^\intercal$ bilden eine $\mathbb{R}$-Vektorraumbasis von $\ker(A)$. Sie bilden sogar eine Gitterbasis, denn eine Linearkombination

$$\lambda \begin{pmatrix} 1 \\ -1 \\ 2 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} 0 \\ 2 \\ -1 \\ -1 \end{pmatrix} = \begin{pmatrix} \lambda \\ -\lambda + 2\mu \\ 2\lambda - \mu \\ -\mu \end{pmatrix}$$

liegt genau dann in $\mathbb{Z}^4$, wenn λ und μ in $\mathbb{Z}$ sind.
Der Vektor $(1, 1, -3, 0)^\intercal = (1, -1, -2, 1) + (0, 2, -1, -1)$ liegt in $\ker(A)$; somit gilt $x_1x_2 - x_3^3 \in I_A$. Es ist aber

$$x_1x_2 - x_3^3 \notin J = \langle\, x_1x_4 - x_2x_3^2,\; x_2^2 - x_3x_4 \,\rangle;$$

denn bezüglich der lexikographischen Ordnung bilden $x_1x_4 - x_2x_3^2$ und $x_2^2 - x_3x_4$ eine Gröbnerbasis von J (vgl. 1.27), aber $x_1x_2 - x_3^3$ läßt sich nicht durch eines der beiden Binome reduzieren (d.h. der Rest von $x_1x_2 - x_3^3$ bei der Hironaka-Division ist $x_1x_2 - x_3^3$ selbst). Man rechnet die folgende Darstellung nach:

$$x_4 \cdot (x_1x_2 - x_3^3) = x_2 \cdot (x_1x_4 - x_2x_3^2) + x_3^2 \cdot (x_2^2 - x_3x_4)$$ □

Nun ergibt sich folgendes Schema zur Berechnung von torischen Idealen, das als Eingabe eine Matrix $A \in \mathbb{Z}^{m\times n}$ und eine monomiale Ordnung $\succ$ auf $\mathbb{N}^n$ erhält und die reduzierte Gröbnerbasis von I_A bezüglich $\succ$ berechnet.

Algorithmus 2.21 (Generischer Algorithmus zur Berechnung torischer Ideale)

(1) Berechne eine Gitterbasis $\mathcal{B}$ von $\ker(A)$.

(2) Setze $J := \langle\, x^{v^+} - x^{v^-} \mid v \in \mathcal{B}\,\rangle$.

(3) Berechne die Saturierung $J : (x_1 \cdot \ldots \cdot x_n)^\infty = I_A$.

(4) Berechne die reduzierte Gröbnerbasis von I_A bezüglich $\succ$. □

Wie man eine Gitterbasis für den Kern einer Matrix berechnet, haben wir bereits im ersten Kapitel gesehen. Aber wie berechnet man die Saturierung eines Ideals nach einem Polynom bzw. nach einer Variablen? Diese Frage beantworten die Algorithmen im folgenden Kapitel auf recht unterschiedliche Weise.

2.3.2 Testmengen

Wir betrachten das ganzzahlige Optimierungsproblem in Standardform

$$\min\{\, c^\intercal X \mid X \in \mathbb{Z}^n,\ AX = b,\ X \geq 0\,\}. \tag{2.8}$$

Für den Rest von Abschnitt 2.3 setzen wir voraus, daß $c \geq 0$ ist, also keine negativen Komponenten besitzt, und fassen c als Gewichtsvektor auf. Nach 1.10 kann die durch c definierte Teilordnung dann zu einer monomialen Ordnung $\succ_c$ verfeinert werden. Damit kann man das Optimierungsproblem abwandeln zu

$$\min\nolimits_{\succ_c}\{\, X \mid X \in \mathbb{Z}^n,\ AX = b,\ X \geq 0\,\}, \tag{2.9}$$

wobei die Vektoren $X \in \mathbb{N}^n$ bezüglich $\succ_c$ verglichen werden (vgl. Seite 3). Was ändert sich daduch an der Lösungsmenge? Problem (2.8) ist genau dann lösbar, wenn Problem (2.9) lösbar ist. In diesem Fall existiert nach 1.7 eine optimale Lösung X_0 von (2.9); da $\succ_c$ eine totale Ordnung ist, ist X_0 eindeutig bestimmt. Ferner ist X_0 eine optimale Lösung von (2.8). Letzteres Problem kann jedoch mehrere optimale Lösungen haben; diese unterscheiden sich dann nicht im Gewicht bezüglich c. Welche Lösung Problem (2.9) davon auswählt, wird entschieden durch die Art, wie die Teilordnung c zur monomialen Ordnung verfeinert wird.

Ist man also nur an *einer* optimalen Lösung interessiert, kann man statt (2.8) Problem (2.9) lösen. Dafür wird nun ein Verfahren entwickelt, das den Buchberger-Algorithmus benutzt. Für den Fall, daß man die gesamte Lösungsmenge aufzählen möchte, kann das Verfahren entsprechend erweitert werden; siehe dazu Kapitel 5 in [30].

Im folgenden bezeichne $\mathrm{IP}_{A,\succ_c}(b)$ das Problem (2.9) für feste A, c, b und

$$\mathrm{IP}_{A,\succ_c} := \{\mathrm{IP}_{A,\succ_c}(b) \mid b \in \mathbb{Z}^m\}$$

die Menge der Probleme, die daraus durch Variation von b enstehen.

Definition 2.22 Eine endliche Menge $\mathcal{U} \subseteq \{u \in \mathbb{Z}^n \mid Au = 0,\ u^+ \succ_c u^-\}$ heißt **Testmenge** für $\mathrm{IP}_{A,\succ_c}$, wenn gilt:

1. Ist v eine zulässige, aber nicht optimale Lösung von $\mathrm{IP}_{A,\succ_c}(b)$, $b \in \mathbb{Z}^m$ beliebig, so existiert ein $u \in \mathcal{U}$, für das $v-u$ ebenfalls zulässig für $\mathrm{IP}_{A,\succ_c}(b)$ ist.

2. Ist v_0 die optimale Lösung von $\mathrm{IP}_{A,\succ_c}(b)$, so ist $v_0 - u$ unzulässig für $\mathrm{IP}_{A,\succ_c}(b)$ für alle $u \in \mathcal{U}$. □

Bemerkung 2.23

1. Statt $u^+ \succ_c u^-$ schreiben wir auch $u \succ_c 0$. Allgemein kann man durch die Festlegung
$$u \succ v \ :\Leftrightarrow\ (u-v)^+ \succ (u-v)^-$$
jede monomiale Ordnung $\succ$ zu einer totalen Ordnung auf $\mathbb{Z}^n$ erweitern. Analog zur Verträglichkeit mit der Teilbarkeit ist diese Ordnung additiv, d.h.:
$$u \succ v \ \Rightarrow\ u+w \succ v+w \ \ \forall\, w \in \mathbb{Z}^n$$
Beweisskizze:
Einzig die Transitivität ist nicht ganz offensichtlich; diese weist man über die Additivität nach. Ist $u \succ v$ und $v \succ w$, so addiere ein geeignetes $z \in \mathbb{N}^n$, so daß $u+z,\ v+z,\ w+z \in \mathbb{N}^n$; mit der Transitivität in $\mathbb{N}^n$ folgt $u+z \succ w+z$, durch Addition von $-z$ die Behauptung $u \succ w$.

2. Der Begriff „Testmenge" soll suggerieren, daß man mit einer solchen Menge eine gegebene Lösung auf Optimalität testen kann. Man kann damit sogar aus einer gegebenen zulässigen Lösung v die optimale Lösung konstruieren:

   Solange ein $u \in \mathcal{U}$ existiert, so daß $v-u$ zulässig ist, (∗)
   ersetze v durch $v-u$.

   Wegen $u \succ_c 0$ für alle $u \in \mathcal{U}$ verbessert ein solcher Schritt die Lösung (wenn nicht hier, sieht man das nach dem folgenden Lemma ein). Die Existenz einer optimalen Lösung folgt aus der Existenz einer zulässigen Lösung mittels 1.7. Nicht unmittelbar einsichtig ist die Terminierung des Verfahrens, d.h. daß die Optimallösung nach endlich vielen Schritten erreicht wird; dies ergibt sich aus den folgenden Ausführungen. □

Bemerkung 2.24

1. Im folgenden identifizieren wir den Vektor $u \in \mathbb{Z}^n$ mit dem Binom $x^{u^+} - x^{u^-}$, den Vektor $v \in \mathbb{N}^n$ auch gelegentlich mit dem Monom x^v – darauf wird dann explizit hingewiesen.

2. Wir bezeichnen die Variablen der Optimierungsprobleme mit X, Y, usw., die der Polynomringe entsprechend mit x, y, usw. Einen Vektor $c \in \mathbb{R}^n_{\geq 0}$ können wir sowohl als Kostenfunktion in X als auch als Teilordnung für Monome in x auffassen. In diesem Zusammenhang identifizieren wir dann X und x. □

Lemma 2.25 *Seien $\succ$ eine monomiale Ordnung, $v \in \mathbb{N}^n$ und $u \in \mathbb{Z}^n$, $u \succ 0$, so daß $v - u \geq 0$ komponentenweise gilt. Dividiert man das Monom x^v mit dem Algorithmus von Hironaka durch das Binom $x^{u^+} - x^{u^-}$, so liefert der erste Divisionsschritt den Rest x^{v-u}.*

Beweis: Der Initialterm von $x^{u^+} - x^{u^-}$ bezüglich $\succ$ ist x^{u^+}, da $u \succ 0$ gilt. Aus $v - u \geq 0$ folgt $u \leq v$ und damit $u^+ \leq v^+ = v$. Also ist x^v teilbar durch $\mathrm{in}(x^{u^+} - x^{u^-}) = x^{u^+}$. Ein Divisionsschritt liefert

$$x^v \;=\; x^{v-u^+} \cdot (x^{u^+} - x^{u^-}) \;+\; x^{v-u^+} \cdot x^{u^-}$$

mit Rest $x^{v-u^+} \cdot x^{u^-} = x^{v-u^+ + u^-} = x^{v-u}$. □

Mit der Identifikation 2.24 entspricht die Division eines Monoms durch ein reines Binom einer Subtraktion von Vektoren. Damit ergibt sich die Terminierung des Verfahrens $(*)$ in Bemerkung 2.23 aus der Terminierung des Hironaka-Algorithmus: Für ein u aus einer Testmenge entspricht der Bedingung $v - u \geq 0$ gerade die Bedingung, daß mit v auch $v - u$ zulässig ist.

2.3.3 Das Lösen ganzzahliger Optimierungsprobleme mittels Reduktion durch torische Ideale

Bemerkung 2.26

1. Das S-Paar zweier Binome der Form $b_u = x^{u_1} - x^{u_2}$, $b_v = x^{v_1} - x^{v_2}$ ist wieder ein Binom oder Null. Im Fall eines Binoms sind die Koeffizienten wieder $+1$ und -1:
   O.E. seien $\mathrm{in}(b_u) = x^{u_1}$, $\mathrm{in}(b_v) = x^{v_1}$.

$$\begin{aligned} S(b_u, b_v) &= \frac{x^{v_1}}{\gcd(x^{u_1}, x^{v_1})} \cdot (x^{u_1} - x^{u_2}) - \frac{x^{u_1}}{\gcd(x^{u_1}, x^{v_1})} \cdot (x^{v_1} - x^{v_2}) \\ &= \frac{x^{u_1}}{\gcd(x^{u_1}, x^{v_1})} \cdot x^{v_2} - \frac{x^{v_1}}{\gcd(x^{u_1}, x^{v_1})} \cdot x^{v_1} \end{aligned}$$

   Läßt man bei b_u, b_v beliebige Koeffizienten zu, so kann das Resultat auch ein Term sein: Für $b_u = x - y$, $b_v = x - 3y$ ist $S(b_u, b_v) = 2y$.

2. Das S-Paar zweier *reiner* Binome ist nicht notwendig ein *reines* Binom:

$$S(x^2y - z, xy^2 - z) = xz - yz$$

3. Reduziert man ein Binom $b_v = x^{v_1} - x^{v_2}$ durch ein Binom $b_u = x^{u_1} - x^{u_2}$, so ist der Rest wieder ein Binom oder Null. Im Fall eines Binoms sind die Koeffizienten wieder $+1$ und -1:
   Da eine Reduktion „monomweise" durchgeführt wird, sieht man dies wie im Beweis von 2.25: Das Reduktionsergebnis eines Monoms durch ein Binom mit Koeffizienten $+1$ und -1 ist wieder ein Monom.
   Läßt man bei b_u, b_v beliebige Koeffizienten zu, so kann das Resultat auch ein Term sein: Reduziert man z.B. $b_v = x - y$ durch $b_u = x - 2y$, erhält man den Rest y.

4. Das Ergebnis der Reduktion eines reinen Binoms durch ein reines Binom ist nicht notwendig ein reines Binom: Z.B. gilt für $b_u = x - z$ und $b_v = xy - z$:

$$b_v = y \cdot b_u + (yz - z) \qquad \square$$

Lemma 2.27 *Die reduzierte Gröbnerbasis $\mathcal{G}$ von I_A (bezüglich einer beliebigen, aber festen monomialen Ordnung) besteht aus reinen Binomen.*

Beweis: Startet man den Buchberger-Algorithmus auf einem binomialen Erzeugendensystem von I_A wie in 2.10, so liefert dieser nach 2.26 eine Gröbnerbasis, die nur aus Binomen mit Koeffizienten $+1$ und -1 besteht. Deren Reduktion ändert ebenfalls nichts an dieser Struktur. Aus der Eindeutigkeit der reduzierten Gröbnerbasis folgt, daß $\mathcal{G}$ nur aus Binomen mit Koeffizienten $+1$ und -1 besteht.

Mit dem nichtreinen Binom $b = x^v x^{u^+} - x^v x^{u^-}$, $0 \neq v \in \mathbb{N}^n$, $0 \neq u \in \mathbb{Z}^n$, liegt nach 2.14 auch das reine Binom $\tilde{b} = x^{u^+} - x^{u^-}$ in I_A. Nach Definition von „Gröbnerbasis“ enthält $\mathcal{G}$ ein Binom, dessen Initialterm den von $\tilde{b}$ teilt und somit den von b echt teilt. b kann infolgedessen nicht zu $\mathcal{G}$ gehören, da $\mathcal{G}$ minimal ist. Daher sind alle Binome in $\mathcal{G}$ rein. $\square$

Mit der Identifikation 2.24 kann man die reduzierte Gröbnerbasis als Menge von Vektoren in $\ker(A) \subseteq \mathbb{Z}^n$ auffassen. Damit läßt sich der folgende Satz beweisen, der unmittelbar einen Algorithmus zur Lösung ganzzahliger Optimierungsprobleme liefert.

Satz 2.28 *Die reduzierte Gröbnerbasis $\mathcal{G}_{\succ_c}$ von I_A bezüglich $\succ_c$ ist eine Testmenge für $IP_{A,\succ_c}$.*

Beweis:
Nach 2.17 gilt $Au = 0$ für die Vektoren $u \in \mathcal{G}_{\succ_c}$. Ferner gilt $u^+ \succ_c u^-$, da die Elemente von $\mathcal{G}_{\succ_c}$ normiert sind, d.h. $+1$ der Leitkoeffizient bzgl. $\succ_c$ ist. Zu zeigen bleibt, daß $\mathcal{G}_{\succ_c}$ die Bedingungen 1. und 2. aus 2.22 erfüllt:

1. Seien v eine zulässige, aber nicht optimale Lösung von $\mathrm{IP}_{A,\succ_c}(b)$ und v_0 die optimale Lösung. Wegen $Av = Av_0 = b$ gilt $v - v_0 \in \ker(A)$, somit

$$b = x^{(v-v_0)^+} - x^{(v-v_0)^-} \in I_A.$$

Aus $v \succ_c v_0$ folgt $\mathrm{in}(b) = x^{(v-v_0)^+}$; also gibt es ein $u \in \mathcal{G}_{\succ_c}$ mit $x^{u^+} \mid x^{(v-v_0)^+}$. Man erhält wegen $v, v_0 \geq 0$

$$u^+ \leq (v - v_0)^+ \leq v^+ = v,$$

also ist $v - u$ zulässig.

2. Umgekehrt kann kein $u \in \mathcal{G}_{\succ_c}$ existieren, so daß $v_0 - u$ zulässig ist. Denn wegen $0 \succ_c -u$ wäre $v_0 \succ_c v_0 - u$ im Widerspruch zur Minimalität von v_0. $\square$

Satz 2.29 *Die Testmenge $\mathcal{G}_{\succ_c}$ aus 2.28 ist* **minimal,** *d.h. $\mathcal{G}_{\succ_c} \backslash \{u\}$ ist keine Testmenge für alle $u \in \mathcal{G}_{\succ_c}$.*

Beweis: Sei $u \in \mathcal{G}_{\succ_c}$ beliebig. Betrachte das Problem $\mathrm{IP}_{A,\succ_c}(Au^+)$. u^+ ist zulässig, aber nicht optimal, denn $Au^- = Au^+$ und $u^- \prec_c u^+$. Da $\mathcal{G}_{\succ_c}$ reduziert ist, ist $x^{u^+} - x^{u^-}$ das einzige Element von $\mathcal{G}_{\succ_c}$, dessen Initialterm x^{u^+} teilt. Somit gilt $\{v \in \mathcal{G}_{\succ_c} \,|\, u^+ - v \text{ zulässig}\} = \{u\}$, d.h. u kann nicht weggelassen werden. □

Der vorangehende Satz ist aus praktischer Hinsicht interessant: Er besagt, daß man nicht auf eine kleinere Testmenge hoffen kann – zumindest nicht, wenn b variabel bleiben soll. In [32] versuchen die Autoren, durch Festlegung einer beschränkten Wertemenge für b kleinere Testmengen zu erreichen.

Nun aber zum angekündigten Algorithmus: Dieser bekommt als Eingabe eine zulässige Lösung v (die sogenannte **Initiallösung**) von $\mathrm{IP}_{A,\succ_c}(b)$ und berechnet daraus die optimale Lösung v_0. Wie ein solcher Algorithmus dazu verwendet werden kann, auch eine zulässige Lösung zu bestimmen, wird im nächsten Kapitel gezeigt.

Algorithmus 2.30 (Integer Programming I)

(1) Bestimme die reduzierte Gröbnerbasis $\mathcal{G}_{\succ_c}$ von I_A mit Algorithmus 2.21.

(2) Reduziere x^v modulo $\mathcal{G}_{\succ_c}$ mit dem Algorithmus von Hironaka. Ist x^{v_0} das Ergebnis dieser Reduktion, so ist v_0 die optimale Lösung von $\mathrm{IP}_{A,\succ_c}(b)$.□

Bemerkung 2.31 (Veranschaulichung)
Während der Reduktion in Schritt (2) wandert der Algorithmus von einer zulässigen Lösung zur nächsten. Als Abstiegsrichtungen kommen dabei gerade die Elemente von $\mathcal{G}_{\succ_c}$ in Frage (nach 2.25 entspricht eine Reduktion gerade einer Vektorsubtraktion). Die Nichtnegativität der Lösungen wird dadurch gewahrt, daß diese als Exponentenvektoren von Monomen codiert sind. □

Bemerkung 2.32

1. Besitzt $\mathrm{IP}_{A,\succ_c}(b)$ eine zulässige Lösung, so auch eine optimale, denn die Reduktion in Schritt (2) terminiert stets. Diese Aussage ergibt sich aber auch aus den Voraussetzungen $c \geq 0$ und $X \geq 0$.

2. Das Bestimmen einer Initiallösung ist für kombinatorische Probleme wie im vorangehenden Abschnitt meistens trivial (siehe 2.47). Bei allgemeinen Matrizen ist es jedoch genauso schwierig wie das Lösen von $\mathrm{IP}_{A,\succ_c}(b)$ selbst (vgl. [26], Abschnitte 16.1 und 18.2); es genügt nicht, das Gleichungssystem $AX = b$ mit Gaußelimination zu lösen, da die Lösung erstens ganzzahlig und zweitens nichtnegativ sein muß.

3. Hat man mehrere Initiallösungen v bzw. mehrere rechte Seiten b, für die man das Problem $\mathrm{IP}_{A,\succ_c}(Av)$ bzw. $\mathrm{IP}_{A,\succ_c}(b)$ lösen möchte, braucht man

Schritt (1) trotzdem nur einmal durchzuführen. Ist die Bestimmung einer Initiallösung einfach, so spart man dadurch sehr viel Rechenzeit; denn die Ausführungszeit von Schritt (2) ist im Vergleich zu der von Schritt (1) äußerst gering (siehe Kapitel 6).

4. Um auch die Zielfunktion zu variieren, muß man die Gröbnerbasis von I_A auf eine neue Termordnung umrechnen. Man spart dabei die Schritte (1) bis (3) in Algorithmus 2.21; außerdem verursachen geringfügige Änderungen der Zielfunktion meist auch nur geringfügige oder sogar keine Änderungen der Gröbnerbasis.

   In diesem Zusammenhang sei auf das erste Kapitel in [30] hingewiesen: Dort wird gezeigt, daß man **universelle Gröbnerbasen** berechnen kann, d.h. Erzeugendensysteme, die Gröbnerbasen für alle monomialen Ordnungen sind. Leider sind diese für praktische Zwecke in der Regel viel zu groß.

5. c darf auch ein (nichtnegativer) reeller Vektor sein – der Algorithmus funktioniert trotzdem ohne Änderung. □

Den soeben beschriebenen Ansatz findet man in leicht abgewandelten Formen in [7], [17] und [30].

2.4 Einschub: Lineare Optimierung und der Simplex-Algorithmus

In diesem Abschnitt ist $c \in \mathbb{R}^n$ wieder beliebig.

Ersetzt man in 2.1 überall $\mathbb{Z}$ durch $\mathbb{R}$, läßt also die Forderung nach Ganzzahligkeit der Koeffizienten und insbesondere der Lösung fallen, so erhält man eine Instanz des **(kontinuierlichen) linearen Optimierungsproblems** oder **Linear Programming Problems (LP-Problems).** Definition 2.1 und Bemerkung 2.2 übertragen sich sinngemäß. Das Produktionsplanungsproblem 2.5 kann, wie dort bereits erwähnt, als LP-Problem aufgefaßt werden, die meisten kombinatorischen Probleme hingegen nicht.

Der **Simplex-Algorithmus** ist das klassische Verfahren zur Lösung von LP-Problemen. Ihm liegt eine sehr anschauliche Idee zugrunde: Der Zulässigkeitsbereich einer Instanz des LP-Problems ist, falls nicht leer, ein (nicht notwendig beschränktes) konvexes Polyeder $P \subseteq \mathbb{R}^n$. Die Menge $c^{-1}(z)$ der Vektoren aus $\mathbb{R}^n$, für die die Zielfunktion c einen bestimmten Wert $z \in \mathbb{R}$ annimmt, ist eine affine Hyperebene $H(z) \subseteq \mathbb{R}^n$. Eine Variation von z entspricht einer Parallelverschiebung dieser Hyperebene.

Wir betrachten nun den Fall, daß P beschränkt ist. Dann kann man z so wählen, daß $H(z)$ keinen Punkt von P enthält. Indem man z erhöht oder erniedrigt, schiebt man die Hyperebene an P heran, bis sie zum erstenmal P berührt. Der entsprechende Wert z^* ist dann der Optimalwert von

$$\min \text{ bzw. } \max\{c^{\mathsf{T}} X \mid X \in P\}. \tag{2.10}$$

Diese Konstruktion wird für den zweidimensionalen Fall im linken Bild von Abbildung 2.1 gezeigt.

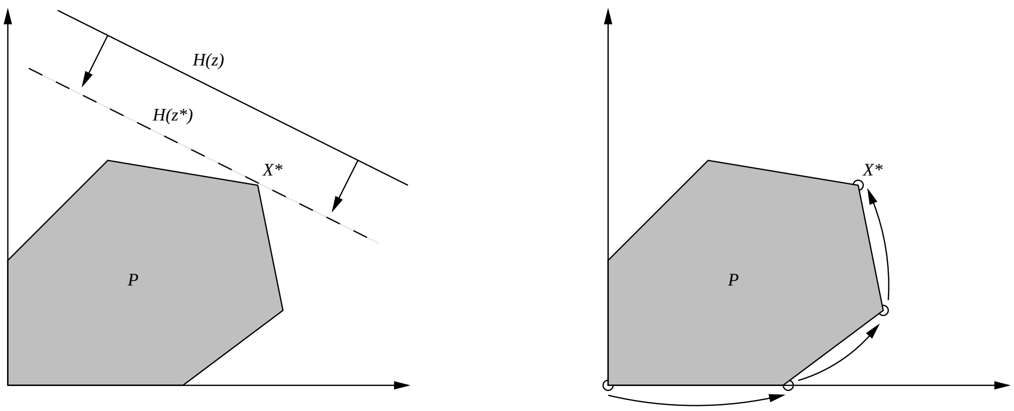


Abbildung 2.1: Geometrie des Simplex-Algorithmus

$H(z^*)$ trifft das Polyeder an einer Ecke X^* (eventuell auch an einer Kante oder einer höherdimensionalen Begrenzungsfläche, aber diese enthält in jedem Fall eine Ecke des *beschränkten* Polyeders). Diese Ecke, als Vektor in $\mathbb{R}^n$ aufgefaßt, ist eine optimale Lösung von (2.10). An dieser Beobachtung setzt der Simplex-Algorithmus an: Er bekommt als Eingabe eine beliebige Ecke des Polyeders. Dann bewegt er sich entlang den Kanten von P von Ecke zu Ecke, wobei der Zielfunktionswert auf den Ecken entlang dieses Weges immer weiter abnimmt. Wenn er an einer Ecke mit optimalem Wert angelangt ist, stoppt der Algorithmus.

Das rechte Bild von Abbildung 2.1 zeigt einen Beispielweg, den der Simplex-Algorithmus in der im linken Bild eingezeichneten Situation nehmen könnte.

Um den Simplex-Algorithmus mathematisch zu formulieren, muß man sich Gedanken über den Umgang mit unlösbaren oder unbeschränkten Instanzen machen. Dennoch genügt einfache Matrizenrechnung, um den Algorithmus exakt aufzuschreiben. Der interessierte Leser findet reichhaltiges Material dazu in [26] oder [27].

Bemerkung 2.33 Das Polyeder P kann exponentiell viele Ecken in der Anzahl n der Variablen haben. Beispielsweise hat der n-dimensionale Einheitswürfel 2^n Ecken, nämlich genau die Vektoren des $\mathbb{R}^n$, die nur Komponenten 0 und 1 haben. Es gibt auch tatsächlich Instanzen, bei denen der Simplex-Algorithmus exponentiell viele Ecken besucht; die erwartete Anzahl dieser Ecken ist jedoch höchstens linear in n, so daß sich der Simplex-Algorithmus in der Praxis als effizient erwiesen hat. Beweise dieser und weiterer Aussagen findet man in Theorem 11.2 und Theorem 11.3 in [26] sowie in Abschnitt 1.6.6 in [27]. □

Bemerkung 2.34

1. Der (vollständig ausformulierte) Simplex-Algorithmus zeigt unter anderem, daß Lemma 2.3 auch für Instanzen des kontinuierlichen LP-Problems gilt; freilich kann der Beweis von dort nicht übertragen werden. Daß die

Zielfunktion c im Falle eines endlichen Infimums dieses tatsächlich annimmt, folgt bei LP-Instanzen aus der Linearität von c und der Nebenbedingungen.

2. Der Simplex-Algorithmus kann auch dazu benutzt werden, eine Initiallösung für die Instanz in Standardform

$$\min\{c^{\mathsf{T}}X \mid AX = b,\ X \geq 0,\ X \in \mathbb{R}^n\} \tag{2.11}$$

mit einer $m \times n$-Matrix A zu finden.

Zunächst kann man erreichen, daß $b_i \geq 0$ für $i = 1, \dots, m$ gilt; ist nämlich $b_i < 0$, so kann man die i-te Gleichung (d.h. die i-te Zeile von A und b) mit -1 multiplizieren. Dadurch ändert sich weder die Lösungsmenge noch verliert die Instanz die Standardform.

Dann betrachtet man die Hilfsinstanz:

$$\min\{1^{\mathsf{T}}Y \mid AX + Y = b,\ X \geq 0,\ Y \geq 0,\ X \in \mathbb{R}^n,\ Y \in \mathbb{R}^m\} \ \Leftrightarrow$$

$$\min\left\{ (0^{\mathsf{T}}, 1^{\mathsf{T}}) \begin{pmatrix} X \\ Y \end{pmatrix} \;\middle|\; (A \mid I_m) \begin{pmatrix} X \\ Y \end{pmatrix} = b,\ \begin{pmatrix} X \\ Y \end{pmatrix} \geq 0,\ \begin{pmatrix} X \\ Y \end{pmatrix} \in \mathbb{R}^{n+m} \right\} \tag{2.12}$$

Dabei ist I_m die $m \times m$-Einheitsmatrix, 0 bezeichnet den Nullvektor in der jeweiligen Dimension und 1 den Vektor in $\mathbb{R}^m$, dessen Komponenten alle $= 1$ sind.

Diese Instanz hat wegen $b \geq 0$ die zulässige Lösung $\binom{X}{Y} = \binom{0}{b}$ und ist durch den Zielfunktionswert 0 nach unten beschränkt. Also besitzt sie eine optimale Lösung.
Hat nun (2.11) eine zulässige Lösung X^0, so ist $\binom{X^0}{0}$ eine optimale Lösung von (2.12), denn $\binom{X^0}{0}$ ist zulässig und die Zielfunktion nimmt darauf den Wert 0 an. Hat andererseits (2.12) eine optimale Lösung $\binom{X^0}{Y^0}$ mit Wert 0, so ist $Y^0 = 0$ und somit X^0 eine zulässige Lösung von (2.11). Insgesamt besitzt (2.11) genau dann eine zulässige Lösung, wenn (2.12) Optimalwert 0 hat. □

Diese Beobachtung liefert einen vollständigen Algorithmus zur Lösung von LP-Problemen in Standardform (2.11) mit $b \geq 0$.

Algorithmus 2.35 (Zweiphasen-Simplex)

(1) **Phase I:** Berechne mit dem Simplex-Algorithmus und der Initiallösung $\binom{0}{b}$ eine optimale Lösung $\binom{X^0}{Y^0}$ von Instanz (2.12).

(2) **Phase II:** Ist $Y^0 \neq 0$, so ist (2.11) unlösbar. Sonst löse (2.11) mit dem Simplex-Algorithmus und der Initiallösung X^0. □

Bemerkung 2.36

1. Im Falle der Unbeschränktheit einer Instanz liefert der Simplex-Algorithmus konkret einen Strahl $\{X^* + zd \mid z \in \mathbb{R}_{\geq 0}\}$, der ganz im Zulässigkeitsbereich liegt und entlang dessen die Zielfunktion immer weiter abnimmt, d.h. $c^\top d < 0$ (d ist **Abstiegsrichtung**).

2. Sind alle Einträge von A, b und c in (2.11) rational, so berechnet der Simplex-Algorithmus eine Lösung, deren Koeffizienten ebenfalls in $\mathbb{Q}$ liegen. □

Dualität

Die Dualitätstheorie liefert wichtige theoretische Sätze und praktische Varianten des Simplex-Algorithmus. Wir geben hier nur einen kleinen Einblick, um später Satz 2.41 einfach beweisen zu können. Für Weiteres sei auf die Literatur verwiesen, z.B. auf Abschnitt 7.4 in [26] und Kapitel 2 in [27].

Definition 2.37 Sei

$$\text{(P)} \quad \min\{c^\top X \mid AX = b,\ X \geq 0,\ X \in \mathbb{R}^n\}$$

ein LP-Problem in Standardform. Das LP-Problem

$$\text{(D)} \quad \max\{b^\top Y \mid A^t Y \leq c,\ Y \in \mathbb{R}^m\}$$

heißt das zu (P) **duale Problem.** (P) wird in diesem Zusammenhang als **primales Problem** bezeichnet. □

Satz 2.38 (Starker Dualitätssatz) (P) ist genau dann optimal lösbar, wenn (D) optimal lösbar ist. In diesem Fall sind die Werte der optimalen Lösungen gleich:

$$\min\{c^\top X \mid AX = b,\ X \geq 0,\ X \in \mathbb{R}^n\} = \max\{b^\top Y \mid A^t Y \leq c,\ Y \in \mathbb{R}^m\}$$

Beweis: Corollary 7.1g in [26] oder Theorem 2.2 in [27]. □

2.5 Zielfunktionen mit negativen Komponenten

Definition 2.39 Seien $c \in \mathbb{R}^n$ beliebig, $A \in \mathbb{R}^{m \times n}$ und $\succ$ eine monomiale Ordnung auf $\mathbb{N}^n$. $\succ$ heißt **kompatibel** (zu c bezüglich A), wenn für alle $v, w \in \mathbb{N}^n$ gilt:

$$Av = Aw \text{ und } c^\top v > c^\top w \quad \Rightarrow \quad v \succ w$$ □

Bemerkung 2.40 Betrachte die Instanz des IP-Problems

$$\min\{c^\top X \mid AX = b,\ X \geq 0,\ X \in \mathbb{Z}^n\} \tag{2.13}$$

mit Zielfunktion $c \in \mathbb{R}^n$, wobei c auch negative Komponenten haben darf. Sei $\succ$ eine monomiale Ordnung, die kompatibel zu c bzgl. A ist. Dann kann man anstatt (2.13) das Problem

$$\min_{\succ}\{X \mid AX = b,\ X \geq 0,\ X \in \mathbb{Z}^n\} \tag{2.14}$$

lösen, denn nach Definition von „kompatibel“ verfeinert $\succ$ die durch c gegebene Teilordnung auf dem Zulässigkeitsbereich (vgl. Probleme (2.8) und (2.9)). Insbesondere folgt aus Bemerkung 2.32, daß (2.13) nicht unbeschränkt sein kann, wenn eine kompatible monomiale Ordnung existiert. □

Man kann also die Methoden aus Abschnitt 2.3 auf allgemeinere Zielfunktionen $c \in \mathbb{R}^n$ anwenden, sofern eine zu c kompatible monomiale Ordnung existiert (und bekannt ist). Wann dies der Fall ist und wie man gegebenenfalls eine kompatible Ordnung berechnen kann, zeigt der folgende Satz.

Satz 2.41 *Hat das LP-Problem*

$$\min\{c^\mathsf{T} X \mid AX = b,\ X \geq 0,\ X \in \mathbb{R}^n\} \tag{2.15}$$

eine optimale Lösung, so existiert eine monomiale Ordnung, die kompatibel zu c bezüglich A ist.

Beweis: Wir konstruieren eine solche als Linearkombination aus c und den Zeilen von A. Nach dem starken Dualitätssatz besitzt mit 2.15 auch das duale Problem

$$\max\{b^\mathsf{T} Y \mid A^\mathsf{T} Y \leq c,\ Y \in \mathbb{R}^m\}$$

eine optimale Lösung Y_0. Definiere $z_0 := (-Y_0, 1)$; dann gilt

$$\mathbb{R}^n \ni w_0 := (A^\mathsf{T} \mid c) z_0 = -A^\mathsf{T} Y_0 + c \geq 0,$$

also läßt sich die Teilordnung w_0 zu einer monomialen Ordnung $\succ_{w_0}$ verfeinern. $\succ_{w_0}$ ist kompatibel zu c bezüglich A: Für $X_1, X_2 \in \mathbb{N}^n$ beliebig gilt nämlich:

$$\begin{aligned} & AX_1 = AX_2 \text{ und } c^\mathsf{T} X_1 > c^\mathsf{T} X_2 \\ \Rightarrow\quad & w_0^\mathsf{T} X_1 = \underbrace{c^\mathsf{T} X_1}_{> c^\mathsf{T} X_2} - Y_0^\mathsf{T} \underbrace{AX_1}_{= AX_2} > c^\mathsf{T} X_2 - Y_0^\mathsf{T} AX_2 = w_0^\mathsf{T} X_2 \end{aligned}$$ □

Definition 2.42 Die **lineare Relaxierung** eines IP-Problems (IP) ist das LP-Problem, daß aus (IP) entsteht, wenn man auf die Forderung der Ganzzahligkeit der Lösung verzichtet. □

Bemerkung 2.43

1. Man kann in Definition 2.39 die Bedingung $v, w \in \mathbb{N}^n$ durch $v, w \in \mathbb{R}^n$ ersetzen, ohne daß sich an den bisher getroffenen Aussagen etwas ändert.

2. Nach dem Satz genügt es für die Existenz einer kompatiblen monomialen Ordnung, daß die lineare Relaxierung (2.15) von (2.13) eine optimale Lösung besitzt. In diesem Falle bleiben für die IP-Instanz (2.13) zwei Möglichkeiten: Entweder sie ist unlösbar oder sie besitzt ebenfalls eine optimale Lösung.

3. Ein Theorem aus [7] (Seite 133) beweist die analoge Aussage für den Fall, daß (2.13) optimal lösbar ist. Die Autoren bringen dort einen geometrischen, aber recht langen Beweis. Aus Satz 2.44 folgt, daß Satz 2.41 dieses Theorem verallgemeinert.

4. Der Beweis des Satzes liefert einen Algorithmus, um gegebenenfalls eine kompatible monomiale Ordnung zu berechnen. Die Lösung des dualen Problems bestimmt man dabei mit dem Simplex-Algorithmus.

2.6 Ein vollständiger Algorithmus

Satz 2.44 *Der Zulässigkeitsbereich des IP-Problems*

$$\text{(IP)} \qquad \min\{c^{\mathsf{T}}X \mid AX = b,\ X \geq 0,\ X \in \mathbb{Z}^n\}$$

sei nicht leer. Dann sind äquivalent:

1. (IP) *ist optimal lösbar.*

2. *Die lineare Relaxierung*

$$\text{(LP)} \qquad \min\{c^{\mathsf{T}}X \mid AX = b,\ X \geq 0\}$$

von (IP) *ist optimal lösbar.*

Beweis:

„$2 \Rightarrow 1$“ Nach Vorraussetzung existiert eine zulässige Lösung für (IP), also ist (IP) entweder optimal lösbar oder unbeschränkt. Letzteres kann nicht vorkommen, da (LP) beschränkt und jede zulässige Lösung von (IP) auch zulässig für (LP) ist.

„$1 \Rightarrow 2$“ Mit (IP) ist auch (LP) zulässig. Wir zeigen:

$$\text{(LP) unbeschränkt} \Rightarrow \text{(IP) unbeschränkt.} \qquad (*)$$

Ist (LP) unbeschränkt, so berechnet der Simplex-Algorithmus nach 2.36 einen Strahl $S^* := \{X^* + zd \mid z \in \mathbb{R}_{\geq 0}\}$, $c^t d < 0$, $d \in \mathbb{Q}^n$, dessen Punkte alle zulässig für (LP) sind.

- Aus $A(X^* + zd) = AX^* + zAd = b = AX^*$ für alle $z \in \mathbb{R}_{\geq 0}$ folgt $Ad = 0$.
- Aus $X^* + zd \geq 0$ für alle $z \geq 0$ folgt $d \geq 0$.

Sei nun $X^0 \in \mathbb{Z}^n$ eine beliebige zulässige Lösung von (IP) (diese existiert nach Voraussetzung).

- Aus $AX^0 = b$ und $Ad = 0$ folgt $A(X^0 + zd) = b$ für alle $z \in \mathbb{R}_{\geq 0}$.
- Aus $X^0 \geq 0$ und $d \geq 0$ folgt $X^0 + zd \geq 0$ für alle $z \geq 0$.

Mit X_0 sind also alle Punkte in

$$S^0 = \{X^0 + zd \mid z \in \mathbb{R}_{\geq 0}\}$$

zulässig für (LP) und damit alle Punkte in

$$S^0_{\mathbb{Z}} := S^0 \cap \mathbb{Z}^n$$

zulässig für (IP). Wegen $d \in \mathbb{Q}^n$ sind dies unendlich viele: Wählt man z als ein Vielfaches des Hauptnenners der Koeffizienten von d, so gilt $X^0 + zd \in \mathbb{Z}^n$. Aus

$$c^{\mathsf{T}}(X^0 + zd) \overset{z\to\infty}{\longrightarrow} -\infty$$

folgt dann die Behauptung $(*)$. □

Der im nächsten Kapitel vorgestellte **Algorithmus von Conti und Traverso** 3.3 liefert in Analogie zum Zweiphasensimplex eine Technik, um zulässige Lösungen eines IP-Problems zu finden. Mit dieser Technik und dem vorangehenden Beweis erhalten wir einen vollständigen Algorithmus, mit dem wir theoretisch IP-Probleme beliebiger Bauart lösen können, d.h. insbesondere ohne Zusatzvoraussetzung an die Zielfunktion. Der Algorithmus erhält als Eingabe eine Instanz (IP) des IP-Problems.

Algorithmus 2.45 (Integer Programming II)

(1) Transformiere (IP) in Standardform $(\overline{\text{IP}})$ gemäß 2.2.

(2) Prüfe die Lösbarkeit von $(\overline{\text{IP}})$ mit dem Conti-Traverso-Algorithmus 3.3.

Ist $(\overline{\text{IP}})$ unlösbar, so melde die Unlösbarkeit von (IP). STOP.

(3) Sonst berechne mit dem Conti-Traverso-Algorithmus eine zulässige Lösung von $(\overline{\text{IP}})$.

(4) Prüfe mit dem Simplex-Algorithmus, ob die lineare Relaxierung von $(\overline{\text{IP}})$ unbeschränkt ist.

Ist dies der Fall, so konstruiere eine Schar $\overline{S}$ von Lösungen von $(\overline{\text{IP}})$ wie im Beweis zu 2.44, die bezüglich der Zielfunktion nach unten unbeschränkt ist. Mache die Transformationen aus Schritt (1) rückgängig und erhalte so eine Lösungsschar S von (IP). Melde die Unbeschränktheit von (IP) und gebe S aus. STOP.

(5) Sonst konstruiere eine zur Zielfunktion kompatible monomiale Ordnung $\succ$ wie im Beweis zu 2.41. Löse $(\overline{\text{IP}})$ mit $\succ$ und Algorithmus 2.30. Transformiere die erhaltene Lösung $\overline{X}^0$ in eine Lösung X^0 von (IP) durch Rücknahme der Transformationen aus Schritt (1). Melde optimale Lösbarkeit und gebe X^0 aus. STOP. □

Die Rücktransformation der unendlichen Schar $\overline{S}$ aus Schritt (4) ist dabei unproblematisch: Der Beweis von 2.44 liefert $\overline{S}$ in Form einer linearen Parametrisierung

$$\overline{S} := \{X^0 + \overline{z}\overline{d} \mid \overline{z} \in \mathbb{N}\},$$

($\overline{d}$ ist d aus dem Beweis, mit dem Hauptnenner der Koeffizienten multipliziert). Transformiert man $\overline{S}$ in dieser Darstellung zurück, so erhält man ebenfalls eine lineare Parametrisierung für das Ergebnis S der Transformation.

Die Algorithmen in Kapitel 3 und die praktischen Diskussionen in den Kapiteln 4 und 5 befassen sich lediglich mit den Schritten (2) und (3) und dem Teil von Schritt (5), der Algorithmus 2.30 entspricht. Ergänzt man das am Ende vorgestellte Programmpaket um eine Implementierung des Simplex-Algorithmus, so kann der vollständige Algorithmus 2.45 damit leicht realisiert werden.

2.7 Ergänzende Bemerkungen

Die beschriebene Übersetzung und die Algorithmen im nächsten Kapitel gehen von einem Problem in Standardform aus. Damit können, wie in Bemerkung 2.2 gesehen, auch Instanzen der allgemeinen Form (2.1) gelöst werden. Allerdings zeigt Beispiel 2.4, daß die Transformation auf Standardform häufig die Variablenzahl vergrößert. Mit der Einführung von zusätzlichen Variablen verbindet sich in der Regel ein dramatischer Effizienzverlust. Daher ist es sinnvoll, auch nach Ansätzen zu suchen, die von einer anderen Form ausgehen. In [33] entwickeln die Autoren einen solchen für Maximierungsprobleme mit Ungleichungsrestriktionen ebenfalls aus Ideen der Computeralgebra.

Bei vielen kombinatorischen Problemen treten Ungleichungen in Form von Variablenbeschränkungen auf, z.B beim Modellieren der häufig vorkommenden 0-1-Variablen. Solche Probleme sollte man nicht in Standardform überführen. Viele Lösungsverfahren können Variablenbeschränkungen sehr effizient behandeln, unter anderem das in [33]; unser Algorithmus 2.30 bietet dafür leider keine erkennbare Möglichkeit.

Für die Beispiele aus Abschnitt 2.2 gelangt man zu sehr unterschiedlichen Aussagen über die Tauglichkeit der erwähnten Ideen:

Beipiel 2.46

1. Das Traveling-Salesman-Problem hat (unter anderem) eine Nebenbedingung für jede nichttriviale Teilmenge der Menge der Städte, also bei n Städten $\geq 2^n - 2$ Stück. Zu seiner Lösung eignet sich unser Ansatz nicht, da hier von einer expliziten Darstellung der Nebenbedingungen ausgegangen wird.

2. Das Rucksackproblem mit 0-1-Variablen eignet sich wegen der oben erklärten Schwierigkeiten eher für den Ansatz aus [33] als für unseren. Die Variante mit unbeschränkten Variablen paßt jedoch gut in unser Konzept.

3. Die Modellierungen des Transport- und Zuweisungsproblems haben bereits die günstige Standardform. Allerdings ist die Variablenzahl als Produkt der Eingabegrößen s, t recht groß. $\square$

Nun gibt es für viele kombinatorische Probleme schnelle Lösungs- oder Approximationsverfahren, für die der Buchberger-Algorithmus keine Konkurrenz darstellt. Insbesondere darf man das Transportproblem mit dem Simplex-Algorithmus lösen, weil die Koeffizientenmatrix **unimodular** ist (d.h. alle Minoren sind 0, 1 oder -1); dann produziert der Simplex-Algorithmus nämlich ganzzahlige Lösungen (vgl z.B. Kapitel 19 in [26]). Es scheint also, als läge das Einsatzgebiet für unsere Algorithmen eher im Bereich von Problemen mit unregelmäßigen Matrizen. Allerdings wird das Testmaterial zeigen, daß der Buchberger-Algorithmus Regelmäßigkeiten in der Eingabe erfreulich gut verarbeitet. Außerdem ist die Bestimmung einer Initiallösung für ein kombinatorisches Problem meistens eine einfache Aufgabe, was nach Bemerkung 2.32 sehr vorteilhaft ist:

Beipiel 2.47 (Bestimmung von Initiallösungen)

1. Für das Traveling-Salesman-Problem wählt man irgendeine Permutation der Städte als Ausgangslösung.

2. Beim Rucksackproblem (Varianten 1 und 2) ist eine zulässige Lösung dadurch gegeben, daß der Einbrecher nichts mitnimmt.

3. Beim Transportproblem bestimmt man eine Initiallösung über die **"Nordwesteckenregel"** (vgl. Seite 216 in [10]). Man schreibt die Variablen als $s \times t$-Matrix:
$$\begin{pmatrix} X_{1,1} & X_{1,2} & \cdots & X_{1,t} \\ X_{2,1} & X_{2,2} & \cdots & X_{2,t} \\ \vdots & \vdots & \ddots & \vdots \\ X_{s,1} & X_{s,2} & \cdots & X_{s,t} \end{pmatrix}$$
Die Aufgabe besteht darin, den Variablen Werte zuzuordnen, so daß mit den Bezeichnungen aus 2.8

   - die Summe der Werte in Zeile i gleich f_i und
   - die Summe der Werte in Spalte j gleich l_j ist.

   Ist nun $f_1 \geq l_1$, so setzt man $X_{1,1} = l_1$ und $X_{i,1} = 0$ für $i = 2, \ldots, s$. Dann startet man rekursiv das Verfahren auf der $s \times (t-1)$-Matrix
$$\begin{pmatrix} X_{1,2} & \cdots & X_{1,t} \\ X_{2,2} & \cdots & X_{2,t} \\ \vdots & \ddots & \vdots \\ X_{s,2} & \cdots & Xs,t \end{pmatrix}$$
mit Zeilensummen $f_1 - l_1, f_2, \ldots, f_s$ und Spaltensummen $l_2, \ldots, l_t$.
Ist $f_1 < l_1$, so setzt man $X_{1,1} = f_1$ und $X_{1,j} = 0$ für $j = 2, \ldots, t$. Dann startet man rekursiv das Verfahren auf der $(s-1) \times t$-Matrix
$$\begin{pmatrix} X_{2,1} & X_{2,2} & \cdots & X_{2,t} \\ \vdots & \vdots & \ddots & \vdots \\ X_{s,1} & X_{s,2} & \cdots & X_{s,t} \end{pmatrix}$$
mit Zeilensummen $f_2, \ldots, f_s$ und Spaltensummen $l_1 - f_1, l_2, \ldots, l_t$. □

Kapitel 3

Algorithmen

Dieses Kapitel beschäftigt sich mit verschiedenen Algorithmen zur ganzzahligen Optimierung, die auf dem Ansatz aus Abschnitt 2.3 basieren. All diese Algorithmen gehen von dem ganzzahligen Optimierungsproblem

$$\min\{c^{\mathsf{T}}X \mid AX = b,\ X \geq 0,\ X \in \mathbb{Z}^n\} \tag{IP}$$

mit $c \geq 0$ aus. Sie sind im wesentlichen Ausprägungen des generischen Algorithmus 2.30. Der zuerst vorgestellte Algorithmus kombiniert diesen mit der Idee des Zweiphasensimplex.

3.1 Der Algorithmus von Conti und Traverso

Der in diesem Abschnitt vorgestellte Algorithmus wird in [7] auf eine recht elementare Weise hergeleitet. Im Sinne einer einheitlichen Notation greifen wir stattdessen auf die Ergebnisse aus Abschnitt 2.3 zurück.

3.1.1 Der nichtnegative Fall

Sei $A \in \mathbb{Z}^{m\times n}$ eine Matrix mit nichtnegativen Einträgen. Wir betrachten die erweiterte Matrix

$$\tilde{A} := \left(\begin{array}{c|c} I_m & A \end{array}\right) \in \mathbb{Z}^{m\times(m+n)},$$

wobei I_m die $(m \times m)$-Einheitsmatrix ist. $\tilde{A}$ hat Rang m, also hat $\ker(\tilde{A})$ die Dimension n. Bezeichnen wiederum $a_j \in \mathbb{Z}^m$ die j-te Spalte von A und $e_j \in \mathbb{Z}^n$ den j-ten Einheitsvektor, so liegen die Vektoren

$$l_j := \begin{pmatrix} a_j \\ -e_j \end{pmatrix},\ j = 1, \ldots, n,$$

im Kern von $\tilde{A}$ und sind linear unabhängig (man betrachte die letzten n Komponenten). Also bilden $l_1, \ldots, l_n$ eine Basis des $\mathbb{R}$-Vektorraums $\ker(\tilde{A})$ und sogar eine Gitterbasis des $\mathbb{Z}$-Moduls $\ker(\tilde{A})$: Ist nämlich

$$z = (z_1, \ldots, z_m, z_{m+1}, \ldots, z_{m+n})^{\mathsf{T}} \in \mathbb{Z}^{m+n}$$

ein beliebiger ganzzahliger Vektor in $\ker(\tilde{A})$, so gilt

$$(z_{m+1},\ldots,z_{m+n}) = \sum_{j=1}^{n}(-z_{m+j})(-e_j),$$

und diese Darstellung ist eindeutig. Da $l_1,\ldots,l_n$ eine Vektorraumbasis des Kerns bilden, muß sogar gelten

$$z = \sum_{j=1}^{n}(-z_{m+j})l_j,$$

also ist z eine *ganzzahlige* Linearkombination der l_j.

Nun betrachtet man das torische Ideal von $\tilde{A}$ in $K[y_1,\ldots,y_m,x_1,\ldots,x_n]$.

Lemma 3.1 $I_{\tilde{A}} = \langle\, y^{a_j} - x_j \mid j = 1,\ldots,n\,\rangle$

Beweis: Die Inklusion „$\supseteq$" gilt nach Definition von $I_{\tilde{A}}$, denn das Binom $y^{a_j} - x_j$ enstpricht über die Identifikation 2.24 gerade dem Vektor l_j. Für die Inklusion „$\subseteq$" bezeichne J das Ideal auf der rechten Seite, a_{ij} den Koeffizient von A in Zeile i und Spalte j, und wie oben sei z ein beliebiger ganzzahliger Vektor im Kern von $\tilde{A}$. Es ist zu zeigen

$$\left(\prod_{1\le i\le m} y_i^{z_i^+}\right)\cdot\left(\prod_{1\le j\le n} x_j^{z_{m+j}^+}\right) - \left(\prod_{1\le i\le m} y_i^{z_i^-}\right)\cdot\left(\prod_{1\le j\le n} x_j^{z_{m+j}^-}\right) \in J,$$

oder, äquivalent,

$$\left(\prod_{1\le i\le m} y_i^{z_i^+}\right)\cdot\left(\prod_{1\le j\le n} x_j^{z_{m+j}^+}\right) - \left(\prod_{1\le i\le m} y_i^{z_i^-}\right)\cdot\left(\prod_{1\le j\le n} x_j^{z_{m+j}^-}\right) \equiv 0 \bmod J,$$

Wegen $x_j \equiv y^{a_j} \bmod J$ erhält man:

$$\begin{aligned}
& \left(\prod_{1\le i\le m} y_i^{z_i^+}\right)\cdot\left(\prod_{1\le j\le n} x_j^{z_{m+j}^+}\right) - \left(\prod_{1\le i\le m} y_i^{z_i^-}\right)\cdot\left(\prod_{1\le j\le n} x_j^{z_{m+j}^-}\right)\\
\equiv\ & \left(\prod_{1\le i\le m} y_i^{z_i^+}\right)\cdot\left(\prod_{1\le j\le n} (y^{a_j})^{z_{m+j}^+}\right) - \left(\prod_{1\le i\le m} y_i^{z_i^-}\right)\cdot\left(\prod_{1\le j\le n} (y^{a_j})^{z_{m+j}^-}\right)\\
=\ & \prod_{i=1}^{m} y_i^{z_i^+ + \sum_{j=1}^n z_{m+j}^+ a_{ij}} - \prod_{i=1}^{m} y_i^{z_i^- + \sum_{j=1}^n z_{m+j}^- a_{ij}}\\
=\ & 0,
\end{aligned}$$

denn die Exponenten von y_i sind in beiden Monomen gleich für $i = 1,\ldots,m$:

$$\begin{aligned}
z \in \ker(\tilde{A}) \quad & \Rightarrow \quad z_i + \sum_{j=1}^{n} a_{ij} z_{m+j} = 0 \text{ für } i = 1,\ldots,m\\
& \Rightarrow \quad z_i^+ + \sum_{j=1}^{n} a_{ij} z_{m+j}^+ = z_i^- + \sum_{j=1}^{n} a_{ij} z_{m+j}^- \text{ für } i = 1,\ldots,m
\end{aligned}$$

□

Wie löst man nun das ganzzahlige Optimierungsproblem (IP) mit Hilfe dieser Matrix? Wir beschränken uns zunächst auf den Fall $b \geq 0$ und betrachten das Gleichungssystem mit der erweiterten Matrix:

$$\tilde{A}\begin{pmatrix} Y \\ X \end{pmatrix} = \begin{pmatrix} I_m & \mid & A \end{pmatrix}\begin{pmatrix} Y \\ X \end{pmatrix} = b,\ X, Y \geq 0,\ X \in \mathbb{Z}^n,\ Y \in \mathbb{Z}^m \qquad (3.1)$$

Die zulässigen Lösungen X von (IP) entsprechen eineindeutig den zulässigen Lösungen $\binom{Y}{X}$ von (3.1) mit $Y = 0$. Wegen $b \geq 0$ kann man für (3.1) sofort eine Initiallösung ablesen, nämlich

$$\begin{pmatrix} Y \\ X \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}.$$

Nun muß man eine Kostenfunktion für das erweiterte System (3.1) finden, bezüglich der dieses die optimale Lösung $\binom{0}{X_0}$ hat, wobei X_0 optimale Lösung von (IP) ist. Ein naheliegender Ansatz besteht darin, die Kostenfunktion c auf den Variablen X beizubehalten und die Variablen Y so teuer zu machen, daß diese in einer Optimallösung Wert 0 haben; d.h. man sucht eine Kostenfunktion $\tilde{c}$ der Form

$$\tilde{c}^{\mathsf{T}} = (\underbrace{M, \ldots, M}_{m \text{ Komponenten}}, c^{\mathsf{T}}),$$

wobei $M \in \mathbb{R}$ hinreichend groß ist. Das Problem liegt dann in der konkreten Wahl von M. Die Übersetzung der Kostenfunktion in eine Termordnung liefert uns eine elegante Möglichkeit, M „unendlich groß" zu machen: Man wählt eine Ordnung $\succ$, die die Elimininationseigenschaft bezüglich den Variablen y erfüllt und die Kostenfunktion c auf den Variablen x verfeinert (x, y sind die X, Y entsprechenden Variablen des Polynomrings, vgl. 2.24). Eine solche erhält man beispielsweise als Blockordnung wie in 1.33.

Nun ergibt sich folgender Algorithmus, der als Eingabe einen Kostenvektor $c \geq 0$, eine Matrix A mit nichtnegativen Einträgen und einen Begrenzungsvektor $b \geq 0$ erhält und die Lösung von (IP) berechnet:

Algorithmus 3.2 (Conti-Traverso-Algorithmus für den nichtnegativen Fall)

(1) Setze $I := \langle\, y^{a_j} - x_j \mid j = 1, \ldots, n \,\rangle$.

(2) Berechne die reduzierte Gröbnerbasis $\mathcal{G}_\succ$ von I bezüglich einer Termordnung $\succ$, die die Eliminationseigenschaft bezüglich $y_1, \ldots, y_m$ hat und auf $x_1, \ldots, x_n$ die Kostenfunktion c verfeinert.

(3) Reduziere y^b modulo $\mathcal{G}_\succ$ mit dem Algorithmus von Hironaka, etwa zu $y^d x^e$.

(4) Ist $\mathbb{N}^m \ni d \neq 0$, so ist (IP) unlösbar. Ist $d = 0$, so ist $e \in \mathbb{N}^n$ die optimale Lösung von (IP). □

Die Variablen y in obigem Algorithmus erfüllen die gleiche Funktion wie die Variablen Y im Zweiphasensimplex. Die Übersetzung der linearen Kostenfunktion in eine monomiale Ordnung erlaubt es jedoch, die beiden Phasen (Lösbarkeitstest bzw. Bestimmung einer Initiallösung sowie Reduktion der Initiallösung zur Optimallösung) zusammenzufassen. Man kann den Conti-Traverso-Algorithmus auch in zwei Phasen durchführen; das ist z.B. im Rahmen von Algorithmus 2.45 sinnvoll, wenn der Kostenvektor negative Komponenten hat und die Beschränktheit der IP-Instanz nicht im voraus feststeht. Diese kann dann zwischen den beiden Phasen getestet werden.

Algorithmus 3.3 (Zweiphasen-Conti-Traverso)

(1) **Phase I:** Setze $I := \langle\, y^{a_j} - x_j \mid j = 1, \ldots, n \,\rangle$. Berechne die reduzierte Gröbnerbasis $\mathcal{G}_\succ$ von I bezüglich einer Termordnung $\succ$, die die Eliminationseigenschaft bezüglich $y_1, \ldots, y_m$ hat. Reduziere y^b modulo $\mathcal{G}_\succ$ mit dem Algorithmus von Hironaka, etwa zu $y^d x^e$. Ist $\mathbb{N}^m \ni d \neq 0$, so ist (IP) unlösbar $\Rightarrow$ STOP.
Sonst ist x_e zulässige Lösung.

(2) **Phase II:** Setze

$$J := \langle\, g \mid g \in \mathcal{G}_\succ \cap K[x_1, \ldots, x_n] \,\rangle \subseteq K[x_1, \ldots, x_n].$$

Berechne die reduzierte Gröbnerbasis $\mathcal{G}_{\succ_c}$ von J bezüglich einer Termordnung $\succ_c$, die c verfeinert. Reduziere x^e modulo $\mathcal{G}_{\succ_c}$, etwa zu x^f. f ist dann die optimale Lösung von (IP). □

Die Korrektheit dieser Variante folgt aus Satz 1.34 und der Gleichheit $J = I_A$; letztere ergibt sich aus nachstehendem Lemma.

Lemma 3.4 $I_A = I_{\tilde{A}} \cap K[x_1, \ldots, x_n]$

Beweis:

„$\subseteq$" I_A wird erzeugt von den Binomen $x^{v^+} - x^{v^-}$, $v \in \ker(A)$. Mit $v \in \ker(A)$ ist $\binom{0}{v} \in \ker(\tilde{A})$, wobei 0 der Nullvektor von $\mathbb{Z}^m$ ist. Nach Definition von $I_{\tilde{A}}$ folgt $x^{v^+} - x^{v^-} = y^{0^+} x^{v^+} - y^{0^-} x^{v^-} \in I_{\tilde{A}}$.

„$\supseteq$" Nach 2.17 und 2.27 besteht eine reduzierte Gröbnerbasis von $I_{\tilde{A}}$ aus Binomen der Form $y^{u_Y} x^{u_X} - y^{v_Y} x^{v_X}$, für die $\binom{u_Y}{u_X} - \binom{v_Y}{v_X} \in \ker(\tilde{A})$ gilt. Berechnet man die reduzierte Gröbnerbasis bezüglich einer Eliminationsordnung für y, so wird nach 1.34 das Ideal $I_{\tilde{A}} \cap K[x_1, \ldots, x_n]$ von deren Elementen erzeugt, die nur x-Variablen involvieren, d.h. von Binomen der Form $x^{u_X} - x^{v_X}$ mit $\binom{0}{u_X} - \binom{0}{v_X} \in \ker(\tilde{A})$. Nach Konstruktion von $\tilde{A}$ folgt $u_X - v_X \in \ker(A)$, also $x^{u_X} - x^{v_X} \in I_A$. □

Die künstlichen Initiallösungen $\binom{b}{0}$ des erweiterten Systems (3.1) sind in der Regel „weit von der Optimallösung entfernt" im Vergleich zu einer Initiallösung des ursprünglichen Systems (IP), d.h. die Anzahl der Reduktionsschritte hin zur

Optimallösung ist weitaus höher. Bei der Lösung einer Instanz, zu der bereits eine Initiallösung von (IP) bekannt ist, möchte man also nicht unbedingt von der künstlichen Lösung aus starten. Obiges Lemma liefert auch für diesen Fall eine passende Variante des Conti-Traverso-Algorithmus. Als Eingabe erhält sie statt einer rechten Seite $b \geq 0$ eine Initiallösung $\mathbb{Z}^n \ni e \geq 0$.

Algorithmus 3.5 (Conti-Traverso-Algorithmus mit Elimination)

(1) Setze $I := \langle\, y^{a_j} - x_j \mid j = 1, \ldots, n \,\rangle$.

(2) Berechne die reduzierte Gröbnerbasis $\mathcal{G}_\succ$ von I bezüglich einer Termordnung $\succ$, die die Eliminationseigenschaft bezüglich $y_1, \ldots, y_m$ hat und auf $x_1, \ldots, x_n$ die Kostenfunktion c verfeinert.

(3) Setze $\mathcal{G}_{\succ_c} := \mathcal{G}_\succ \cap K[x_1, \ldots, x_n]$.

(4) Reduziere x^e modulo $\mathcal{G}_\succ$ mit dem Algorithmus von Hironaka, etwa zu x^f.

(5) $f \in \mathbb{N}^n$ ist die optimale Lösung von (IP). □

3.1.2 Der allgemeine Fall

In diesem Abschnitt sei $A \in \mathbb{Z}^{m \times n}$ eine Matrix mit beliebigen ganzzahligen Einträgen. Wir erweitern $\tilde{A}$ wie im vorigen Abschnitt um eine weitere Spalte zur Matrix

$$\bar{A} = \begin{pmatrix} -1 & | & \tilde{A} \end{pmatrix} = \begin{pmatrix} -1 & | & I_m & | & A \end{pmatrix} \in \mathbb{Z}^{m \times (1+m+n)},$$

wobei $-1 \in \mathbb{Z}^m$ den Vektor bezeichne, dessen Komponenten alle gleich -1 sind. Erweitert man die l_j aus Abschnitt 3.1.1 um eine erste Komponente mit Wert 0, so liegen die entstandenen Vektoren $\bar{l}_j$ im Kern von $\bar{A}$. $\ker(\bar{A})$ hat Dimension $n + 1$; definiert man

$$\bar{l}_{n+1} := (\underbrace{1, \ldots, 1}_{m+1}, \underbrace{0, \ldots, 0}_{n}),$$

so bilden die $\bar{l}_j$, $j = 1, \ldots, n+1$ eine Gitterbasis von $\ker(\bar{A})$ (man betrachte die erste und die letzten n Komponenten).

Nun betrachtet man das torische Ideal von $\bar{A}$ in $K[t, y_1, \ldots, y_m, x_1, \ldots, x_n]$. a_j bezeichne wiederum die j-te Spalte von A.

Lemma 3.6 $I_A = \langle\, \{\, y^{a_j^+} - y^{a_j^-} x_j \mid j = 1, \ldots, n \,\} \cup \{\, t \cdot y_1 \cdot \ldots \cdot y_m - 1 \,\} \,\rangle$

Beweis: Die angegebenen Binome entsprechen mit der Identifikation 2.24 den Gitterbasisvektoren $\bar{l}_1, \ldots, \bar{l}_{n+1}$. Nach 2.18 ist also nur zu zeigen, daß das von ihnen erzeugte Ideal saturiert ist. Wegen $t \cdot y_1 \cdot \ldots \cdot y_m - 1 \in I_{\bar{A}}$ sind die Variablen t und $y_1, \ldots, y_m$ invertierbar modulo $I_{\bar{A}}$ und damit auch die Produkte $y^{a_j^+}$, $j = 1, \ldots, n$. Sei $g_j \in K[t, y_1, \ldots, y_m, x_1, \ldots, x_n]$ das Inverse von $y^{a_j^+}$ modulo $I_{\bar{A}}$; dann folgt:

$$y^{a_j^+} - y^{a_j^-} x_j \in I_{\bar{A}} \;\Rightarrow\; 0 \equiv g_j (y^{a_j^+} - y^{a_j^-} x_j) \equiv 1 - g_j y^{a_j^-} x_j \bmod I_{\bar{A}},$$

also ist auch x_j invertierbar modulo $I_{\bar{A}}$. Damit sind alle Variablen invertierbar, und die Behauptung folgt aus 1.43. □

Mit der erweiterten Matrix $\bar{A}$ kann man das ganzzahlige Optimierungsproblem (IP) für beliebige rechte Seiten $b \in \mathbb{Z}^m$ lösen. Man geht dabei wieder von einem Gleichungssystem mit $\bar{A}$ aus:

$$\bar{A}\begin{pmatrix} T \\ Y \\ X \end{pmatrix} = (-1|\, I_m\, |A)\begin{pmatrix} T \\ Y \\ X \end{pmatrix} = b,\ X,Y,T \geq 0,\ X \in \mathbb{Z}^n,\ Y \in \mathbb{Z}^m,\ T \in \mathbb{Z} \tag{3.2}$$

Die zulässigen Lösungen X von (IP) entsprechen eineindeutig den zulässigen Lösungen $(T, Y^\mathsf{T}, X^\mathsf{T})$ von (3.2) mit $T = 0$ und $Y = 0$. Eine Initiallösung von (3.2) ist gegeben durch

$$\begin{pmatrix} T \\ Y \\ X \end{pmatrix} = \begin{pmatrix} d \\ b+d \\ 0 \end{pmatrix}$$

wobei $b + d$ als Addition von d zu jeder Komponente von b zu verstehen ist und $\mathbb{R} \ni d \geq 0$ so gewählt wird, daß $b_i + d \geq 0$ für $i = 1, \ldots, m$ gilt (also z.B. $d = -\min\{\, b_i \mid 1 \leq i \leq m,\ b_i \leq 0\,\}$).

Analog zum nichtnegativen Fall findet man eine Kostenfunktion für das erweiterte System, bezüglich der dieses die optimale Lösung

$$\begin{pmatrix} T \\ Y \\ X \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ X_0 \end{pmatrix}$$

hat, wobei X_0 optimale Lösung von (IP) ist: Man wählt eine Ordnung $\succ$, die die Elimininationseigenschaft bezüglich den Variablen t und y erfüllt und die Kostenfunktion c auf den Variablen x verfeinert (hier sind wir wieder im Polynomring, vgl. 2.24).

Der allgemeine Conti-Traverso-Algorithmus erhält als Eingabe einen Kostenvektor $c \geq 0$ sowie eine Matrix A und eine rechte Seite b mit beliebigen ganzzahlige Einträgen und berechnet daraus die Lösung von (IP):

Algorithmus 3.7 (Conti-Traverso-Algorithmus)

(1) Setze $I := \langle\, \{\, y^{a_j^+} - y^{a_j^-} x_j \mid j = 1, \ldots, n\,\} \cup \{\, t \cdot y_1 \cdot \ldots \cdot y_m - 1\,\}\,\rangle$.

(2) Berechne die reduzierte Gröbnerbasis $\mathcal{G}_\succ$ von I bezüglich einer Termordnung $\succ$, die die Eliminationseigenschaft bezüglich $t, y_1, \ldots, y_m$ hat und auf $x_1, \ldots, x_n$ die Kostenfunktion c verfeinert.

(3) Setze $d = -\min\{\, b_i \mid 1 \leq i \leq m,\ b_i \leq 0\,\}$.

(4) Reduziere $t^d y^{b+d}$ modulo $\mathcal{G}_\succ$ mit dem Algorithmus von Hironaka, etwa zu $t^e y^f x^g$.

(5) Ist $\mathbb{N} \ni e \neq 0$ oder $\mathbb{N}^m \ni f \neq 0$, so ist (IP) unlösbar. Sonst ist $g \in \mathbb{N}^n$ die optimale Lösung von (IP). □

Die Varianten 3.3 und 3.5 für den nichtnegativen Fall lassen sich problemlos auf den allgemeinen Fall übertragen, ebenso Lemma 3.4.

3.2 Der Algorithmus von Pottier

Der Algorithmus, den wir in diesem Abschnitt vorstellen, stammt aus der ältesten Quelle unserer Sammlung ([25]), die torische Ideale mit Hilfe von Gitterbasen berechnet.

Sei $A \in \mathbb{Z}^{m\times n}$ und $v_1, \ldots, v_r$ eine Gitterbasis von $\ker(A)$. Dann gilt nach 2.18 im Polynomring $K[x_1, \ldots, x_n]$:

$$I_A = \langle\, x^{v_k^+} - x^{v_k^-} \mid k = 1, \ldots, r\rangle : (x_1 \cdot \ldots \cdot x_n)^\infty$$

Nun gehen wir analog zum Übergang vom nichtnegativen zum allgemeinen Fall beim Conti-Traverso-Algorithmus vor: Wir führen eine Hilfsvariable t ein und betrachten das Ideal J in $K[t, x_1, \ldots, x_n]$, das von den $x^{v_k^+} - x^{v_k^-}$, $k = 1, \ldots, r$, und dem Binom $t \cdot x_1 \cdot \ldots, \cdot x_n - 1$ erzeugt wird. Wegen dem letzten Erzeuger und dem Beweis von 1.43 ist J saturiert nach allen Variablen. In der Tat handelt es sich bei J um das torische Ideal einer erweiterten Matrix, nämlich von

$$\tilde{A} := \left(\begin{array}{c|ccc} -\sum_{j=1}^n a_{1,j} & a_{1,1} & \ldots & a_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ -\sum_{j=1}^n a_{m,j} & a_{m,1} & \ldots & a_{m,n} \end{array} \right).$$

Der Kern von $\tilde{A}$ hat Dimension $\dim(\ker(A)) + 1 = r + 1$, und die Vektoren $\binom{0}{v_i}$, $i = 1, \ldots, r$, und $(1, \ldots, 1)^\top \in \mathbb{Z}^{n+1}$ bilden eine Gitterbasis dieses Kerns.

Lemma 3.8 $I_A = J \cap K[x_1, \ldots, x_n]$

Beweis: Völlig analog zu 3.4. □

Daraus ergibt sich der folgende Algorithmus zur Lösung von (IP), der als Eingabe den nichtnegativen Kostenvektor c, die Matrix A und eine Initiallösung $\mathbb{Z}^n \ni e \geq 0$ von (IP) erhält:

Algorithmus 3.9 (Algorithmus von Pottier)

(1) Berechne eine Gitterbasis $v_1, \ldots, v_r$ von $\ker(A)$.

(2) Setze $J := \langle \{\, x^{v_k^+} - x^{v_k^-} \mid k = 1, \ldots, r\,\} \cup \{t \cdot x_1 \cdot \ldots \cdot x_n - 1\}\rangle$.

(3) Berechne die reduzierte Gröbnerbasis $\mathcal{G}_\succ$ von J bezüglich einer Termordnung $\succ$, die die Eliminationseigenschaft bezüglich t hat und auf $x_1, \ldots, x_n$ die Kostenfunktion c verfeinert.

(4) Setze $\mathcal{G}_{\succ_c} := \mathcal{G}_\succ \cap K[x_1, \ldots, x_n]$.

(5) Reduziere x^e modulo $\mathcal{G}_{\succ_c}$ mit dem Algorithmus von Hironaka, etwa zu x^f.

(6) $f \in \mathbb{N}^n$ ist die optimale Lösung von (IP). □

Wegen 3.8 und 1.34 handelt es sich um eine Ausprägung des generischen Algorithmus 2.30, so daß zur Korrektheit nichts zu zeigen bleibt.

3.3 Der Algorithmus von Hoşten und Sturmfels

Der Algorithmus von Hoşten und Sturmfels ([16]) berechnet torische Ideale ohne die Einführung zusätzlicher Variablen. Der Schlüssel dazu liegt in der Ausnutzung der Tatsache, daß nach den Variablen und nicht nach beliebigen Polynomen saturiert wird.

Im ganzen Abschnitt wird vorausgesetzt, daß I_A homogen bezüglich einer (bekannten) positiven Graduierung ist.

Lemma 3.10 *Für $A \in \mathbb{Z}^{m\times n}$ sind äquivalent:*

1. *I_A ist homogen bezüglich einer positiven Graduierung.*
2. *Der Zeilenraum von A enthält einen Vektor, dessen Komponenten alle positiv sind.*

Beweis:

„2. $\Rightarrow$ 1.“ Sei ω ein Vektor mit nur positiven Komponenten im Zeilenraum von A. I_A wird von Binomen der Form $x^{v^+} - x^{v^-}$, $v \in \ker(A)$, erzeugt; wegen

$$v \in \ker(A) \;\Rightarrow\; 0 = \omega^\intercal v = \omega^\intercal (v^+ - v^-) \;\Rightarrow\; \omega^\intercal v^+ = \omega^\intercal v^-$$

sind diese homogen bezüglich der durch ω definierten Graduierung. Nach 1.36 impliziert dies die Homogenität von I_A.

„1. $\Rightarrow$ 2.“ Wir zeigen zunächst, daß I_A keine Monome enthält:
Ist $A \in \mathbb{Z}^{m\times n}$, so gilt $I_A \subseteq I_0$, wobei 0 die $m \times n$-Nullmatrix bezeichnet; denn der Kern von A ist im Kern von 0 enthalten. Es genügt also zu zeigen, daß I_0 keine Monome enthält. Da I_0 nach 2.14 saturiert nach allen Variablen ist, ist dies äquivalent zu $1 \notin I_0$. Nach 2.18 gilt

$$I_0 = \langle\, x_i - 1 \,|\, i = 1\ldots,n \,\rangle.$$

Denn die Binome $x_i - 1$ entsprechen gerade den Einheitsvektoren e_i des $\mathbb{Z}^n$, die eine Gitterbasis von $\ker(0)$ bilden; außerdem macht der Erzeuger $x_i - 1$ die Variable x_i invertierbar modulo dem Ideal J auf der rechten Seite, so daß J bereits saturiert nach x_i ist (Lemma 1.43).

Das Erzeugendensystem $\{x_i - 1 \,|\, i = 1\ldots,n\}$ bildet bezüglich einer beliebigen monomialen Ordnung sogar eine Gröbnerbasis, weil alle Initialterme $\mathrm{in}(x_i - 1) = x_i$ paarweise teilerfremd sind und der Buchberger-Algorithmus daher keine neuen Erzeuger produziert (Satz 1.27). Offensichtlich reduziert 1 modulo dieser Gröbnerbasis nicht zu 0, also folgt $1 \notin I_0$.

Nun sei ω eine positive Graduierung auf I_A. Wäre $x^{v^+} - x^{v^-} \in I_A$ nicht homogen bezüglich ω, so müßten seine homogenen Komponenten, also seine Monome, ebenfalls in I_A liegen. Dies ist aber unmöglich. Somit gilt $\omega^\intercal v = 0$ für alle $v \in \ker(A)$, und ω liegt im Zeilenraum von A. □

Bemerkung 3.11 Um gegebenenfalls einen Vektor mit nur positiven Komponenten im Zeilenraum von A zu bestimmen, muß man das Ungleichungssystem

$$\sum_{i=1}^{m} a^i z_i > 0 \;\Leftrightarrow\; A^\intercal z > 0 \tag{3.3}$$

lösen, wobei a^i die i-te Zeile von A bezeichnet und $z = (z_1, \dots, z_m)^\intercal \in \mathbb{R}^m$ ist. Für eine Lösung z ist $\omega := A^\intercal z$ eine Linearkombination der Zeilen von A mit positiven Komponenten. Das Ungleichungssystem sieht ungünstig aus, da es sich um echte Ungleichungen „$>$" handelt, kann aber durch einen einfachen Trick in ein System mit Ungleichungen „$\geq$" transformiert werden:

Mit einer Lösung z von (3.3) ist auch jedes positive Vielfache λz, $\lambda \geq 0$, eine Lösung von (3.3). Wählt man λ hinreichend groß, so gilt $\lambda z \geq 1$ komponentenweise, so daß man statt (3.3) das System

$$\sum_{i=1}^{m} a^i z_i \geq 1 \;\Leftrightarrow\; A^\intercal z \geq 1 \tag{3.4}$$

lösen kann. Dies kann mit den Techniken aus Kapitel 2 geschehen: Man führt eine (beliebige) Kostenfunktion auf den Variablen z ein, transformiert das lineare Optimierungsproblem, das sich aus (3.4) ergibt, auf Standardform und benutzt den Simplex-Algorithmus zum Auffinden einer Initiallösung. (Warum dieses Vorgehen nicht viel zu kompliziert ist, erklärt [26], Abschnitt 10.3, Theorem 10.4, Stichwort: polynomielle Äquivalenz.) □

Lemma 3.12 *Sei $I \subseteq K[x_1, \dots, x_n]$ ein homogenes Ideal bezüglich einer positiven Graduierung $\omega \in \mathbb{R}^n_{>0}$. $\succ_\omega$ bezeichne die mit ω gewichtete revers-lexikographische Ordnung wie in 1.9 und $\mathcal{G}_{\succ_\omega}$ die reduzierte Gröbnerbasis von I bezüglich $\succ_\omega$. Dann erhält man eine Gröbnerbasis $\hat{\mathcal{G}}_{\succ_\omega}$ von $(I : x_n^\infty)$ bezüglich $\succ_\omega$, indem man jedes Element von $\mathcal{G}_{\succ_\omega}$ durch die höchstmögliche Potenz von x_n teilt. Insbesondere gilt also:*

$$(I : x_n^\infty) = \langle\, g \mid x_n \nmid g,\ \exists\, k \in \mathbb{N} \text{ mit } f = x_n^k \cdot g \in \mathcal{G}_{\succ_\omega} \rangle$$

Beweis: (vgl. Lemma 12.1 in [30]) Wegen $g \in (I : x_n^\infty)$ für alle $g \in \hat{\mathcal{G}}_\succ$ genügt es zu zeigen:

$$\mathrm{in}(I : x_n^\infty) = \langle\, \mathrm{in}(g) \mid g \in \hat{\mathcal{G}}_{\succ_\omega} \rangle$$

„$\supseteq$" ist klar.

„$\subseteq$" Sei $h \in (I : x_n^\infty)$ beliebig. Dann existiert ein $k \in \mathbb{N}$ mit $x_n^k \cdot h \in I$. Nach Definition von „Gröbnerbasis" gibt es ein $f \in \mathcal{G}_{\succ_\omega}$ mit $\mathrm{in}(f) \mid \mathrm{in}(x_n^k \cdot h) = x_n^k \cdot \mathrm{in}(h)$. Da I homogen bezüglich der Graduierung ω ist, ist nach 1.39 auch f homogen bezüglich ω. Nach Wahl von $\succ_\omega$ ist der Initialterm von f ein Term von f, in dem x_n mit minimaler Potenz vorkommt, etwa mit Potenz $l \in \mathbb{N}$. Dann liegt $\frac{f}{x_n^l}$ in $\hat{\mathcal{G}}_{\succ_\omega}$, und aus der Multiplikativität von $\succ_\omega$ folgt:

$$\mathrm{in}\left(\frac{f}{x_n^l}\right) = \frac{\mathrm{in}(f)}{x_n^l} \mid x_n^k \cdot \mathrm{in}(h)$$

Da x_n im Initialterm von $\frac{f}{x_n^l}$ nicht vorkommt, gilt sogar $\mathrm{in}(\frac{f}{x_n^l}) \mid \mathrm{in}(h)$, also $\mathrm{in}(h) \in \langle\, \mathrm{in}(g) \mid g \in \hat{\mathcal{G}}_\succ \rangle$. □

Der folgende Algorithmus erhält als Eingabe eine Matrix A, einen Kostenvektor $c \geq 0$, eine Initiallösung $\mathbb{Z}^n \ni e \geq 0$ von (IP) und zusätzlich eine positive Graduierung $\omega \in \mathbb{R}^n_{>0}$ des Ideals I_A.

Algorithmus 3.13 (Hoşten-Sturmfels-Algorithmus)

(1) Berechne eine Gitterbasis $v_1, \ldots, v_r$ von $\ker(A)$.

(2) Setze $I_0 := \langle\, x^{v_k^+} - x^{v_k^-} \mid k = 1, \ldots, r \,\rangle$.

(3) Für $j = 1, \ldots, n$ berechne $I_j := (I_{j-1} : x_j^\infty)$ mit 3.12.

(4) Berechne die reduzierte Gröbnerbasis $\mathcal{G}_\succ$ von $I_n = I_A$ bezüglich einer Termordnung $\succ$, die die Kostenfunktion c verfeinert.

(5) Reduziere x^e modulo $\mathcal{G}_\succ$ mit dem Algorithmus von Hironaka, etwa zu x^f.

(6) $f \in \mathbb{N}^n$ ist die optimale Lösung von (IP). □

Zeile (3) erledigt man, indem man die Reihenfolge der Variablen vertauscht, also statt der Variablenreihenfolge $x_1, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_{n-1}, x_n$ bei Saturierung nach x_j die Reihenfolge $x_1, \ldots, x_{j-1}, x_n, x_{j+1}, \ldots, x_{n-1}, x_j$ wählt. (Dabei behalten aber sowohl x_n als auch x_j ihr ursprüngliches Gewicht ω_n bzw. ω_j.) Die Behauptung $I_n = I_A$ in Zeile (4) ist gerade die Aussage von Satz 2.18.

3.4 Der Algorithmus von Di Biase und Urbanke

Der Algorithmus von Hoşten und Sturmfels braucht keine Zusatzvariablen, setzt aber eine positive Graduierung des torischen Ideals I_A voraus. Eine solche ist nicht immer gegeben: Wählt man z.B. $A = (1, -1)$, so wird $I_A \subseteq K[x, y]$ erzeugt vom Binom $xy - 1$, das offensichtlich bezüglich keiner positiven Graduierung homogen ist.

Der Algorithmus von Di Biase und Urbanke benötigt eine andere Zusatzvoraussetzung: Er verlangt, daß der Kern von A einen Vektor enthält, dessen Komponenten alle ungleich 0 sind. Das ist auch nicht immer der Fall, wie z.B. die Matrix $A = (1, 0)$ zeigt.

Man findet den Algorithmus von Di Biase und Urbanke in der Originalarbeit [12] sowie in [16] in Form eines Vergleichs mit dem Algorithmus von Hoşten und Sturmfels.

Notation 3.14 Für $\{j_1, \ldots, j_s\} \subseteq \{1, \ldots, n\}$ bezeichne $A_{j_1,\ldots,j_s}$ die Matrix, die durch Multiplikation der Spalten $j_1, \ldots, j_s$ mit -1 aus A hervorgeht. □

Bemerkung 3.15

1. Es gilt $A_{j_1,\ldots,j_s} = (\ldots(A_{j_1})_{j_2}\ldots)_{j_s}$.

2. Ein Vektor $(v_1, \dots, v_{j-1}, v_j, v_{j+1}, \dots, v_n)$ liegt genau dann im Kern von A, wenn der Vektor $(v_1, \dots, v_{j-1}, -v_j, v_{j+1}, \dots, v_n)$ im Kern von A_j liegt. Somit liegt das Binom $x_j^k x^d - x^e$, $x_j \not| x^d$, $x_j \not| x^e$, genau dann in I_A, wenn das Binom $x^d - x_j^k x^e$ in I_{A_j} liegt (vgl. 2.17).

3. Man erhält eine Gitterbasis von A aus einer Gitterbasis von A_j, indem man die j-ten Komponenten der Basisvektoren mit -1 multipliziert. □

Die Idee des Algorithmus von Di Biase und Urbanke beruht auf Lemma 2.19: Wählt man $\{j_1, \dots, j_s\} \subseteq \{1, \dots, n\}$ geschickt, so enthält der Kern der Matrix $A_{j_1,\dots,j_s}$ einen Vektor mit nur positiven Komponenten (an dieser Stelle kommt unsere Voraussetzung zum Tragen!), und man kann das torische Ideal von $A_{j_1,\dots,j_s}$ sofort aus einer Gitterbasis des Kerns ablesen. Das folgende Lemma liefert eine Technik, um aus dem torischen Ideal von $A_{j_1,\dots,j_s}$ sukzessive das torische Ideal von A zu berechnen:

Lemma 3.16 (Variablenflip) Sei $\succ_j$ eine monomiale Ordnung mit der Eliminationseigenschaft bezüglich x_j und

$$\mathcal{G}_j = \{\, x_j^{k_l} x^{d_l} - x^{e_l} \mid x_j \not| x^{d_l},\ x_j \not| x^{e_l},\ l = 1, \dots, t \,\}$$

eine Gröbnerbasis von I_{A_j} bezüglich $\succ_j$. Dann ist

$$\mathcal{G} := \{\, x^{d_l} - x_j^{k_l} x^{e_l} \mid l = 1, \dots, t \,\}$$

ein Erzeugendensystem von I_A.

Beweis: Da nach 3.15 alle Elemente von $\mathcal{G}$ in I_A liegen, ist das von $\mathcal{G}$ erzeugte Ideal in I_A enthalten. Für die umgekehrte Inklusion genügt es zu zeigen, daß ein beliebiges reines Binom $I_A \ni f := x^d - x_j^k x^e$ in dem von $\mathcal{G}$ erzeugten Ideal liegt. Aus $f \in I_A$ folgt $f_j := x_j^k x^d - x^e \in I_{A_j}$. Da $\mathcal{G}_j$ Gröbnerbasis von I_{A_j} ist, liefert der Divisionsalgorithmus von Hironaka eine Darstellung:

$$f_j = x_j^k x^d - x^e = \sum_{l=1}^{t} g_l(x_1, \dots, x_{j-1}, x_j, x_{j+1}, \dots, x_n)(x_j^{k_l} x^{d_l} - x^{e_l}) \tag{3.5}$$

Die Wahl der Termordnung als Eliminationsordnung für x_j und die Vorgehensweise beim Divisionsalgorithmus implizieren, daß x_j in g_l mit Grad $\leq k - k_l$ vorkommt (insbesondere ist $g_l = 0$ für $k < k_l$).
Nun rechnet man formal im Laurentpolynomring $K[x_1, x_1^{-1}, \dots, x_n, x_n^{-1}]$. Ersetzt man in (3.5) x_j durch x_j^{-1}, so ergibt sich die Gleichung:

$$(x_j^{-1})^k x^d - x^e = \sum_{l=1}^{t} g_l(x_1, \dots, x_{j-1}, x_j^{-1}, x_{j+1}, \dots, x_n)((x_j^{-1})^{k_l} x^{d_l} - x^{e_l}) \tag{3.6}$$

Multipliziert man (3.6) mit x_j^k durch, erhält man:

$$\underbrace{x^d - x_j^k x^e}_{=f} = \sum_{l=1}^{t} x^{k-k_l} g_l(x_1, \dots, x_{j-1}, x_j^{-1}, x_{j+1}, \dots, x_n)\, \underbrace{(x^{d_l} - x_j^{k_l} x^{e_l})}_{\in \mathcal{G}}$$

Da x_j in g_l mit Grad $\leq k - k_l$ vorkommt, sind die Produkte

$$x^{k-k_l} g_l(x_1, \ldots, x_{j-1}, x_j^{-1}, x_{j+1}, \ldots, x_n)$$

Polynome in $K[x_1, \ldots, x_n]$. Also liegt f in dem von $\mathcal{G}$ erzeugten Ideal. □

Die Voraussetzung an die Form von $\mathcal{G}_j$ in Lemma 3.16 ist tatsächlich erfüllbar, da die reduzierte Gröbnerbasis eines torischen Ideals nach 2.27 aus reinen Binomen besteht.

Der Algorithmus von Di Biase und Urbanke erhält als Eingabe eine Matrix A, einen Kostenvektor $c \geq 0$ und eine Initiallösung $\mathbb{Z}^n \ni e \geq 0$ von (IP).

Algorithmus 3.17 (Di-Biase-Urbanke-Algorithmus)

(1) Berechne eine Gitterbasis $\hat{v}_1, \ldots, \hat{v}_r$ von $\ker(A)$.

(2) Prüfe die Existenz eines Vektors ohne Nullkomponenten in $\ker(A)$. Existiert kein solcher, beende mit Meldung „Algorithmus ungeeignet zur Lösung der Instanz“.

(3) Sonst transformiere $\hat{v}_1, \ldots, \hat{v}_r$ in eine Gitterbasis $v_1, \ldots, v_r$, die einen Vektor v ohne Nullkomponenten enthält.

(4) Bestimme die negativen Komponenten $j_1, \ldots, j_s$ von v und multipliziere die Komponenten $j_1, \ldots, j_s$ von $v_1, \ldots, v_r$ mit -1.

(5) Setze $I_0 := \langle\, x^{v_k^+} - x^{v_k^-} \mid k = 1, \ldots, r \,\rangle$.

(6) Für $l = 1$ bis s berechne die reduzierte Gröbnerbasis

$$\mathcal{G}_{j_l} = \{\, x_{j_l}^{k_{i_l}} x^{d_{i_l}} - x^{e_{i_l}} \mid x_{j_l} \not| x^{d_{i_l}},\ x_{j_l} \not| x^{e_{i_l}},\ i_l = 1, \ldots, t_l \,\}$$

von I_{l-1} bezüglich einer Eliminationsordnung für x_{j_l} und setze

$$I_l := \langle\, x^{d_{i_l}} - x_{j_l}^{k_{i_l}} x^{e_{i_l}} \mid x_{j_l}^{k_{i_l}} x^{d_{i_l}} - x^{e_{i_l}} \in \mathcal{G}_{j_l} \,\rangle.$$

(7) Berechne die reduzierte Gröbnerbasis $\mathcal{G}_\succ$ von $I_s = I_A$ bezüglich einer Termordnung $\succ$, die die Kostenfunktion c verfeinert.

(8) Reduziere x^e modulo $\mathcal{G}_\succ$ mit dem Algorithmus von Hironaka, etwa zu x^f.

(9) $f \in \mathbb{N}^n$ ist die optimale Lösung von (IP). □

Nach Ausführung von Schritt (4) bilden $v_1, \ldots, v_r$ eine Gitterbasis der Kerns von $A_{j_1,\ldots,j_s}$, die einen Vektor mit nur positiven Komponenten enthält (es wurden gerade alle negativen Komponenten von v mit -1 multipliziert). I_0 aus Schritt (5) ist saturiert nach 2.19. Schritt (6) führt gerade den Variablenflip aus Lemma 3.16 durch; induktiv sieht man leicht, daß dann die in Schritt (7) behauptete Gleichheit $I_s = I_A$ gilt. Daraus folgt die Korrektheit des Algorithmus.

Bei der Frage, wie man die Schritte (2) und (3) realisiert, hilft uns eine einfache Idee: Gegeben sei die Basis $v_1, \ldots, v_r$ eines Gitters $L \subseteq \mathbb{Z}^n$. Definiert man für $\lambda_2, \ldots, \lambda_r \in \mathbb{Z}$

$$v := v_1 + \lambda_2 v_2 + \ldots + \lambda_r v_r,$$

so bilden auch die Vektoren $v, v_2, \ldots, v_r$ eine Gitterbasis von L, denn die Übergangsmatrix

$$\left(\begin{array}{c|c} 1 & 0 \ldots 0 \\ \hline \lambda_2 & \\ \vdots & I_{r-1} \\ \lambda_r & \end{array}\right)$$

ist invertierbar in $\mathbb{Z}^{r \times r}$. Diese Beobachtung liefert einen Algorithmus zum Auffinden einer Gitterbasis wie in Schritt (3); er erhält als Eingabe eine beliebige Gitterbasis $V := \{v_1, \ldots, v_r\}$.

Algorithmus 3.18 (Gitterbasisvektor ohne Nullkomponenten)

(1) Wähle $v \in V$ beliebig. Setze $V := V \backslash \{v\}$.

(2) Sind alle Komponenten von v ungleich Null, so hat die Gitterbasis $v, v_2, \ldots, v_r$ die gewünschte Form.

(3) Sonst wähle $k \in \{1, \ldots, n\}$, so daß die k-te Komponente von v gleich Null ist. Existiert kein Vektor in V, dessen k-te Komponenten ungleich Null ist, beende mit Meldung „Gitter enthält keinen Vektor ohne Nullkomponenten".

(4) Wähle $\hat{v} \in V$, so daß die k-te Komponente von $\hat{v}$ ungleich Null ist. Wähle $\lambda \in \mathbb{Z}$, so daß die Menge der Nullkomponenten von $v + \lambda\hat{v}$ eine echte Teilmenge der Menge der Nullkomponenten von v ist. Ersetze v durch $v + \lambda\hat{v}$ und V durch $V \backslash \{\hat{v}\}$ und gehe zu Schritt (2) zurück. □

Die Korrektheit folgt aus der Tatsache, daß die Anzahl der Nullkomponenten von v im Laufe des Algorithmus immer geringer wird – siehe Schritt (4). Dort probiert man beispielsweise die Werte $1, -1, 2, -2, 3, -3, \ldots$ nacheinander für λ aus. Für jede Komponente k von v, die ungleich Null ist, existiert höchstens ein λ, so daß die Komponente k von $v + \lambda\hat{v}$ gleich Null ist. Also findet man nach spätestens q Versuchen ein geeignetes λ, wobei q die Anzahl der Nichtnullkomponenten von v ist.

Die Schritte (2) bis (4) werden höchstens r-mal ausgeführt, da jeder Vektor, der als Summand in die Linearkombination v eingeht, aus V gelöscht wird. Schritt (2) testet die n Komponenten von v, Schritt (3) betrachtet zusätzlich die k-ten Komponenten der $\leq r-1$ Vektoren in V. In Schritt (4) müssen $\leq n-1$ Werte für λ getestet und für jeden Wert die Komponenten von $v + \lambda\hat{v}$ betrachtet werden. Somit ist die Gesamtlaufzeit des Algorithmus durch $\mathcal{O}(r \cdot n^2)$ beschränkt (beachte $r \leq n$).

Das in Schritt (4) bestimmte λ kann betragsmäßig durch $\frac{n}{2}$ beschränkt werden. Ist M das Maximum der Beträge der Koeffizienten von $v_1, \ldots, v_r$, so wächst der Betrag jedes Koeffizienten von v bei Ausführung von (4) um einen Summanden $\leq \frac{n}{2} \cdot M$. Also haben am Ende des Algorithmus alle Koeffizienten von v einen Betrag $\leq M + (r-1) \cdot \frac{n}{2} \cdot M$.

Dieses Koeffizientenwachstum ist unerwünscht, da es den Grad der Idealerzeuger erhöht und somit die Laufzeit des Buchberger-Algorithmus ungünstig beeinflußt (vgl. Abschnitt 1.5). Allerdings geht die Abschätzung vom seltenen worst case aus, daß alle Vektoren aus V an der Konstruktion von v beteiligt sind und kleine λ in Schritt (4) stets eine Nichtnullkomponente von v annullieren. Zudem kann man den Algorithmus durch Heuristiken ergänzen, zum Beispiel, indem als Startvektor v immer ein Vektor mit wenig Nullkomponenten ausgewählt wird.

3.5 Der Algorithmus von Bigatti, La Scala und Robbiano

Die Arbeit [3] stellt zwei Versionen des Buchberger-Algorithmus zur Berechnung torischer Ideale vor. Der Schwerpunkt liegt dabei auf praktischen Aspekten. Wir gehen zunächst auf die theoretischen Ergebnisse ein, die eine weitere Möglichkeit zur Berechnung der Saturierung liefern. Die praktischen Ideen werden in den folgenden Kapiteln behandelt.

Der in [3] als EATI, **E**liminations-**A**lgorithmus für **t**orische **I**deale, bezeichnete Algorithmus ist gewissermaßen die homogene Variante des Algorithmus von Pottier. Wie der Algorithmus von Hoşten und Sturmfels geht er davon aus, daß I_A homogen bezüglich einer positiven Graduierung ω ist.

Sei $v_1, \ldots, v_r$ eine Gitterbasis von $\ker(A)$. Wir betrachten die erweiterte Matrix

$$\tilde{A} := \left(\begin{array}{ccc|c} a_{1,1} & \ldots & a_{1,n} & \sum_{j=1}^n a_{1,j} \\ \vdots & \ddots & \vdots & \vdots \\ a_{m,1} & \ldots & a_{m,n} & \sum_{j=1}^n a_{m,j} \end{array} \right).$$

Der Kern von $\tilde{A}$ hat Dimension $\dim(\ker(A)) + 1 = r + 1$, und die Vektoren $\binom{v_i}{0}$, $i = 1, \ldots, r$, und $(1, \ldots, 1, -1)^\intercal \in \mathbb{Z}^{n+1}$ bilden nach Konstruktion eine Gitterbasis dieses Kerns. Man beachte die Ähnlichkeit mit der erweiterten Matrix im Algorithmus von Pottier: Die zusätzliche Spalte unterscheidet sich nur durch das Vorzeichen. (Sie wird hinten statt vorne angefügt, um Verwirrungen bezüglich der Variablenreihenfolge vorzubeugen.)

Ist nun $I_A \subseteq K[x_1, \ldots, x_n]$ homogen bezüglich $\omega \in \mathbb{R}^n_{>0}$, so ist $I_{\tilde{A}} \subseteq K[x_1, \ldots, x_n, u]$ homogen bezüglich

$$\tilde{\omega} := (\omega_1, \ldots, \omega_n, \sum_{i=1}^n \omega_i) \in \mathbb{R}^{n+1}_{>0}.$$

Das folgende Lemma liefert eine Möglichkeit, ein Erzeugendensytem von I_A aus einem Erzeugendensystem von $I_{\tilde{A}}$ zu berechnen, wobei alle Berechnungen mit homogenen Polynomen stattfinden. Erfahrungsgemäß wirkt sich dies in den meisten Fällen günstig auf die Laufzeit des Buchberger-Algorithmus aus.

Lemma 3.19 *Sei $I \subseteq K[x_1, \ldots, x_n]$ ein homogenes Ideal bezüglich einer positiven Graduierung ω und $0 \neq g \in K[x_1, \ldots, x_n]$ ein homogenes Polynom vom Gewicht d bezüglich ω. Dann gilt:*

1. *Das Ideal $J \subseteq K[x_1, \ldots, x_n, u]$, das von den Elementen von I und dem Binom $g - u \in K[x_1, \ldots, x_n, u]$ erzeugt wird, ist homogen bezüglich der positiven Graduierung*
$$\tilde{\omega} := (\omega_1, \ldots, \omega_n, d).$$
2. *Es seien $\succ_{\tilde{\omega}}$ die mit $\tilde{\omega}$ gewichtete revers-lexikographische Ordnung auf $K[x_1, \ldots, x_n, u]$ und*
$$\{u^{k_j} f_j \mid u \not| f_j,\ j = 1, \ldots, t\}$$
die reduzierte Gröbnerbasis von J bezüglich $\succ_{\tilde{\omega}}$. Dann gilt
$$(I : g^\infty) = \langle \hat{f}_j \mid j = 1, \ldots, t \rangle,$$
wobei $\hat{f}_j$ aus f_j hervorgeht, indem u durch g substituiert wird:
$$\hat{f}_j(x_1, \ldots, x_n) := f_j(x_1, \ldots, x_n, g) \in K[x_1, \ldots, x_n]$$

Beweis: Die erste Aussage ist klar nach 1.36: Betrachtet man ein bezüglich ω homogenes Erzeugendensystem von I, so sind dessen Elemente, als Polynome in $K[x_1, \ldots, x_n, u]$ aufgefaßt, auch homogen bezüglich $\tilde{\omega}$. Die zweite Aussage beweisen wir in drei Schritten:

1. Es gilt $I = J \cap K[x_1, \ldots, x_n]$:
Für eine Eliminationsordnung $\succ_u$ bezüglich u gilt $\mathrm{in}_{\succ_u}(g - u) = -u$, und in den anderen Erzeugern von J aus Teil 1 des Satzes kommt u nicht vor. Nach 1.27 reduzieren alle S-Paare von $g - u$ mit anderen Erzeugern zu Null. Die reduzierte Gröbnerbasis von J bezüglich $\succ_u$ enthält somit außer $u - g$ nur Elemente, in denen u nicht vorkommt. Aus 1.34 folgt die Behauptung.
2. Es gilt $I \subseteq \langle \hat{f}_j \mid j = 1, \ldots, t \rangle \subseteq (I : g^\infty)$:
Ist $f \in I \subseteq J$, so existiert eine Darstellung
$$f = \sum_{j=1}^{t} g_j(x_1, \ldots, x_n, u) \cdot u^{k_j} \cdot f_j.$$
Setzt man auf beiden Seiten g für u ein, ergibt sich (da u in f nicht vorkommt)
$$f = \sum_{j=1}^{t} g_j(x_1, \ldots, x_n, g) \cdot g^{k_j} \cdot \hat{f}_j,$$
also $f \in \langle \hat{f}_j \mid j = 1, \ldots, t \rangle$. Daraus folgt die erste Inklusion.
Wegen $u^{k_j} f_j \in J$ und $u \equiv g \bmod J$ erhält man durch Ersetzung von u durch g
$$g^{k_j} \hat{f}_j \in J \cap K[x_1, \ldots, x_n] \overset{1.}{=} I,$$
also $\hat{f}_j \in (I : g^\infty)$ für $j = 1, \ldots, t$. Daraus ergibt sich die zweite Inklusion.

3. Wegen 2. genügt es zu zeigen, daß $\langle \hat{f}_j \mid j = 1, \dots, t \rangle$ nach g saturiert ist. Sei dazu $g^k f \in \langle \hat{f}_j \mid j = 1, \dots, t \rangle \subseteq K[x_1, \dots, x_n]$. Zu zeigen ist $f \in \langle \hat{f}_j \mid j = 1, \dots, t \rangle$.
   Mit $f_j \in (J : u^\infty)$ liegt wegen $u \equiv g \bmod J$ auch $\hat{f}_j$ in $(J : u^\infty)$, also
   $$g^k f \in (J : u^\infty) \overset{g \equiv u \bmod J}{\Rightarrow} u^k f \in (J : u^\infty)$$
   $$\Rightarrow \ f \in (J : u^\infty) \overset{\text{Lemma 3.12}}{=} \langle f_j \mid j = 1, \dots, t \rangle$$
   Somit existiert eine Darstellung
   $$f = \sum_{j=1}^{t} g_j(x_1, \dots, x_n, u) \cdot f_j.$$
   Setzt man auf beiden Seiten g für u ein, ergibt sich (da u in f nicht vorkommt)
   $$f = \sum_{j=1}^{t} g_j(x_1, \dots, x_n, g) \cdot \hat{f}_j,$$
   also $f \in \langle \hat{f}_j \mid j = 1, \dots, t \rangle$. □

Daraus ergibt sich folgender Algorithmus zur Lösung von (IP), der als Eingabe den nichtnegativen Kostenvektor c, die Matrix A, eine Initiallösung $\mathbb{Z}^n \ni e \geq 0$ von (IP) und zusätzlich eine positive Graduierung $\omega \in \mathbb{R}^n_{>0}$ des Ideals I_A erhält:

Algorithmus 3.20 (Algorithmus von Bigatti, La Scala und Robbiano)

(1) Berechne eine Gitterbasis $v_1, \dots, v_r$ von $\ker(A)$.

(2) Setze $J := \langle \{ x^{v_k^+} - x^{v_k^-} \mid k = 1, \dots, r \} \cup \{x_1 \cdot \ldots \cdot x_n - u\} \rangle$.

(3) Setze $d := \sum_{j=1}^{n} \omega_i$ und $\tilde{\omega} := (\omega_1, \dots, \omega_n, d)$.

(4) Berechne die reduzierte Gröbnerbasis $\mathcal{G}$ von J bezüglich der mit $\tilde{\omega}$ gewichteten revers-lexikographischen Ordnung auf $K[x_1, \dots, x_n, u]$.

(5) Schreibe jedes Element von $\mathcal{G}$ in der Form $u^k f(x_1, \dots, x_n, u)$, $u \not| f(x_1, \dots, x_n, u)$; ersetze es dann durch $\hat{f} := f(x_1, \dots, x_n, x_1 \cdot \ldots \cdot x_n) \in K[x_1, \dots, x_n]$.

(6) Berechne die reduzierte Gröbnerbasis $\mathcal{G}_\succ$ des von den $\hat{f}$ aus (5) erzeugten Ideals bezüglich einer Termordnung $\succ$, die die Kostenfunktion c verfeinert.

(7) Reduziere x^e modulo $\mathcal{G}_\succ$ mit dem Algorithmus von Hironaka, etwa zu x^f.

(8) $f \in \mathbb{N}^n$ ist die optimale Lösung von (IP). □

Nach Ausführung von Schritt (5) bilden die $\hat{f}$ ein Erzeugendensystem des Ideals I_A; dies folgt aus 2.18 und Lemma 3.19, indem man g als das Produkt $x_1 \cdot \ldots \cdot x_n$ der Variablen wählt. Daraus folgt dann die Korrektheit.

Man beachte, daß die Zusatzvariable u zwar aus dem Erzeugendensystem $\mathcal{G}$ „hinausgeworfen“ wird, es sich aber nicht um eine Elimination im Sinne von Abschnitt 1.6 handelt. Da die betrachtete monomiale Ordnung auf $K[x_1, \ldots, x_n]$ graduiert ist, hat sie auch nicht die Eliminationseigenschaft bezüglich u. Das Ersetzen von u durch das Produkt der ursprünglichen Variablen bezeichnen wir als **Pseudoelimination**.

3.6 Das Verfahren von Hoşten und Shapiro

Die vorangehenden Algorithmen unterscheiden sich vor allem in der Art und Weise, auf die sie die Saturierung ihres Startideals nach allen Variablen berechnen. Das Verfahren von Hoşten und Shapiro liefert keinen neuen Algorithmus, aber einen Ansatz, mit dem sich die Effizienz vieler vorgestellter Algorithmen verbessern läßt. Konkret erkennt es Variablen, nach denen man nicht zu saturieren braucht.

Lemma 3.21 *Sei $I \subseteq K[x_1, \ldots, x_n]$ ein Ideal und $\mathcal{G}$ ein endliches Erzeugendensystem von I aus reinen Binomen. Dann existiert eine Teilmenge $E \subseteq \{1, \ldots, n\}$ mit $\leq \frac{n}{2}$ Elementen, so daß gilt: Sind die Variablen x_e, $e \in E$, invertierbar modulo I, so sind alle Variablen, die in $\mathcal{G}$ vorkommen, invertierbar modulo I.*

Beweis: Man beginnt mit einem beliebigen Element von $\mathcal{G}$, etwa mit $x^{v^+} - x^{v^-}$. O.E. sei x^{v^+} dasjenige der beiden Monome, das die wenigstens Variablen involviert (bei gleicher Anzahl darf man sich das Monom aussuchen). E bezeichne die Menge $\{e \in \{1, \ldots, n\} \,|\, x_e \text{ kommt in } x^{v^+} \text{ vor}\}$. Existieren nun die Inversen $h_e \in K[x_1, \ldots, x_n]$ von x_e, $e \in E$, modulo I, so ist h^{v^+} (analog zu x^{v^+} definiert) das Inverse zu $x^{v^+} \equiv x^{v^-}$ modulo I. Bezeichnet $\overline{E} := \{e \in \{1, \ldots, n\} \,|\, x_e \text{ kommt in } x^{v^-} \text{ vor}\}$, so gilt für $e \in \overline{E}$:

$$x_e \cdot (x_e^{-v_e - 1} \cdot (\prod_{j \in \overline{E},\, j \neq e} x_j^{-v_j}) \cdot h^{v^+}) = x^{v^-} \cdot h^{v^+} \equiv x^{v^+} \cdot h^{v^+} \equiv 1 \bmod I$$

Somit ist x_e invertierbar modulo I; die Invertierbarkeit der Variablen mit Index aus E impliziert also die Invertierbarkeit der Variablen mit Index aus $\overline{E}$.

Umfaßt nun $E \cup \overline{E}$ alle Variablen, die in $\mathcal{G}$ vorkommen, so ist man fertig. Sonst wählt man ein weiteres Binom aus $\mathcal{G}$, das mindestens eine *Restvariable* mit Index aus $\{1, \ldots, n\} \backslash (E \cup \overline{E})$ involviert, etwa $x^{w^+} - x^{w^-}$. O.E. sei x^{w^+} dasjenige der beiden Monome, in dem die wenigstens Restvariablen vorkommen. Diese nimmt man zu E hinzu, die Restvariablen in x^{w^-} zu $\overline{E}$. Diese Prozedur setzt man fort, bis alle Variablen, die in $\mathcal{G}$ vorkommen, auf E und $\overline{E}$ verteilt sind. Wegen $E \cap \overline{E} = \emptyset$ und $|E| \leq |\overline{E}|$ nach Konstruktion hat die Menge E die im Lemma geforderten Eigenschaften. □

Beipiel 3.22 Sei $I = \langle x_1 g - 1, x_1 - x_2 x_3, x_2 x_4 - x_1 x_3 \rangle \subseteq K[x_1, \ldots, x_4]$, wobei $0 \neq g \in K[x_1, \ldots, x_4]$ beliebig ist. Dann ist x_1 invertierbar modulo I. Wegen dem zweiten Erzeuger sind auch x_2 und x_3 invertierbar modulo I, wegen dem dritten dann x_4. □

Der Beweis von Lemma 3.21 ist konstruktiv und ergibt das **Verfahren von Hoşten und Shapiro.**

Dieses liefert unmittelbar kleine Verbesserungen für die Algorithmen von Pottier bzw. Bigatti, La Scala und Robbiano, die die Saturiertheit ihres Startideals durch Inversion aller Variablen erreichen. Nach dem Lemma kann man nämlich das Produkt $x_1 \cdot \ldots \cdot x_n$ in den zusätzlichen Erzeugern $t \cdot x_1 \cdot \ldots \cdot x_n - 1$ bzw. $x_1 \cdot \ldots \cdot x_n - u$ durch das Produkt $\prod_{e \in E} x_e$ ersetzen – um Variablen, die im Erzeugendensystem nicht vorkommen, braucht man sich übrigens nicht zu kümmern, da das Ideal nach ihnen bereits saturiert ist. Durch die Ersetzung erhält man Binome mit kleinerem Initialterm, wodurch man sich eine leichte Effizienzverbesserung beim Buchberger-Algorithmus erhoffen darf. Im Algorithmus von Bigatti, La Scala und Robbiano muß der Grad von u dabei entsprechend reduziert werden. Das Verfahren von Hoşten und Shapiro selbst läßt sich übrigens sehr effizient implementieren (siehe Anhang A.2.9) und verursacht dann nur geringen Zusatzaufwand.

Weit interessanter ist das Ergebnis für den Algorithmus von Hoşten und Sturmfels. Nach 1.43 folgt aus der Invertierbarkeit einer Variablen modulo eines Ideals schon die Saturiertheit des Ideals nach dieser Variablen. Wäre diese Aussage umkehrbar, so würde das Verfahren von Hoşten und Shapiro die Anzahl der sukzessiven Saturierungen um mindestens die Hälfte reduzieren, so daß man wie beim Algorithmus von Di Biase und Urbanke mit $\leq \frac{n}{2} + 1$ Gröbnerbasisberechnungen auskäme.

Im allgemeinen gilt die Umkehrung nicht: So ist z.B. das Ideal $\langle x_2 - x_1 \rangle \subseteq K[x_1, x_2]$ saturiert nach x_1, aber wegen $g \cdot x_1 - 1 \notin \langle x_2 - x_1 \rangle$ für alle $g \in K[x_1, x_2]$ ist x_1 nicht invertierbar modulo $\langle x_2 - x_1 \rangle$. Die gewünschte Verbesserung des Algorithmus von Hoşten und Sturmfels kann man dennoch erreichen:

Satz 3.23 *Seien $A \in \mathbb{Z}^{m \times n}$, $v_1, \ldots, v_r$ eine Gitterbasis von $\ker(A) \subseteq \mathbb{Z}^n$ und*

$$J := \langle\, x^{v_k^+} - x^{v_k^-} \mid k = 1, \ldots r \rangle.$$

Sei E wie im Beweis von Lemma 3.13 aus den angegebenen Erzeugern von J konstruiert. Dann gilt:

$$I_A = J : \left(\prod_{e \in E} x_e \right)^{\infty}$$

Beweis: Da die Herleitung der benötigten Theorie sehr aufwendig wäre, sei auf die Literatur verwiesen, und zwar auf Theorem 4.1 in [15] oder Theorem 5.3 in [17]. Der Schluß von der Aussage des Lemmas 3.21 auf die Aussage des Satzes erfordert Kenntnisse über die **Primärzerlegung** torischer Ideale, die in [15] hergeleitet und in [17] kurz erwähnt werden. Alternativ findet man in [3] einen (nicht sehr detaillierten) Beweis mit Hilfe von **Lokalisierungen.** □

In [15] wird ferner bewiesen, daß stets $m + 1$ geeignet gewählte Variablen ausreichen, um das Ideal J aus 3.23 zu saturieren. Dies geschieht über die Hermite-Normalform von A (vgl. Abschnitt 1.10) und bedeutet im Fall $m + 1 < \frac{n}{2}$ eine weitere Verbesserung gegenüber Satz 3.23.

3.7 Ergänzungen

3.7.1 Ein Algorithmenvergleich a priori

Die bisherige Auswahl an Algorithmen wirft natürlich die Frage auf, welche davon für praktische Zwecke am besten geeignet sind. Wir stellen nun vergleichend einige Eigenschaften der Algorithmen zusammen, die bei der Interpretation der Testdaten in Kapitel 6 helfen.

Der **Algorithmus von Conti und Traverso** braucht viele Hilfsvariablen, nämlich m bzw. $m+1$ Stück, wobei m die Anzahl der Zeilen der Matrix A ist. Da die Laufzeit des Buchberger-Algorithmus in der Regel mit der Variablenzahl stark ansteigt, läßt dies ein eher ungünstiges Laufzeitverhalten erwarten.

Die anderen Algorithmen wurden nicht zuletzt entwickelt, um die hohe Variablenzahl des Conti-Traverso-Algorithmus zu reduzieren. Allerdings ist dabei etwas verlorengegangen, das in der Praxis den Conti-Traverso-Algorithmus vielleicht zur einzigen Option macht: Dieser benötigt im Gegensatz zu den anderen Algorithmen keine Initiallösung des Gleichungssystems $AX = b$, sondern kann sich diese aus der rechten Seite b konstruieren.

Den Conti-Traverso-Algorithmus mit Elimination sollte man in dieser Hinsicht als rein theoretisches Konstrukt einordnen; denn er besitzt die Fähigkeit zum Auffinden einer Initiallösung nicht mehr, aber noch alle Nachteile. Weil er leicht zu implementieren ist, findet er dennoch sein praktisches Einsatzgebiet, nämlich bei Korrektheitstests von Implementierungen komplizierterer Algorithmen.

Bei der nichtnegativen Variante des Conti-Traverso-Algorithmus fällt die einfache Struktur der Erzeuger auf, die der Buchberger-Algorithmus vielleicht zu seinen Gunsten nutzen kann. Allerdings rechnen alle Varianten mit Eliminationsordnungen, die in der Regel nicht besonders günstig für den Buchberger-Algorithmus sind.

Der **Algorithmus von Pottier** benötigt nur eine einzige Hilfsvariable. Er rechnet immer noch mit Eliminationsordnungen. Die Gitterbasisberechnung bedeutet im Vergleich mit dem Buchberger-Algorithmus nur einen geringen zusätzlichen Aufwand (außer beim Programmieren), der durch die Einsparung von Variablen leicht kompensiert wird.

Dem **Algorithmus von Hoşten und Sturmfels** gelingt es, ganz auf Hilfsvariablen zu verzichten. Die Eliminationsordnungen machen den günstigeren gewichteten Ordnungen Platz, die Berechnungen finden sogar vorwiegend mit homogenen Polynomen statt. Bezahlt werden diese Errungenschaften dadurch, daß statt einer gleich mehrere Gröbnerbasen berechnet werden müssen ($n+1$ in der naiven, bis zu $\frac{n}{2}+1$ in der mit Hoşten-Shapiro „getunten“ Version). Man darf aber erwarten, daß diese Gröbnerbasen durch die Abwesenheit von Hilfsvariablen deutlich kleiner ausfallen als zuvor und sich zudem während der Phase der Saturierung nur wenig voneinander unterscheiden, da die monomialen Ordnungen ähnlich sind.

Ein nicht zu behebender Nachteil liegt in der Tatsache, daß der Algorithmus nicht mehr universell einsetzbar ist, denn die Korrektheit der Berechnungen gewährleisten nur homogene Ideale (vgl. 3.10).

Der **Algorithmus von Di Biase und Urbanke** tauscht ebenfalls die Hilfsvariablen gegen eine größere Anzahl von Gröbnerbasisberechnungen ein, höchstens $\frac{n}{2}+1$ Stück. Die Rechnungen sind nicht homogen, benutzen sogar Eliminationsordnungen. Außerdem macht die Zusatzvoraussetzung an den Kern den Algorithmus für manche Matrizen unbrauchbar.

Der **Algorithmus von Bigatti, La Scala und Robbiano** benötigt eine Hilfsvariable und zwei Gröbnerbasisberechnungen. Er rechnet mit homogenen Idealen, was – wie bereits erwähnt – vorteilhaft für den Buchberger-Algorithmus und nachteilig für die Einsetzbarkeit ist.

3.7.2 Weitere Ideen

Obwohl die Algorithmen aus den Abschnitten 1 bis 5 dieses Kapitels in der Originalliteratur aus recht unterschiedlichen Perspektiven betrachtet werden, entdeckt man bei genauerer Betrachtung viele Ähnlichkeiten. Der Algorithmus von Pottier benutzt dieselbe Methode zur Berechnung der Saturierung wie der Conti-Traverso-Algorithmus für den allgemeinen Fall. Der Algorithmus von Bigatti, La Scala und Robbiano kombiniert eben diese Methode mit der Idee der Homogenisierung.

Sind die Algorithmen einmal in einer gemeinsamen Sprache formuliert, kann man sie leicht zu neuen Ansätzen kombinieren, die sich für bestimmte Problemklassen besonders eignen. Möchte man zum Beispiel beim Conti-Traverso-Algorithmus beliebige Matrizen, aber nur nichtnegative Begrenzungsvektoren zulassen, so braucht man die Hilfsvariable t bei der Reduktion der künstlichen Initiallösungen, die aus den Begrenzungsvektoren gebaut werden, nicht. Somit kann man – sofern ein positiver Vektor im Zeilenraum der Matrix $\tilde{A} := (I_m|A)$ existiert – die Saturierung von $\tilde{A}$ mit der homogenen Methode von Bigatti, La Scala und Robbiano berechnen: Man ersetzt den Erzeuger $t \cdot y_1 \cdot \ldots \cdot y_m - 1$ durch $y_1 \cdot \ldots \cdot y_m - u$, wobei der Grad von u passend gewählt wird.

Unbefriedigend ist die riesige Zahl von Hilfsvariablen in dem Fall, daß keine Initiallösung gegeben ist und nur der Conti-Traverso-Algorithmus als Option bleibt. Möchte man jedoch ein IP-Problem nur für *eine* rechte Seite b lösen und nicht für mehrere, kommt man mit einer Hilfsvariablen aus: Statt dem Gleichungssystem $AX = b$ betrachtet man die um eine Spalte erweiterte Matrix

$$\tilde{A} := (\ b \quad | \quad A\ ) \in \mathbb{Z}^{m\times(1+n)}$$

und das Gleichungssystem $\tilde{A}\binom{T}{X} = b$. Dann ist $\binom{T_0}{X_0} = \binom{1}{0} \in \mathbb{Z}^{1+n}$ eine nichtnegative Initiallösung dieses Systems. Man berechnet nun das torische Ideal von $\tilde{A}$, genauer eine Gröbnerbasis bezüglich einer monomialen Ordnung, die t eliminiert und die Kostenfunktion auf x verfeinert. Durch diese wird dann die Initiallösung $\binom{T_0}{X_0}$ reduziert. Ist die erste Komponente des Ergebnisses ungleich Null, so ist das Gleichungssystem $AX = b$ und damit das IP-Problem unlösbar; ansonsten liefern die X-Komponenten wie üblich die optimale Lösung.

3.7.3 Ein kleines Beispiel

Wir demonstrieren den Ablauf der verschiedenen Algorithmen an einem kleinen Beispiel. Dazu betrachten wir wie in 2.20 die Matrix

$$A = \begin{pmatrix} 5 & 1 & 2 & 0 \\ 2 & 1 & 1 & 1 \end{pmatrix} \in \mathbb{Z}^{2\times 4}.$$

Nach 2.20 bilden die Vektoren $(1,-1,-2,1)^\intercal$ und $(0,2,-1,-1)^\intercal$ eine Gitterbasis von $\ker(A)$. Wir lösen nun mit den verschiedenen Algorithmen das ganzzahlige Optimierungsproblem

$$\min\left\{(1,1,1,1)\cdot X \;\middle|\; \begin{pmatrix} 5 & 1 & 2 & 0 \\ 2 & 1 & 1 & 1 \end{pmatrix}\cdot X = \begin{pmatrix} 4 \\ 4 \end{pmatrix},\ X \geq 0,\ X \in \mathbb{Z}^4\right\} \quad (3.7)$$

mit Kostenvektor $c = (1,1,1,1)^\intercal$ und rechter Seite $b = (4,4)^\intercal$. Eine Initiallösung dafür ist z.B. $X_{ini} = (0,2,1,1)^\intercal$.

Beipiel 3.24 (Conti-Traverso) Da alle Koeffizienten von A und b nichtnegativ sind, können wir die Version aus Abschnitt 3.1.1 verwenden. Neben den gewichteten Variablen x_1, x_2, x_3, x_4 benötigt diese zwei Eliminationsvariablen y_1, y_2. Man startet mit dem Ideal

$$I := \langle\, y_1^5 y_2^2 - x_1,\ y_1 y_2 - x_2,\ y_1^2 y_2 - x_3,\ y_2 - x_4\rangle.$$

Als monomiale Ordnung kann man z.B. die Blockordnung wählen, die gegeben ist durch die lexikographische Ordnung auf y_1, y_2, verfeinert durch die graduierte lexikographische Ordnung auf x_1, x_2, x_3, x_4. Letztere stimmt gerade mit der mit c gewichteten lexikographischen Ordnung überein. Die reduzierte Gröbnerbasis $\mathcal{G}$ von I bezüglich dieser Ordnung ist

$$\mathcal{G} = \{x_2^2 - x_3x_4,\ x_3^3 - x_1x_2,\ x_2x_3^2 - x_1x_4,\ y_2 - x_4,\ y_1x_4 - x_2,\ y_1x_2 - x_3,\ y_1x_3^2 - x_1\};$$

insbesondere ist nach 1.34 und 3.4 die Menge

$$\{x_2^2 - x_3x_4,\ x_3^3 - x_1x_2,\ x_2x_3^2 - x_1x_4\}$$

die reduzierte Gröbnerbasis von I_A bezüglich der graduierten lexikographischen Ordnung.

Die rechte Seite b entspricht dem Monom $y_1^4 y_2^4$. Reduziert man dieses durch $\mathcal{G}$, so erhält man das Monom $x_3^2 x_4^2$, das der optimalen Lösung $X_{opt} = (0,0,2,2)^\intercal$ von (3.7) entspricht. □

Beipiel 3.25 (Pottier) Unter Benutzung der obigen Gitterbasis und einer Eliminationsvariablen t erhält man das Anfangsideal

$$J := \langle\, t x_1x_2x_3x_4 - 1,\ x_1x_4 - x_2x_3^2,\ x_2^2 - x_3x_4\rangle.$$

Als monomiale Ordnung wählen wir wiederum eine Blockordnung mit Variablenblöcken t und x_1, x_2, x_3, x_4. Auf dem t-Block sind alle Termordnungen identisch, auf x_1, x_2, x_3, x_4 betrachten wir wieder die graduierte lexikographische Ordnung. Die reduzierte Gröbnerbasis von J bezüglich dieser Ordnung ist

$$\{x_2^2 - x_3x_4,\ x_3^3 - x_1x_2,\ x_2x_3^2 - x_1x_4,\ tx_1x_2x_3x_4 - 1,\ tx_1^2x_4^2 - x_3,\ tx_1x_3^2x_4^2 - x_2\};$$

wie im vorigen Beispiel ergibt sich nach 1.34 und 3.8

$$\mathcal{G} = \{x_2^2 - x_3x_4,\, x_3^3 - x_1x_2,\, x_2x_3^2 - x_1x_4\}$$

als die reduzierte Gröbnerbasis von I_A bezüglich der graduierten lexikographischen Ordnung.
Statt von der rechten Seite b müssen wir die Reduktion von einer zugehörigen Initiallösung aus starten, z.B. $X_{ini} = (0,2,1,1)^\intercal$. Diese entspricht dem Monom $x_2^2x_3x_4$. Reduktion dieses Monoms durch $\mathcal{G}$ ergibt das Monom $x_3^2x_4^2$, entsprechend der optimalen Lösung $X_{opt} = (0,0,2,2)^\intercal$ von (3.7). Bemerkenswert ist, daß die Kosten der Initiallösung und der Optimallösung bezüglich c übereinstimmen: $c^\intercal X_{ini} = 4 = c^\intercal X_{opt}$. Die IP-Instanz (3.7) hat also keine eindeutige optimale Lösung. Daß X_{ini} zu X_{opt} reduziert wird, liegt daran, daß das Gewicht c durch die lexikographische Ordnung verfeinert wird und X_{opt} lexikographisch kleiner ist als X_{ini}. □

Beipiel 3.26 (Hoşten-Sturmfels) Das Anfangsideal J_0 ergibt sich unmittelbar aus der Gitterbasis:

$$J_0 := \langle\, x_1x_4 - x_2x_3^2,\, x_2^2 - x_3x_4 \rangle$$

Wir demonstrieren den Ablauf sowohl des Originalalgorithmus als auch der durch das Hoşten-Shapiro-Verfahren verbesserten Version. Beide brauchen zunächst eine positive Graduierung von I_A, die z.B. durch den zweiten Zeilenvektor $(2,1,1,1)$ von A gegeben ist. Zur Saturierung benutzen wir gewichtete revers-lexikographische Ordnungen, wobei x_1 das Gewicht 2, x_2, x_3, x_4 jeweils Gewicht 1 erhalten. Die Reihenfolge, in der die Variablen betrachet werden, variiert im Laufe des Algorithmus.

Der Originalalgorithmus saturiert nacheinander nach jeder Variablen. Startet man den Buchberger-Algorithmus mit der mit $(1,1,1,2)$ gewichteten reverslexikographischen Ordnung bezüglich der Variablenreihenfolge x_4, x_2, x_3, x_1, so erhält man als reduzierte Gröbnerbasis von J_0 die Menge

$$\mathcal{G}_0 = \{x_2^2 - x_4x_3,\, x_2x_3^2 - x_4x_1,\, x_4x_3^3 - x_4x_2x_1\}.$$

Nach Satz 3.12 muß man alle Elemente von $\mathcal{G}_0$ durch die höchstmögliche Potenz von x_1 teilen, um eine Gröbnerbasis von $J_1 := (J_0 : x_1^\infty)$ bezüglich derselben Ordnung zu erhalten. Da kein Element durch x_1 teilbar ist, ist $\mathcal{G}_0$ bereits Gröbnerbasis von J_1 und insbesondere J_0 schon nach x_1 saturiert.
Um nach x_2 zu saturieren, berechnet man die reduzierte Gröbnerbasis $\mathcal{G}_1$ von J_1 bezüglich der mit $(2,1,1,1)$ gewichteten revers-lexikographischen Ordnung für die Variablenreihenfolge x_1, x_4, x_3, x_2:

$$\mathcal{G}_1 = \{x_4x_3 - x_2^2,\, x_1x_4 - x_3^2x_2,\, x_3^3x_2 - x_1x_2^2\}.$$

Den letzten Erzeuger kann man tatsächlich durch x_2 teilen; somit ist

$$\{x_4x_3 - x_2^2,\, x_1x_4 - x_3^2x_2,\, x_3^3 - x_1x_2\}$$

Gröbnerbasis von $J_2 := (J_1 : x_2^\infty)$.
Die reduzierte Gröbnerbasis $\mathcal{G}_2$ von J_2 bezüglich der mit $(2,1,1,1)$ gewichteten revers-lexikographischen Ordnung für die Variablenreihenfolge x_1, x_2, x_4, x_3 ist

$$\mathcal{G}_2 = \{x_2^2 - x_4x_3,\, x_1x_4 - x_2x_3^2x,\, x_1x_2 - x_3^3\}.$$

Da kein Erzeuger durch x_3 teilbar ist, folgt $J_3 := (J_2 : x_3^\infty) = J_2$.
Schließlich ergibt sich die reduzierte Gröbnerbasis $\mathcal{G}_3$ von J_3 bezüglich der mit $(2,1,1,1)$ gewichteten revers-lexikographischen Ordnung für die Variablenreihenfolge x_1, x_2, x_3, x_4 als

$$\mathcal{G}_3 = \{x_2^2 - x_3x_4,\, x_2x_3^2 - x_1x_4,\, x_1x_2 - x_3^3,\, x_3^5 - x_1^2x_4\},$$

und man hat $I_A = J_4 := (J_3 : x_4^\infty) = J_3$.
Zuletzt berechnet man aus dem gefundenen Erzeugendensystem $\mathcal{G}_3$ von I_A die Gröbnerbasis $\mathcal{G}$ bezüglich einer Ordnung, die c verfeinert. Wir wählen wieder die graduierte lexikographische Ordnung (für die ursprüngliche Reihenfolge x_1, x_2, x_3, x_4) und erhalten wie in den vorangehenden Beispielen

$$\mathcal{G} = \{x_2^2 - x_3x_4,\, x_3^3 - x_1x_2,\, x_2x_3^2 - x_1x_4\}.$$

Wie zuvor reduziert man X_{ini} durch diese Menge, um die Optimallösung von (3.7) zu erhalten.

Anhand der Rechnungen erkennt man, daß in der Idealkette $J_0 \subseteq J_1 \subseteq J_2 \subseteq J_3 \subseteq J_4$ nur die Inklusion $J_1 \subset J_2$ echt ist. Es genügt also, J_0 nach x_2 zu saturieren. Tatsächlich erkennt das Hoşten-Shapiro-Verfahren diese Tatsache und erspart somit drei Gröbnerbasisberechnungen – vorausgesetzt, man startet mit dem zweiten Erzeuger von J_0: Dessen erstes Monom involviert nur x_2, das zweite x_3 und x_4. Also ist $(J_0 : x_2^\infty)$ bereits nach x_3 und x_4 saturiert. Es bleibt nur noch x_1 übrig: Der erste Erzeuger verrät, daß zur Saturierung nach x_1 die Saturierung nach x_2 und x_3 genügt. Diese ist mit der Berechnung von $(J_0 : x_2^\infty)$ bereits vollbracht. Somit berechnet der verbesserte Algorithmus lediglich $(J_0 : x_2^\infty) = I_A$ und dessen reduzierte Gröbnerbasis bezüglich einer Ordnung, die c verfeinert. Startet man allerdings das Hoşten-Shapiro-Verfahren mit dem ersten Erzeuger von J_0, so muß man zwei Saturierungen durchführen: nach x_1 und x_4 oder nach x_2 und x_3. Das Verfahren berechnet also nicht notwendig eine minimale Variablenmenge zur Saturierung. □

Beipiel 3.27 (Di Biase-Urbanke) Mit $(1,-1,-2,1)^\intercal$ hat man bereits einen Gitterbasisvektor von $\ker(A)$ ohne Nullkomponenten. Die zweite und dritte Komponente sind negativ. Also startet man die Berechnungen mit der Matrix

$$A_{2,3} = \begin{pmatrix} 5 & -1 & -2 & 0 \\ 2 & -1 & -1 & 1 \end{pmatrix},$$

für deren Kern die Vektoren $(1,1,2,1)^\intercal$ und $(0,-2,1,-1)^\intercal$ eine Gitterbasis bilden. Da $(1,1,2,1)^\intercal$ nur positive Komponenten hat, gilt

$$I_{A_{2,3}} = \langle\, x_1x_2x_3^2x_4 - 1,\, x_3 - x_2^2x_4 \,\rangle.$$

Daraus berechnet man zunächst ein Erzeugendensystem von I_{A_3}. Dazu wählt man eine monomiale Ordnung mit der Eliminationseigenschaft bezüglich x_2, z.B. die lexikographische Ordnung bezüglich der Variablenreihenfolge x_2, x_1, x_3, x_4. Die reduzierte Gröbnerbasis $\mathcal{G}_{2,3}$ von $I_{A_{2,3}}$ bezüglich dieser Ordnung ist

$$\mathcal{G}_{2,3} = \{x_1^2x_3^5x_4 - 1,\, x_2 - x_1x_3^3\}.$$

Durch Variablenflip von x_2 erhält man das Erzeugendensystem

$$\{x_1^2x_3^5x_4 - 1,\ 1 - x_2x_1x_3^3\}$$

von I_{A_3} – das erste Binom bleibt unverändert, weil x_2 in ihm nicht vorkommt, beim zweiten Binom wird x_2 vom ersten ins zweite Monom geschoben.
Nun braucht man eine monomiale Ordnung mit der Eliminationseigenschaft bezüglich x_3, z.B. die lexikographische Ordnung bezüglich der Variablenreihenfolge x_3, x_2, x_1, x_4, um ein Erzeugendensystem von I_A aus dem Erzeugendensystem von I_{A_3} zu berechnen. Die reduzierte Gröbnerbasis $\mathcal{G}_3$ von I_{A_3} ist dann

$$\mathcal{G}_3 = \{x_2^5 - x_1x_4^3,\ x_3x_1x_4^2 - x_2^3,\ x_3x_2^2 - x_4,\ x_3^2x_1x_4 - x_2,\ x_3^3x_2x_1 - 1\},$$

also ergibt Flippen von x_3

$$I_A = \langle\, x_2^5 - x_1x_4^3,\ x_1x_4^2 - x_3x_2^3,\ x_2^2 - x_3x_4,\ x_1x_4 - x_3^2x_2,\ x_2x_1 - x_3^3 \,\rangle.$$

Die reduzierte Gröbnerbasis von I_A bezüglich der graduierten lexikographischen Ordnung auf $K[x_1, x_2, x_3, x_4]$ (mit dieser Variablenreihenfolge) ergibt sich dann wieder als

$$\mathcal{G} = \{x_2^2 - x_3x_4,\ x_3^3 - x_1x_2,\ x_2x_3^2 - x_1x_4\}.$$

□

Beipiel 3.28 (Bigatti-La Scala-Robbiano) Die zweite Zeile $(2, 1, 1, 1)$ von A definiert wieder die notwendige positive Graduierung von I_A. Die Hilfsvariable u erhält dann das Gewicht $2 + 1 + 1 + 1 = 5$. Für das Startideal

$$I = \langle\, x_1x_2x_3x_4 - u, x_1x_4 - x_2x_3^2, x_2^2 - x_3x_4 \,\rangle$$

berechnet man die reduzierte Gröbnerbasis $\mathcal{G}$ bezüglich der mit $(2, 1, 1, 1, 5)$ gewichteten revers-lexikographischen Ordnung auf $K[x_1, x_2, x_3, x_4, u]$ (in dieser Variablenreihenfolge):

$$\begin{aligned}\mathcal{G} = \{&x_2^2 - x_3x_4,\ x_2x_3^2 - x_1x_4,\ x_1x_2x_4 - x_3^3x_4,\ x_3^4x_4 - u,\ x_1x_3^2x_4^2 - x_2u,\\ &x_1^2x_4^2 - x_3u,\ x_1x_2u - x_3^3u,\ x_3^5u - x_1^2x_4u\}\end{aligned}$$

Die Pseudoelimination tut folgendes: Zunächst wird jedes Binom in $\mathcal{G}$ durch die höchstmögliche Potenz von u geteilt, d.h. die letzten beiden Binome werden durch $x_1x_2 - x_3^3$ und $x_3^5 - x_1^2x_4$ ersetzt. Dann wird jedes Vorkommen von u in den Binomen durch das Variablenprodukt $x_1 \cdot x_2 \cdot x_3 \cdot x_4$ ersetzt; insgesamt erhält man

$$\begin{aligned}I_A = \langle\, &x_2^2 - x_3x_4,\ x_2x_3^2 - x_1x_4,\ x_1x_2x_4 - x_3^3x_4,\ x_3^4x_4 - x_1x_2x_3x_4,\\ &x_1x_3^2x_4^2 - x_1x_2^2x_3x_4,\ x_1^2x_4^2 - x_1x_2x_3^2x_4,\ x_1x_2 - x_3^3,\ x_3^5 - x_1^2x_4 \,\rangle.\end{aligned}$$

Wie üblich ist wieder

$$\mathcal{G} = \{x_2^2 - x_3x_4,\ x_3^3 - x_1x_2,\ x_2x_3^2 - x_1x_4\}$$

die reduzierte Gröbnerbasis von I_A bezüglich der graduierten lexikographischen Ordnung. □

Die Zwischenergebnisse in den Beispielen wurden mit *SINGULAR* ([28]) berechnet.

Kapitel 4

Der Buchberger-Algorithmus für torische Ideale

Natürlich kann der Buchberger-Algorithmus unverändert auf torische Ideale angewandt werden. In praktischen Anwendungen, die nur mit torischen Idealen rechnen, wäre es aber mehr als ungeschickt, eine Implementierung für allgemeine Ideale zu benutzen. Denn die einfache Struktur dieses Idealtyps erlaubt Spezialisierungen, die die Effizienz des Buchberger-Algorithmus entscheidend verbessern. Folgende bereits bekannte Tatsachen erweisen sich dabei als wichtig:

- Torische Ideale werden von reinen Binomen erzeugt (2.16).
- Ihre reduzierten Gröbnerbasen bestehen aus reinen Binomen (2.27).
- Die Operationen des Buchberger-Algorithmus erhalten die binomiale Struktur (2.26).

4.1 Herausdividieren gemeinsamer Monomfaktoren

Nach 2.14 ist das torische Ideal I_A zu einer Matrix $A \in \mathbb{Z}^{m \times n}$ saturiert nach allen Variablen. Mit $x^\alpha x^\beta - x^\alpha x^\gamma$ liegt folglich auch das Binom $x^\beta - x^\gamma$ in I_A. Der Buchberger-Algorithmus darf also während der Berechnung einer Gröbnerbasis von I_A jeden Erzeuger $x^u - x^v$ durch den größten gemeinsamen Teiler $\gcd(x^u, x^v)$ seiner Monome teilen. Dies ist günstig, da die Initialterme der Erzeuger dadurch kleiner werden und eine reduzierte Gröbnerbasis ohnehin nur Erzeuger enthält, deren Initialterme minimal bezüglich der durch Teilbarkeit definierten Teilordnung sind.

Diese Strategie des **Herausdividierens gemeinsamer Monomfaktoren (trivial factors elimination)** läßt sich bei allen vorgestellten Algorithmen anwenden: Mit Ausnahme des Algorithmus von Hoşten und Sturmfels rechnen diese stets mit torischen (und somit saturierten) Idealen. Der Algorithmus von Hoşten und Sturmfels startet zwar mit einem Ideal I, das im allgemeinen nicht saturiert ist; sein einziges Ziel besteht jedoch darin, es möglichst schnell zu saturieren. Das Kürzen gemeinsamer Monomfaktoren während des Buchberger-Algorithmus ersetzt dann Schritt (3).

4.2 Mehrfachreduktion

Seien $m, g \in K[x_1, \dots, x_n] \setminus \{0\}$, m Monom, g beliebig. Wir schreiben g bezüglich einer beliebigen, aber festen Termordnung in der Form $g = \mathrm{in}(g) - \tilde{g}$. Es sei $k_0 := \max\{k \in \mathbb{N} : \mathrm{in}(g)^k \mid m\}$. Dann läßt sich m schreiben als $m = \tilde{m} \cdot \mathrm{in}(g)^{k_0}$, $\tilde{m}$ Monom. Nun dividieren wir m durch g mit Hilfe des Algorithmus von Hironaka 1.12: Ist $k_0 = 0$, so wird kein Divisionsschritt durchgeführt, der Rest der Division ist m selbst. Sonst wird m im ersten Divisionschritt ersetzt durch

$$m - \tilde{m} \cdot \mathrm{in}(g)^{k_0 - 1} g = \tilde{m} \cdot \mathrm{in}(g)^{k_0 - 1} \cdot \tilde{g};$$

im Fall $k_0 \geq 2$ reduziert dies weiter durch g zu $\tilde{m} \cdot \mathrm{in}(g)^{k_0 - 2} \cdot \tilde{g}^2$. Setzt man das Verfahren fort, so erhält man als Ergebnis der Reduktion von m durch g das Polynom $\tilde{m} \cdot \tilde{g}^{k_0}$, d.h.

$$m = \tilde{m} \cdot \mathrm{in}(g)^{k_0} \longrightarrow_{\{g\}} \tilde{m} \cdot \tilde{g}^{k_0}.$$

Ersetzt man m direkt durch dieses Endergebnis, anstatt schrittweise zu reduzieren, so spricht man von **Mehrfachreduktion (multiple reduction)**. Der Algorithmus von Hironaka kann leicht so erweitert werden, daß er immer Mehrfachreduktionen durchführt. Dies ist jedoch nicht unbedingt sinnvoll: Besteht $\tilde{g}$ aus mehreren Termen und ist $k_0 > 1$, so kann die explizite Berechnung der Potenz $\tilde{g}^{k_0}$ teuer und die Anzahl ihrer Monome riesig werden. Die sukzessive Reduktion ist dann die bessere Strategie, da sich im Laufe der Berechnungen Terme wegkürzen können.

Ist g allerdings ein Binom, d.h. $\tilde{g}$ ein Monom, so taucht diese Schwierigkeit nicht auf. Die Mehrfachreduktion erweist sich dann als vorteilhaft, weil man die Exponentenvektoren des zu reduzierenden Monoms und des Initialterms von g jeweils nur einmal betrachtet statt bei jedem einzelnen Divisionsschritt. Die Zahl k_0 kann man an den Exponenten nämlich leicht ablesen.

4.3 Binome als Vektoren

Die Strategien aus den Abschnitten 4.1 und 4.2 wurden bereits von Conti und Traverso in [7] vorgeschlagen. Sie greifen direkt in den Ablauf des Buchberger- bzw. Hironaka-Algorithmus ein. Pottier ([25]) setzt an einer völlig anderen Stelle an, nämlich an der grundlegenden Datenstruktur „Polynom“.

Der Buchberger-Algorithmus für torische Ideale rechnet nur mit Binomen der Form $x^\alpha - x^\beta$, sofern man ihm als Eingabe ein Erzeugendensystem aus solchen Binomen gibt (vgl. 2.26). $x^\alpha - x^\beta$ kann durch das Paar (α, β) der Exponentenvektoren seiner Monome repräsentiert werden. Sind x^α und x^β teilerfremd, so genügt auch ein einziger Vektor, nämlich $\alpha - \beta$; diese Identifikation von reinen Binomen mit Vektoren ist uns schon aus 2.24 bekannt. Nun führen S-Paar-Bildungen und Reduktionen von reinen Binomen nicht notwendig zu *reinen* Binomen; aber der Buchberger-Algorithmus für torische Ideale darf wegen Abschnitt 4.1 von reinen Binomen ausgehen, d.h. mit Vektoren rechnen. Die trivial factors elimination erledigt sich in dieser Darstellung sogar von selbst: Das Binom $x^\gamma x^{\alpha^+} - x^\gamma x^{\alpha^-} = x^{\gamma + \alpha^+} - x^{\gamma + \alpha^-}$, $\alpha \in \mathbb{Z}^n$, $\gamma \in \mathbb{N}^n$, wird mit dem

Vektor $(\gamma+\alpha^+)-(\gamma+\alpha^-)=\alpha$ identifiziert, ohne das die explizite Feststellung des größten gemeinsamen Teilers x^γ seiner Monome notwendig ist.

Wie sehen nun die Operationen des Buchberger-Algorithmus auf diesen Vektoren aus?

Lemma 4.1 *Seien $u, v \in \mathbb{Z}^n \backslash \{0\}$ und $\succ$ eine monomiale Ordnung, bezüglich der $\operatorname{in}(x^{u^+}-x^{u^-})=x^{u^+}$, $\operatorname{in}(x^{v^+}-x^{v^-})=x^{v^+}$ gilt.*

1. *Wenn x^{v^+} durch x^{u^+} teilbar ist, so erhält man nach einem Divisionsschritt von $x^{v^+}-x^{v^-}$ durch $x^{u^+}-x^{u^-}$ wie in 1.12 und anschließendem Herausdividieren gemeinsamer Monomfaktoren das Ergebnis $x^{(v-u)^+}-x^{(v-u)^-}$.*
   Notation: $v \xrightarrow{+}_u v-u$

2. *Wenn x^{v^-} durch x^{u^+} teilbar ist, so erhält man nach einem Divisionsschritt von $x^{v^+}-x^{v^-}$ durch $x^{u^+}-x^{u^-}$ und anschließendem Herausdividieren gemeinsamer Monomfaktoren das Ergebnis $x^{(v+u)^+}-x^{(v+u)^-}$.*
   Notation: $v \xrightarrow{-}_u v+u$

3. *Das S-Polynom von $x^{u^+}-x^{u^-}$ und $x^{v^+}-x^{v^-}$ ist nach dem Herausdividieren gemeinsamer Monomfaktoren gleich $x^{(u-v)^+}-x^{(u-v)^-}$.*
   Notation: $S(u,v)=u-v$

Beweis:

1. Da der Algorithmus von Hironaka monomweise vorgeht, können wir Lemma 2.25 auf das Leitmonom x^{v^+} anwenden; seine Voraussetzungen sind wegen $\operatorname{in}(x^{u^+}-x^{u^-})=x^{u^+}$ und $x^{u^+} | x^{v^+}$ erfüllt. Es liefert als Ergebnis des Divisionsschrittes $x^{v^+-u}-x^{v^-}$. Herausdividieren gemeinsamer Monomfaktoren ergibt wegen $(v^+-u)-v^-=(v^+-v^-)-u=v-u$ das Binom $x^{(v-u)^+}-x^{(v-u)^-}$.

2. Aus $x^{u^+} \succ x^{u^-}$ folgt $x^{u^+} \neq 1$. Da x^{v^+} und x^{v^-} teilerfremd sind, ist x^{v^+} somit nicht durch x^{u^+} teilbar, so daß x^{v^+} von dem Divisionsschritt nicht betroffen ist. Anwendung von 2.25 auf das Monom x^{v^-} liefert als Ergebnis des Divisionsschrittes das Binom $x^{v^+}-x^{v^--u}$, Herausdividieren gemeinsamer Monomfaktoren ergibt wegen $v^+-(v^--u)=v+u$ das Binom $x^{(v+u)^+}-x^{(v+u)^-}$.

3. Das S-Polynom von $x^{u^+}-x^{u^-}$ und $x^{v^+}-x^{v^-}$ ist das Binom
$$x^{(u^+-v^+)^+}x^{v^-}-x^{(v^+-u^+)^+}x^{u^-}.$$
Die Differenz der Exponenten seiner Monome ist
$$\begin{aligned} & ((u^+-v^+)^+ + v^-) - ((v^+-u^+)^+ + u^-) \\ = \; & ((u^+-v^+)^+ + v^-) - ((u^+-v^+)^- + u^-) \\ = \; & (u^+-v^+) + v^- - u^- = (u^+-u^-) - (v^+-v^-) = u-v, \end{aligned}$$
also liefert Herausdividieren gemeinsamer Monomfaktoren das Binom $x^{(u-v)^+}-x^{(u-v)^-}$ $\square$

Bemerkung 4.2 Die Operationen des Buchberger-Algorithmus auf torischen Idealen kann man also als Vektoradditionen bzw. -subtraktionen auffassen. □

Bemerkung 4.3 Unsere IP-Algorithmen aus Kapitel 3 müssen nicht nur torische Ideale berechnen, sondern auch Monome modulo diesen Idealen reduzieren. Wie behandelt man nun Monome im Rahmen der Binomdarstellung mit Vektoren? Wir identifizieren ein Monom $x^v \in K[x_1, \ldots, x_n]$ mit dem Vektor $v \in \mathbb{N}^n$; über die Korrespondenz Vektoren–Binome entspricht dieser dem *Binom* $x^v - 1$. Der hinzugekommene Summand -1 stört aber nicht: Denn das Monom 1 ist nach dem Beweis von 3.10 in keinem torischen Ideal I_A enthalten; reduziert also x^v modulo I_A zu x^w, so reduziert $x^v - 1$ zu $x^w - 1$. Für das Reduktionsergebnis w von v ist es somit egal, ob man v mit dem Monom x^v oder dem Binom $x^v - 1$ identifiziert. □

Die Ergebnisse dieses Abschnitts haben enorme Auswirkungen hinsichtlich einer Implementierung des Buchberger-Algorithmus für torische Ideale: Sie reduzieren die Polynomstruktur und den Algorithmus selbst auf eine Form, in der beide für allgemeinere Ideale völlig nutzlos sind. Der Buchberger-Algorithmus wird dadurch schneller, die Implementierung der Polynome leichter. Einen Eindruck, wieviel Sorgfalt das im allgemeinen Fall erfordert, bekommt man in [1].

4.4 Binäre Trägervektoren

Ein Großteil der Arbeit des Buchberger-Algorithmus besteht darin, Monome auf Teilbarkeit zu testen. Das sollte man daher möglichst effizient tun. Die naive Vorgehensweise testet Teilbarkeit anhand der Exponentenvektoren:

$$x^\alpha \mid x^\beta \;\Leftrightarrow\; \alpha \leq \beta \text{ komponentenweise}$$

Für jeden Teilbarkeitstest müssen also bis zu n ganze Zahlen verglichen werden, wobei n wie üblich die Anzahl der Variablen ist.

Definition 4.4 Sei $x^\alpha \in K[x_1, \ldots, x_n]$, $\alpha \in \mathbb{N}^n$. Der **Träger (support)** von x^α bzw. α ist definiert als die Menge

$$\operatorname{supp}(x^\alpha) := \operatorname{supp}(\alpha) := \{i \in \{1, \ldots, n\} \mid \alpha_i \neq 0\}.$$

Der **binäre Trägervektor** $\sigma_\alpha \in \{0,1\}^n$ zu α ist gegeben durch

$$(\sigma_\alpha)_i = \begin{cases} 1, & \text{falls } i \in \operatorname{supp}(\alpha), \\ 0, & \text{falls } i \notin \operatorname{supp}(\alpha), \end{cases} \quad i = 1, \ldots, n. \square$$

Bemerkung 4.5 Die Teilmengen $\mathcal{S}$ von $\{1, \ldots, n\}$ entsprechen eineindeutig den binären Vektoren $\sigma \in \{0,1\}^n$ vermöge der Abbildung

$$\mathcal{P}(\{1, \ldots, n\}) \ni \mathcal{S} \longmapsto \sigma \in \{0,1\}^n$$

$$\text{mit } \sigma_i = \begin{cases} 1, & \text{falls } i \in \mathcal{S}, \\ 0, & \text{falls } i \notin \mathcal{S}, \end{cases} \quad i = 1, \ldots n.$$

Dabei bezeichne $\mathcal{P}(\mathcal{M})$ die Potenzmenge einer Menge $\mathcal{M}$.

Die binären Vektoren $\sigma \in \{0,1\}^n$ entsprechen eineindeutig den ganzen Zahlen $s \in \{0,\ldots,2^n-1\}$ vermöge der Abbildung

$$\{0,1\}^n \ni \sigma = (\sigma_1,\ldots,\sigma_n) \longmapsto s = \sum_{i=1}^{n} \sigma_i \, 2^{i-1} \in \{0,\ldots,2^n-1\};$$

Anschaulich interpretiert diese Abbildung den binären Vektor σ als Binärdarstellung von s. □

Der Computer rechnet mit solchen Binärdarstellungen: Ganze Zahlen werden in der Regel als Bitvektoren fester Länge realisiert. Über obige Interpretationen, zwischen denen wir im folgenden nicht unterscheiden, können wir den Träger eines Monoms als Binärzahl auffassen und sehr effizient implementieren.

Beipiel 4.6 Betrachte das Monom $m = x_1x_2^2x_4 \in K[x_1,x_2,x_3,x_4]$. x_3 kommt als einzige Variable in m nicht vor; man erhält also $\mathrm{supp}(m) = \{1,2,4\}$ und $\sigma_m = (1,1,0,1)$. σ_m entspricht der Zahl

$$1011_2 = 1\cdot 2^0 + 1\cdot 2^1 + 0\cdot 2^2 + 1\cdot 2^3 = 11_{10},$$

wobei der Index angibt, zu welcher Basis die Zahl gelesen werden soll. □

Definition 4.7 Sei $b = x^\alpha - x^\beta$, $\alpha,\beta \in \mathbb{N}^n$. Dann heißt x^α der **Kopf (head)** und x^β der **Schwanz (tail)** von b. Ist $\alpha \in \mathbb{Z}^n$, so bezeichnen wir $\alpha^+ \in \mathbb{N}^n$ als **Kopf** und $\alpha^- \in \mathbb{N}^n$ als **Schwanz** von α. □

Bemerkung 4.8

1. Die Bezeichnungen sind so gewählt, daß man äquivalent vom Kopf bzw. Schwanz des Binoms $x^{\alpha^+} - x^{\alpha^-}$ oder vom Kopf bzw. Schwanz des Vektors α reden kann.

2. Bei reinen Binomen ist der Träger des Kopfes disjunkt zum Träger des Schwanzes. □

Was nutzt es uns nun, Trägervektoren zu betrachten?

Bemerkung 4.9

1. Ein Monom x^β ist durch ein Monom x^α teilbar genau dann, wenn α komponentenweise kleiner als β ist. Wegen $\alpha,\beta \geq 0$ impliziert das, daß der Träger von β den Träger von α umfaßt:

$$\mathrm{supp}(\alpha) \subseteq \mathrm{supp}(\beta)$$

ist eine notwendige Bedingung für die Teilbarkeit. Sie ist nicht hinreichend, wie man am Beispiel $x^\alpha = x_1x_2^2$, $x^\beta = x_1^2x_2$ sieht. Sie hilft jedoch, viele Teilbarkeitstests zu vermeiden: Testet man die Teilbarkeit mit komponentenweisem Vergleich der Exponentenvektoren, so muß man für $x^\beta = x_1x_2x_3$ und $x^\alpha = x_1x_2x_3x_4$ bis zu vier Komponenten vergleichen, bis man feststellt, daß x^β nicht durch x^α teilbar ist. Wegen $\mathrm{supp}(\alpha) = \{1,2,3,4\} \not\subseteq \{1,2,3\} = \mathrm{supp}(\beta)$ kann man das an den Trägervektoren sofort ablesen – wie, dazu später.

2. Eine effiziente Implementierung des Buchberger-Algorithmus nutzt einige der Kriterien aus 1.27, 1.28 und 1.29 zur Erkennung von S-Paaren, die zu 0 reduzieren. Kriterium 1.27 prüft, ob zwei Monome x^α und x^β teilerfremd sind. Dies ist genau dann der Fall, wenn keine Variable in beiden gleichzeitig vorkommt, d.h. wenn gilt

$$\operatorname{supp}(\alpha) \cap \operatorname{supp}(\beta) = \emptyset.$$

Prüft man Teilerfremdheit anhand der Exponentenvektoren α und β, so muß man eventuell alle Komponenten betrachten.

3. Die Kriterien 1.28 und 1.29 prüfen Monome auf Teilbarkeit, Gleichheit und Ungleichheit. Eine notwendige Bedingung für die Gleichheit zweier Monome x^α und x^β ist natürlich

$$\operatorname{supp}(\alpha) = \operatorname{supp}(\beta);$$

äquivalent ist

$$\operatorname{supp}(\alpha) \neq \operatorname{supp}(\beta)$$

eine hinreichende Bedingung für Ungleichheit.

4. Ferner prüfen diese Kriterien das kleinste gemeinsame Vielfache $\operatorname{lcm}(x^\alpha, x^\beta)$ zweier Monome x^α und x^β. Eine Variable kommt im kleinsten gemeinsamen Vielfachen genau dann vor, wenn sie in mindestens einem der beiden Monome vorkommt. Somit gilt

$$\operatorname{supp}(\operatorname{lcm}(x^\alpha, x^\beta)) = \operatorname{supp}(\alpha) \cup \operatorname{supp}(\beta).$$

Man muß also das kleinste gemeinsame Vielfache nicht explizit berechnen, um seinen Träger zu bestimmen und zu benutzen. □

Man sieht, daß man den Vergleich von Vektorkomponenten an vielen Stellen durch Mengenoperationen auf den Trägern ersetzen kann. Damit das sinnvoll ist, müssen die Mengenoperationen sehr effizient sein. Wir erklären nun , wie man sie in einem konkreten Fall in konstanter Zeit ausführen kann.

Definition 4.10 Sei $r \in \mathbb{N}$. Die Funktionen

1. **bitweises Und:**

$$\& : \{0,1\}^r \times \{0,1\}^r \longrightarrow \{0,1\}^r; (u,v) \longmapsto (u\&v),$$

wobei $(u\&v)_i := \begin{cases} 1, & \text{falls } u_i = 1 \text{ und } v_i = 1, \\ 0, & \text{sonst,} \end{cases} \quad i = 1,\ldots,n;$

2. **bitweises Oder:**

$$| : \{0,1\}^r \times \{0,1\}^r \longrightarrow \{0,1\}^r; (u,v) \longmapsto (u|v),$$

wobei $(u|v)_i := \begin{cases} 1, & \text{falls } u_i = 1 \text{ oder } v_i = 1, \\ 0, & \text{sonst,} \end{cases} \quad i = 1,\ldots,n;$

3. **bitweises Komplement:**

$$\sim : \{0,1\}^r \longrightarrow \{0,1\}^r ; u \longmapsto (\sim u),$$

$$\text{wobei } (\sim u_i) := \begin{cases} 1, & \text{falls } u_i = 0, \\ 0, & \text{falls } u_i = 1, \end{cases} \quad i = 1, \ldots, n,$$

heißen **bitweise Funktionen** auf $\{0,1\}^r$. □

Für den Rest des Abschnitts setzten wir folgendes voraus:

(V) Die Umgebung, für die wir den Buchberger-Algorithmus implementieren, realisiert ganze Zahlen als Bitvektoren fester Länge, etwa eine ganze Zahl als r aufeinanderfolgende Bits. Auf diesen Bitvektoren können die bitweisen Funktionen aus Definition 4.10 in konstanter Zeit ausgeführt werden. Ebenso können zwei Bitvektoren in konstanter Zeit auf (Un-)Gleichheit getestet werden.

Wenn man die Programmiersprache C ([19]) und ihre eingebauten Typen `short`, `int` oder `long` für ganze Zahlen benutzt, ist (V) in der Regel erfüllt. Die bitweisen Operationen aus der Definition wurden auch wie in C benannt. Natürlich ergibt sich bei obiger Zahlendarstellung das Problem, daß man nur einen begrenzten Zahlenbereich darstellen kann. Dem gehen wir zunächst aus dem Wege, indem wir $n \leq r$ voraussetzen. Dann passen unsere Trägervektoren, die ja Länge n haben, auf jeden Fall in den für eine ganze Zahl vorgesehenen Speicherplatz.

Lemma 4.11 *Es seien $\mathcal{S}_1$ und $\mathcal{S}_2$ zwei Teilmengen von $\{1,\ldots,r\}$ und $\sigma_1, \sigma_2 \in \{0,1\}^r$ die zugehörigen binären Vektoren im Sinne von 4.5. Dann gilt:*

1. *Der Menge $\mathcal{S}_1 \cap \mathcal{S}_2$ entspricht der Vektor $(\sigma_1 \& \sigma_2)$.*
2. *Der Menge $\mathcal{S}_1 \cup \mathcal{S}_2$ entspricht der Vektor $(\sigma_1 | \sigma_2)$.*
3. *Der Menge $\{1,\ldots,n\} \backslash \mathcal{S}_1$ entspricht der Vektor $\sim\sigma_1$.*

Beweis: Ergibt sich sofort aus der Definition der bitweisen Funktionen. □

Lemma 4.12 *Seien $\mathcal{S}_1, \mathcal{S}_2, \sigma_1, \sigma_2$ wie in 4.11. dann gilt:*

1. $\mathcal{S}_1 \subseteq \mathcal{S}_2 \;\Leftrightarrow\; \mathcal{S}_1 \cap \mathcal{S}_2 = \mathcal{S}_1 \;\Leftrightarrow\; \mathcal{S}_1 \cup \mathcal{S}_2 = \mathcal{S}_2$
2. $\mathcal{S}_1 \subseteq \mathcal{S}_2 \;\Leftrightarrow\; (\sigma_1 \& \sigma_2) = \sigma_1 \;\Leftrightarrow\; (\sigma_1 | \sigma_2) = \sigma_2$

Beweis: 1. ist trivial, 2. folgt unmittelbar aus 1. und 4.11. □

Mit Hilfe dieser Lemmata können wir die Bedingungen bzw. Berechnungen für Träger aus Bemerkung 4.9 in konstanter Zeit überprüfen bzw. ausführen. Es bleibt nur zu bemerken, daß der leeren Menge in Punkt 2. der Nullvektor entspricht.

Die Berechnung des Trägers von x^α erfordert es, jede Komponente von α anzusehen. Das lohnt sich im Rahmen des Buchberger-Algorithmus nur, weil ein einmal berechneter Träger in der Regel mehrmals genutzt werden kann: Ein Polynom verändert sich nämlich höchstens bei einer Reduktion, nicht aber z.B. bei der S-Paar-Bildung. Am häufigsten braucht man die Träger der Initialmonome, z.B. bei der Überprüfung von S-Paar-Kriterien. Die der anderen Monome kann man bei Teilbarkeitstests im Zusammenhang mit Reduktionen verwenden.

Wie wertvoll die Information, die im Träger steckt, für den Buchberger-Algorithmus ist, hängt vom Aussehen des betrachteten Ideals ab. Torische Ideale werden von Binomen mit teilerfremden Monomen erzeugt. Daß ein Monom alle Variablen enthält, kommt bei solchen Erzeugern relativ selten vor – eine wichtige Voraussetzung für die erfolgreiche Anwendung von 4.9.

Bemerkung 4.13 Wir haben bis jetzt $n \leq r$ vorausgesetzt, so daß unsere Trägervektoren in den Speicherplatz passen, der für eine ganze Zahl vorgesehen ist. Dies ist zu restriktiv, denn gerade bei hoher Variablenzahl ist die Trägernutzung interessant. Für den Fall $n > r$ haben wir zwei prinzipielle Alternativen:

1. Wir betrachten weiterhin alle Variablen und teilen den Trägervektor in $\lceil \frac{n}{r} \rceil$ Stücke, die jeweils in r Bits passen. Diese Methode verursacht einen beträchtlichen Programmier- und Verwaltungsaufwand.

2. Einfacher ist es, den Träger nur bezüglich einer Teilmenge von maximal r Variablen zu bilden. Das genügt immer noch, um in vielen Fällen Gleichheit oder Teilbarkeit von Monomen auszuschließen. Die restlichen Fälle behandelt man mit den „herkömmlichen“ Methoden; Details findet man in der in Kapitel 5 vorgestellten Implementierung. □

Bemerkung 4.14 (Erweiterte Trägernutzung)
Da wir Träger als ganze Zahlen implementieren, eignen sie sich gut zur Indizierung von Feldern. Dadurch erhält man eine interessante Möglichkeit zur Speicherung von Idealerzeugern: Statt *einer* Liste verwaltet man ein Feld $\mathcal{L}_0, \ldots, \mathcal{L}_{2^k-1}$ von 2^k Listen, auf die die Erzeuger gemäß ihres Trägers bezüglich der ersten k Variablen verteilt werden. Konkret wird ein Polynom mit Initialterm x^α in der Liste mit Index $(supp(\alpha) \bmod 2^k)$ gespeichert. Die Vorteile dieser Aufspaltung erkennt man beim Reduzieren eines Monoms modulo der Idealerzeuger: Einen Teil der Listen braucht man nicht zu durchmustern, da man schon anhand der Indizes weiß, daß die Initialterme ihrer Elemente das Monom nicht teilen. Auf diese Weise spart man einige Vergleiche von Trägervektoren.

Bei der Wahl von k sollte man allerdings bescheiden bleiben: Ist k zu groß, verursachen leere Listen mehr Arbeit, als man durch die Aufspaltung spart. □

Die Einführung von Trägervektoren für Teilbarkeitstests wurde in [2] vorgeschlagen, allerdings in einem anderen Kontext, so daß die meisten anderen Anwendungen dort nicht zur Sprache kommen.

Kapitel 5

Eine Implementierung

Dieses Kapitel führt in die Implementierung der Algorithmen aus Kapitel 3 ein, deren Quellcode man in Anhang A findet. Die Programme wurden in C++ geschrieben (siehe [29] für eine Einführung in diese Programmiersprache); entsprechend ist das Design weitgehend **objektorientiert:** Die wichtigsten Datentypen sind als **Klassen** implementiert, mit deren Hilfe sich die eigentlichen Algorithmen kurz formulieren lassen.

Der Programmtext ist ausführlich kommentiert, und die Namen von Variablen und Funktionen sind so gewählt, daß sie deren Bedeutung wiedergeben. Nicht laufzeitkritische Funktionen schützen sich weitgehend gegen Mißbrauch, indem sie die Werte ihrer Argumente überprüfen.

Dennoch tauchen im Code schwierige Stellen auf: Zum Beispiel ist die Verwendung von Zeigern und Referenzen sorgfältig optimiert und nicht immer leicht zu verstehen. Solche Kompromisse sind unumgänglich, weil es bei Algorithmen zur ganzzahligen Optimierung sehr auf die Effizienz ankommt.

Wer das Programm nur benutzen und nicht abändern möchte, braucht sich darum selbstverständlich nicht zu kümmern. Abschnitt 5.3 erklärt, was man dazu wissen muß.

5.1 Die Klassen

Wir geben zunächst einen kurzen Überblick über die Klassenstruktur. Dabei spezifizieren wir nicht detailliert die Schnittstellen der Klassen; solche Spezifikationen kann man im Kommentar der Header-Dateien (`.h`-Dateien, Anhang A.1) nachlesen. Stattdessen erklären wir den Aufbau der Klassen und Zusammenhänge, die man aus dem Quelltext nicht sofort ablesen kann.

Alle Klassen stellen Routinen folgender Art zur Verfügung:

- **Konstruktoren,** die neue Objekte der Klasse (d.h. Variablen dieses Typs) anlegen. Sie erhalten oft Argumente, die vorschreiben, wie die Objekte zu initialisieren sind.
- Einen **Destruktor,** der ein Objekt der Klasse zerstört, d.h. den von ihm belegten Speicher wieder freigibt.
- **Zugriffsfunktionen,** die gewisse Eigenschaften eines Objektes abfragen bzw. verändern können.

- **Zuweisungsoperatoren,** die einem Objekt die Eigenschaften eines anderen Objekts (nicht notwendig aus derselben Klasse) zuweisen.
- **Vergleichsoperatoren,** die Relationen zwischen zwei Objekten testen (z.B. $=$, $\leq$).
- **Ausgaberoutinen** zum Schreiben eines Objekts auf den Bildschirm oder in Dateien. Die Eingabe geschieht in unseren Programmen über die Konstruktoren.

Solche Prozeduren sind meist einfach zu verstehen und zu implementieren. Deshalb gehen wir in unseren Klassenbeschreibungen nicht näher auf sie ein.

Da unsere Implementierung nicht zuletzt dazu gedacht ist, verschiedene Möglichkeiten für den torischen Buchberger-Algorithmus auszuprobieren, bietet sie einige Alternativen zur Konfiguration an. Damit dies nicht zu ständigen Abfragen auf Kosten der Effizienz führt, benötigt man globale Konstanten und Makros. Diese sind in der Datei `globals.h` zusammengefaßt, die alle Klassen direkt oder indirekt einbinden; insbesondere kann man dort mit Hilfe des Präprozessors Optionen für die Compilierung einstellen.

5.1.1 Binome

Wir rechnen nur mit reinen Binomen. Ein solches Binom in $K[x_1, \ldots, x_n]$ speichern wir als Vektor, d.h. als Feld der Größe n mit ganzzahligen Einträgen, wie in Abschnitt 4.3 beschrieben.

C++ bietet mehrere Typen für ganze Zahlen, unter anderem `short`, `int` und `long`, die sich in der Anzahl der Bits und damit in der Größe des darstellbaren Zahlenbereichs unterscheiden. Die Datei `globals.h` erlaubt es, einen dieser Typen über `typedef` auszuwählen; der benutzte Typ heißt im Quelltext stets `Integer`. Die Voreinstellung benutzt den Typ `long`; die Verwendung von `short` benötigt zwar weniger Speicher, kann aber schon bei kleinen Instanzen einen Überlauf verursachen (siehe Kapitel 6).

Zusätzlich speichern wir zu jedem Binom zwei Bitvektoren vom Typ `long`, die die Träger von Kopf und Schwanz enthalten – genauer die Teile, die in eine `long`-Variable passen. Man hat zwei Möglichkeiten, die Variablen zur Bildung der Träger auszuwählen, falls die Variablenzahl zu groß ist: Die Option `SUPPORT_VARIABLES_FIRST` nimmt die vorderen, `SUPPORT_VARIABLES_LAST` die hinteren Variablen. Einstellen muß man das wiederum in der Datei `globals.h`, wo man die Benutzung der Trägervektoren auch ganz abschalten kann – man ersetze `SUPPORT_DRIVEN_METHODS` durch `NO_SUPPORT_DRIVEN_METHODS`.

Aus technischen Gründen fassen wir übrigens auch das Nullpolynom als Binom auf und repräsentieren es durch den Nullvektor.

Die wichtigsten Funktionen der Binomklasse stammen aus drei Kategorien:

1. **Grundroutinen für den Buchberger-Algorithmus:** Dazu gehören die S-Paar-Bildung und die Reduktion von Binomen. Letztere realisieren wir als Mehrfachreduktion wie in Abschnitt 4.2.
2. Routinen zur **Überprüfung der S-Paar-Kriterien** aus 1.27 bis 1.29.

3. **Spezielle Routinen für die IP-Algorithmen** aus Kapitel 3: Darunter fallen Funktionen, die die Variablenreihenfolge ändern, Variablen eliminieren oder Kopf und Schwanz an eine neue monomiale Ordnung anpassen.

Die Deklaration der Binomklasse (`class binomial`) findet man in der Datei `binomial.h`, ihre Implementierung in `binomial.cc`.

5.1.2 Monomiale Ordnungen

Monomiale Ordnungen tauchen als Argument vieler Funktionen der Binomklasse auf. Umgekehrt benötigt die Klasse der monomialen Ordnungen auch die Binomklasse. Als Benutzer dieser Klassen muß man darauf achten, daß die Binome und die monomialen Ordnungen hinsichtlich der Variablenzahl zusammenpassen. Für unsere IP-Algorithmen genügen recht einfache Ordnungen, nämlich Blockordnungen mit zwei Blöcken, wobei der eine Block die Eliminationsvariablen umfaßt (und eventuell leer ist), der andere die mit der Kostenfunktion gewichteten Variablen. Wir speichern also für eine monomiale Ordnung zwei Zahlen, die die Größe dieser Blöcke angeben. Deren Summe entspricht dann der Anzahl von Variablen, für die die Termordnung ausgelegt ist.

Nun braucht man eine Möglichkeit, die monomialen Ordnungen auf den Blöcken auszuwählen. Für den Eliminationsblock stehen die lexikographische, die graduierte lexikographische und die revers-lexikographische Ordnung zur Verfügung. Für den gewichteten Block muß man einen Gewichtsvektor angeben sowie eine monomiale Ordnung zur Verfeinerung der durch das Gewicht gegebenen Teilordnung. Neben den drei erwähnten Ordnungen bieten wir dafür die gewichtete revers-lexikographische Ordnung an – falls der Gewichtsvektor nur positive Komponenten besitzt.

Für jeden der beiden Blöcke wird also eine weitere Zahl gespeichert, die eine der obigen Ordnungen codiert; damit man sich die Codierung leichter merken kann, bekommen die Zahlen in `globals.h` schöne Namen: Zum Beispiel bezeichnet `LEX` die Zahl 1, die für die lexikographische Ordnung auf dem Eliminationsblock steht.

Die Klasse der monomialen Ordnungen stellt darüberhinaus eine Option `homogeneous` zur Verfügung, um effizient mit homogenen Idealen rechnen zu können, was z.B. den Algorithmen von Hoşten und Sturmfels sowie von Bigatti, La Scala und Robbiano entgegenkommt. Ist `homogeneous` für ein Objekt `w` der Klasse auf `TRUE` gesetzt, so vergleicht `w` bei der Bestimmung des Initialterms eines Binoms `b` dessen Monome nicht erst anhand des Gewichts, sondern direkt anhand der verfeinernden Ordnung. Dieses Vorgehen ist natürlich nur dann korrekt, wenn `b` homogen bezüglich der durch den Gewichtsvektor gegebenen Graduierung ist. Die Konstruktion ist jedoch nicht gegen Mißbrauch geschützt.

Die Klasse der monomialen Ordnungen (`class term_ordering) ist in term_ordering.h` deklariert und in `term_ordering.cc` implementiert. Sie stellt unter anderem Funktionen folgender Art zur Verfügung:

1. **Vergleichsfunktionen** zur Bestimmung des Initialterms eines Binoms bzw. zum Vergleich der Initialterme zweier Binome bezüglich der betrachteten Termordnung.

2. Routinen zur **Manipulation von monomialen Ordnungen:** Die Algorithmen aus Kapitel 3 müssen z.B. zwischen Eliminationsordnungen und gewichteten Ordnungen wechseln und den Gewichtsvektor an veränderte Variablenreihenfolgen anpassen können.

5.1.3 Matrizen

Bei einem IP-Problem besteht ein wesentlicher Teil der Eingabe aus der Matrix A, die zusammen mit der rechten Seite b die Nebenbedingungen definiert. In den Dateien `matrix.h` und `matrix.cc` findet man Deklaration und Definition einer einfachen Matrixklasse (`class matrix`). Neben der Möglichkeit, Matrizen aus einer Datei einzulesen, bietet die Klasse Funktionen, die einen Teil der Vorarbeit für die IP-Algorithmen erledigen, nämlich

1. die Berechnung einer LLL-reduzierten **Gitterbasis** des Matrixkerns,
2. die Berechnung eines Gitterbasisvektors ohne Nullkomponenten für den Algorithmus von Di Biase und Urbanke,
3. die Berechnung der Flipvariablen für diesen sowie
4. die Berechnung von Inversionsvariablen mit der Hoşten-Shapiro-Prozedur.

Eine Matrix besteht hauptsächlich aus einem zweidimensionalen `Integer`-Feld, in dem ihre Koeffizienten gespeichert werden. Zusätzlich merkt sie sich, sobald berechnet, ihre Gitterbasis. Der Zugriff auf diese ist praktisch nur der Idealklasse gestattet – eine nicht sehr schöne, aber einfache Lösung, die gewählt wurde, weil die Matrixklasse ohnehin als Hilfsklasse für die Klasse Ideal 5.1.5 konzipiert wurde. Sie stellt keinerlei Funktionen bereit, die sie für eine allgemeinere Verwendung qualifizieren, insbesondere keine arithmetischen Operationen.

5.1.4 Listen

Der Buchberger-Algorithmus muß eine Menge von Polynomen verwalten. Da man nicht im voraus sagen kann, wie groß diese Menge für ein gegebenes Ideal wird, implementiert man sie am besten als eine Liste. Dazu stellen die Dateien `list.h` und `list.cc` zwei Klassen zur Verfügung: eine Listenklasse (`class list`) und eine Iteratorenklasse (`class list_iterator`), die die Verwaltung der Erzeuger übernehmen. Ebenso wie die Matrixklasse sind diese Klassen auf die speziellen Erfordernisse zugeschnitten und nicht für eine allgemeinere Verwendung gedacht. Weshalb und wie sie sich die Arbeit teilen, wird weiter unten erläutert.

Die Listenklasse besteht im wesentlichen aus einem Zeiger auf den Listenanfang. Ein Listenelement setzt sich zusammen aus

- einem Zeiger auf ein Binom. Das referenzierte Binom ist das, was den Buchberger-Algorithmus eigentlich interessiert. Man speichert einen Zeiger, da das Umsetzen von Zeigern billiger ist, als Binome im Speicher hin- und herzuschieben. Entsprechend geschieht das Einfügen bzw. Löschen von Elementen meistens durch Umsetzen von Zeigern: Man bewirkt dadurch, daß ein Binom über die Liste gefunden bzw. nicht mehr gefunden wird.

- zwei `BOOLEAN`-Variablen `done` und `head_reduced` für den Buchberger-Algorithmus. Mit deren Hilfe verhindert dieser beispielsweise, daß ein S-Paar zweimal berechnet wird.

- einem Zeiger `next` auf das nächste Element.

- einem Zeiger `previous` auf das vorhergehende Element, falls man eine doppelt verkettete Liste benutzt (vgl. Beispiel 5.1); bei einer einfach verketteten Liste entfällt dieser Zeiger.

`class list` erledigt hauptsächlich das Einfügen von Elementen. Neue Idealerzeuger, die vom Buchberger-Algorithmus berechnet werden, können im Prinzip immer am Anfang der Liste eingefügt werden, was nur konstante Zeit benötigt. Um geordnete Erzeugerlisten zu implementieren, was manche Strategien für den Buchberger-Algorithmus vorsehen, muß man allerdings an beliebiger Stelle in die Liste einfügen können. Das Suchen der entsprechenden Position kann Zeit proportional zur Anzahl der Listenelemente beanspruchen.

Die Listenklasse stellt zu Vergleichszwecken beide Möglichkeiten zur Verfügung. Man beachte jedoch, daß geordnete Erzeugerlisten in Kombination mit der Listenaufspaltung bei erweiterter Trägernutzung wenig sinnvoll sind.

Alle anderen Manipulationen von Listenelementen werden von der Iteratorklasse vorgenommen. Iteratoren sind im Prinzip nichts anderes als Zeiger, mit denen man über die Listenelemente laufen kann. Dadurch, daß man diese Zeiger in eine Klasse packt, wird der Programmcode leserlicher und weniger fehleranfällig. Iteratoren braucht man immer dann, wenn im Buchberger-Algorithmus nacheinander alle Erzeuger angesehen werden – also ständig. Betrachtet man wie bei der S-Paar-Bildung Paare von Erzeugern, braucht man zwei Iteratoren, bei Tripeln (Gebauer-Möller-Kriterien 1.28) entsprechend drei. Daß die Iteratoren nicht innerhalb der Listenklasse gespeichert werden, liegt einerseits an ihrer variierenden Anzahl; andererseits braucht man sich so nicht zu merken, auf welcher Liste man gerade iteriert, was bei Aufspaltung der Erzeuger in mehrere Listen außerordentlich bequem ist.

Da die Iteratoren beim Buchberger-Algorithmus immer auf die gerade betrachteten Elemente zeigen, liegt es nahe, der Iteratorklasse neben dem Abfragen von Elementen auch das Löschen und die Manipulation der Variablen `done` und `head_reduced` zu übertragen. Für all diese Zwecke stellt die Iteratorklasse Funktionen bereit.

Leider verschlingt die Zeigerverfolgung viel Laufzeit, ohne etwas zu den eigentlichen Berechnungen beizutragen. Deshalb ist eine Optimierung der Listenstruktur wichtig. Folgendes Beispiel vermittelt einen Eindruck, wieviel Sorgfalt man in eine effiziente Implementierung investieren muß:

Beipiel 5.1

1. Wir möchten ein Element `e` aus einer Liste löschen. (Der Einfachheit halber nehmen wir an, daß `e` nicht das erste oder das letzte Listenelement ist.) Dazu müssen wir den `next`-Zeiger des vorangehenden Elements auf den Nachfolger von `e` setzen. Das vorangehende Element kann nur über den `previous`-Zeiger von `e` angesprochen werden.

   Zunächst sieht es so aus, als bräuchte man eine doppelt verkettete Liste, denn nur diese verwaltet `previous`-Zeiger. Mit einem Trick gelingt es dennoch, eine einfach verkettete Liste zu benutzen, die pro Element einen Zeiger einspart: Man kopiert die Komponenten des nächsten Elements in die des aktuellen und löscht dann das nächste Element.

   Das hört sich umständlich an. Da man wenige und einfache Einträge hat, erweist sich dieses Vorgehen aber tatsächlich als schneller als die Verwendung einer doppelt verketteten Liste (siehe dazu die Testdaten in Kapitel 6, Tabellen 6.1 und 6.14).

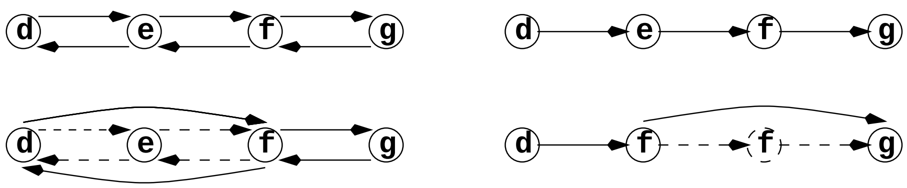


Abbildung 5.1: Löschen in doppelt und einfach verketteten Listen

2. Allerdings birgt die einfach verkettete Liste schwer erkennbare Gefahren: Wenn z.B. ein Iterator `it1` auf das Element `e` und ein Iterator `it2` auf seinen Nachfolger zeigt und man `e` in diesem Moment löscht, so zeigt der scheinbar unbeteiligte Iterator `it2` plötzlich ins Leere. □

In `list.h` kann man zwischen der Verwendung von doppelt und einfach verketteten Listen (`DL_LIST` und `SL_LIST`) wählen. Die Gefahrenstellen bei Verwendung einfach verketteter Listen sind im Buchberger-Algorithmus sorgfältig kommentiert.

5.1.5 Ideale

Die komplizierteste unserer Klassen ist die der Ideale. Ihre Struktur hängt davon ab, ob man sich für erweiterte Trägernutzung entscheidet oder nicht. Dies kann man in `globals.h` über die Option `SUPPORT_DRIVEN_METHODS_EXTENDED /NO_SUPPORT_DRIVEN_METHODS_EXTENDED` tun. Ohne erweiterte Trägernutzung besteht ein Ideal im wesentlichen aus einer Liste zum Speichern der Erzeuger und einer monomialen Ordnung, bezüglich der alle Berechnungen durchgeführt werden. Letztere wird direkt mit dem Ideal gespeichert, um Effizienzverluste zu vermeiden; andernfalls müßte sie beinahe allen Funktionen der Idealklasse als Argument übergeben werden. Die Idealklasse hat noch einige weitere Member (Variablen, die zur Klasse gehören); diese dienen der Kontrolle des Buchberger-

Algorithmus und sind in der Datei `ideal.h` erklärt, die die Klassendeklaration enthält.

Im Falle der erweiterten Trägernutzung spaltet sich die Erzeugerliste in mehrere Listen auf, und zwar in 2^l Stück, wobei l der Wert der Konstanten `List_Support_Variables` in `globals.h` ist. Die Voreinstellung ist $l = 8$. Neben diesen Listen verwalten wir einen `subset_tree`, der Teilmengenbeziehungen wiedergibt: Zu jeder Teilmenge der l Variablen, die für die erweiterte Trägernutzung betrachtet werden – also zu jedem möglichen Träger –, speichern wir eine Liste ihrer Teilmengen (natürlich als Bitvektoren). So kann man z.B. bei der Reduktion eines Monoms sofort ablesen, in welchen Listen reduzierende Binome stecken können – nämlich in denen, die mit einer Teilmenge seines Trägers indiziert sind.

Die Implementierung der Idealklasse verteilt sich auf drei Dateien:

1. `ideal.cc` enthält die Konstruktoren, die Ausgabe- und viele Hilfsroutinen. Die Konstruktoren für die Idealklasse sind nicht trivial: Der wichtigste erhält als Argument eine Nummer, die einen der Algorithmen aus Kapitel 3 codiert (als Merkhilfe erhalten die Nummern in `globals.h` vielsagende Namen wie `CONTI_TRAVERSO`). Dann baut er aus einer Matrix, die ebenfalls als Argument übergeben wird, das Ideal auf, mit dem der codierte Algorithmus seine Berechnungen startet.
2. In `Buchberger.cc` kann man verschiedene Varianten des Buchberger-Algorithmus nachlesen, die Abschnit 5.2.2 noch genauer erklärt. Ferner implementiert diese Datei die Reduktion eines Binoms durch das gegebene Ideal.
3. In der Datei `ideal_stuff.cc` findet man alle Sonderfunktionen, die die Optimierungsalgorithmen benötigen. Dazu gehören Elimination, Pseudoelimination, Wechsel von Termordnungen, Änderungen der Variablenreihenfolge und der Variablenflip des Algorithmus von Di Biase und Urbanke. Die meisten dieser Funktionen iterieren über die Erzeugerliste(n) und rufen entsprechende Funktionen der Binomklasse auf.

5.2 Die Algorithmen

5.2.1 Der LLL-Algorithmus

Beide Varianten 1.56 und 1.57 des LLL-Algorithmus finden in unser Programmpaket Eingang. Die Deklaration der beiden Hauptroutinen, die zu keiner Klasse gehören, steht in der Datei `LLL.h`, deren Implementierung samt Hilfsprozeduren in `LLL.cc`.

Für die Berechnungen des LLL-Algorithmus erweisen sich die in C++ eingebauten Typen für ganze Zahlen als zu beschränkt. Die Ergebnisvektoren haben zwar im allgemeinen recht kleine Koeffizienten, aber einige Hilfsparameter wie die $\lambda_{k,i}$ wachsen im Laufe der Berechnungen sehr schnell. Experimente zeigen, daß schon bei kleinen Instanzen – etwa 5×8-Matrizen mit Einträgen zwischen -10 und 10 – ein Überlauf sehr wahrscheinlich ist.

Man braucht also einen Datentyp für große ganze Zahlen.Unsere Programme benutzen standardmäßig die **GNU multiple precision arithmetic library (GMP)**, die auf vielen UNIX- und LINUX-Systemen installiert ist; für eine einfachere Handhabung haben wir die Funktionen in eine Klasse `class BigInt` verpackt, deren Deklaration und Implementierung man in den Dateien `BigInt.h` und `BigInt.cc` nachlesen kann (Dank an Hans Schönemann für seine Vorlage). Alternativ bietet die **LEDA**-Bibliothek ([21]) eine einfach zu benutzende Klasse für große ganze Zahlen.

Wer andere Implementierungen benutzen will, muß die Dateien `globals.h`, `ideal.cc` und `LLL.cc` entsprechend ändern. Falls überhaupt kein Datentyp für große ganze Zahlen zur Verfügung steht, kann man die Zeile `#define GMP` aus `globals.h` löschen und neu compilieren. Dann benutzt das Programm den C-Typ `long`, der immerhin ein Notbehelf für winzige Instanzen ist.

Effizienzverluste durch die Verwendung großer ganzer Zahlen braucht man übrigens nicht zu befürchten, da diese im Buchberger-Algorithmus nicht mehr gebraucht werden.

5.2.2 Der Buchberger-Algorithmus

Die Routinen des Buchberger-Algorithmus sind in `ideal.h` deklariert und in `Buchberger.cc` implementiert. Ein Aufruf des Buchberger-Algorithmus erfolgt über die Funktion `reduced_Groebner_basis`, die zur Idealklasse gehört und drei optionale Argumente hat: die Variante des Algorithmus, die zu benutzenden S-Paar-Kriterien und die Zwischenreduktionsquote.

Varianten des Algorithmus

Der Buchberger-Algorithmus ist, wie bereits in Kapitel 1 gesehen, kein starres Schema; er läßt ungemein viele Freiheiten hinsichtlich der Reihenfolge, in der seine Operationen ausgeführt werden. Unsere Implementierung testet vier Varianten, die alle in zwei Phasen arbeiten.

Die erste Phase gehört der Berechnung und Reduktion von S-Binomen; dabei werden stets alle noch nicht betrachteten Paare von Idealerzeugern abgearbeitet. Die zweite Phase führt eine Zwischenreduktion durch, falls die erste Phase genügend neue Erzeuger produziert hat (siehe unten).

Man wiederholt beide Phasen solange, bis die S-Paar-Berechnung keine neuen Erzeuger mehr produziert. Die Varianten unterscheiden sich in der Organisation der S-Paar-Berechnung und -Reduktion.

Version 1 berechnet zunächst die S-Polynome aller Paare von Erzeugern, deren `done`-Komponenten in der Liste nicht beide den Wert `TRUE` haben, und speichert sie in einer Hilfsliste. Danach setzt Version 1 die `done`-Komponenten aller beteiligten Erzeuger auf `TRUE`, damit bei der nächsten S-Paar-Berechnung kein Paar ein zweites Mal betrachtet wird.

Anschließend erfolgt die polynomweise Reduktion der S-Binome in der Hilfsliste durch die Idealerzeuger: Ein Rest, der nicht Null ist, wird unmittelbar nach seiner Berechnung in die Erzeugerliste(n) eingefügt, nicht etwa erst nach der Reduktion aller S-Polynome. Auf diese Weise können zu reduzierende S-Polynome von bereits reduzierten profitieren. Die `done`-Komponente jedes neuen Erzeugers wird auf `FALSE` gesetzt, um ihn an der nächsten S-Paar-Berechnung zu beteiligen.

Version 1a unterscheidet sich von Version 1 nur dadurch, daß die S-Paare geordnet nach wachsendem Initialterm in die Hilfsliste eingefügt werden. Bei der Reduktion werden S-Binome mit kleinem Initialterm dann zuerst betrachtet – in der Hoffnung, daß deren Reste im folgenden besonders viele andere S-Binome reduzieren.

Version 2 gestaltet die Bildung und Reduktion der S-Paare viel dynamischer: Sie reduziert jedes S-Polynom unmittelbar nach seiner Berechnung durch die Idealerzeuger; ist der Rest ungleich Null, wird er sofort dazu benutzt, um umgekehrt die Binome in den Erzeugerlisten zu reduzieren. Dadurch fallen eventuell Erzeuger weg, bevor unnötige S-Paare mit ihnen gebildet werden. Ein Teil der Zwischenreduktion erledigt sich also schon während der S-Paar-Berechnung, so daß man im Durchschnitt auf weniger Erzeuger und weniger S-Paare hoffen darf.

Auch Version 2 sammelt die S-Paar-Reste getrennt von den alten Erzeugern in einer Hilfsliste, um die `done`-Komponenten bequem verwalten zu können. Wenn man will, daß wie in den Versionen 1 und 1a zu reduzierende S-Polynome von bereits reduzierten profitieren, kann man das zum Beispiel über gelegentliche Zwischenreduktionen der Hilfsliste erreichen.

Version 3 unterscheidet sich von Version 2 dadurch, daß die Hilfsliste für die S-Paar-Reste im Rahmen der erweiterten Trägernutzung in mehrere Listen aufgespalten wird; auf diese Weise kann man die Reste effizienter untereinander reduzieren.

Die `head_reduced`-Komponenten der Listenelemente erfüllen einen ähnlichen Zweck für die Zwischenreduktion wie die `done`-Komponenten für die S-Paar-Bildung.

S-Paar-Kriterien

Über das zweite Argument der Funktion `reduced_Groebner_basis` steuert man die Benutzung der S-Paar-Kriterien 1.27 bis 1.29. Da dieses Argument den Typ `short` hat, muß man die S-Paar-Kriterien entsprechend codieren. Dazu ist jedem Kriterium eine Zweierpotenz zugeordnet, und zwar:

Teilerfremde Initialterme:	1
Kriterium M:	2
Kriterium F:	4
Kriterium B:	8
Zweites-Buchberger-Kriterium:	16

Man übergibt der Funktion dann die Summe der Zahlen, die den gewünschten Kriterien entsprechen (binäre Codierung der Kombination). So steht der Default-Wert $11_{10} = 01011_2$ für die Kombination der Kriterien teilerfremde Initialterme, M und B. Man kann die Zahl 11 auch durch den Text `REL_PRIMENESS`

`+ M_CRITERION + B_CRITERION` ersetzen, denn die Zweierpotenzen werden in `globals.h` wieder mit schönen Namen versehen.

Zwischenreduktion

Das dritte Argument der Funktion `reduced_Groebner_basis` entscheidet, wie oft der Buchberger-Algorithmus Zwischenreduktionen durchführt. Genauer gibt es den Anteil in Prozent an, den neue Erzeuger an der Gesamtzahl der Erzeuger ausmachen müssen, damit solch eine Reduktion stattfindet. Als neue Erzeuger bezeichnen wir dabei solche, die seit der letzten Zwischenreduktion hinzugekommen sind.

Der Default-Wert des Arguments ist 12, für 12%. Der Wert 0 verursacht eine Zwischenreduktion nach jeder Phase der S-Paar-Berechnung. Um Zwischenreduktionen zu verbieten, übergibt man (nicht ganz der Intuition entsprechend) einen negativen Wert.

Beim Aufruf des Buchberger-Algorithmus speichern wir aus Effizienzgründen sowohl die S-Paar-Kriterien als auch die Zwischenreduktionsquote in der Idealstruktur; auf diese Weise müssen wir sie nicht als Argumente an Subroutinen weitergeben.

5.2.3 Die IP-Algorithmen

In den Dateien `IP_algorithms.h` und `IP_algorithms.cc` findet man die Implementierung der Algorithmen zur ganzzahligen Optimierung aus Kapitel 3. Um eine flexible Benutzung zu ermöglichen, führen wir die Berechnung des torischen Ideals und die Reduktion in verschiedenen Funktionen durch und speichern alle Ergebnisse in Dateien. So kann man einmal berechnete Ideale für spätere Reduktionen nutzen und auch die Kostenfunktion problemlos variieren. Die implementierten Funktionen sind nicht in eine Klasse eingebettet.

Berechnung torischer Ideale

Zur Berechnung torischer Ideale hat man die Wahl zwischen den fünf vorgestellten Algorithmen. Der Conti-Traverso-Algorithmus ist gleich in drei Varianten implementiert: In der allgemeinen Form 3.7, der Variante für den nichtnegativen Fall 3.2 sowie einer Variante mit Elimination, die im Gegensatz zu 3.5 für allgemeine Matrizen geeignet ist. Die Implementierungen der Algorithmen von Pottier, von Hoşten und Sturmfels sowie von Bigatti, La Scala und Robbiano nutzen die Verbesserungen des Hoşten-Shapiro-Verfahrens. Der Algorithmus von Di Biase und Urbanke kümmert sich selbst um die Berechnung einer geeigneten Gitterbasis für den Kern der Eingabematrix. Die positiven Graduierungen für die Algorithmen von Hoşten und Sturmfels sowie von Bigatti, La Scala und Robbiano besorgt unsere Implementierung hingegen (noch) nicht, weil dazu die Programmierung des Simplex-Algorithmus, also eine Menge Aufwand, nötig wäre.

Als Argument erhalten alle Algorithmen den *Namen* einer Datei, die die Eingabematrix mitsamt Kostenfunktion enthalten sollte. Auf diese Weise können die Funktionen ihre Ausgabedatei nach der Eingabedatei benennen. Wie die

Benennung genau funktioniert und welches Format die Dateien haben, erfährt man in `IP_algorithms.h` oder in der Hilfe zu den ausführbaren Programmen (siehe Abschnitt 5.3).

Optional kann man den Algorithmen weitere Argumente mitgeben, z.B. das Argument `verbose`. Mit dem Wert `TRUE` bewirkt dieses, daß sowohl die Optionen für den Buchberger-Algorithmus als auch Compiliereinstellungen, z.B.zur Trägernutzung, in die Ausgabedatei geschrieben werden – eine nützliche Sache für Testreihen.

Die Implementierung aller Algorithmen gliedert sich in drei Teile:

1. Ein **Eingabeteil** liest die Eingabedatei und prüft ihr Format.
2. Ein **Berechnungsteil** bestimmt mit dem jeweiligen Algorithmus das torische Ideal zur Eingabematrix, und zwar in Form seiner reduzierten Gröbnerbasis bezüglich einer Ordnung, die den gegebenen Kostenvektor verfeinert.
3. Ein **Ausgabeteil** schreibt die Ergebnisse in geeignetem Format in die Ausgabedatei.

Ein- und Ausgabeteil sind größtenteils trivial, stellen aber den überwiegenden Anteil am Code. Die eigentlichen Berechnungen der Algorithmen geschehen übersichtlich in wenigen Zeilen, weil die Funktionen der Idealklasse entsprechend darauf zugeschnitten wurden.

Lösen ganzzahliger Optimierungsprobleme

Um eine IP-Instanz lösen zu können, fehlt uns noch der Reduktionsteil der Algorithmen: Die Funktion `solve` erhält als Argumente die Namen zweier Dateien, von denen die zweite die zu reduzierenden Vektoren enthalten sollte, die erste die Idealerzeuger, durch die reduziert wird. Will man also die Instanzen

$$\min\{c^\intercal X \mid AX = b_k,\ X \geq 0,\ X \in \mathbb{Z}^n\},\ k = 1, \ldots, s,$$

lösen, so wählt man als erste Eingabedatei die Ausgabedatei eines der obigen Algorithmen bei Aufruf mit der Matrix A und der Kostenfunktion c. In die zweite Datei schreibt man je nach benutztem Algorithmus die rechten Seiten b_k oder passende Initiallösungen.

Der Eingabeteil der `solve`-Funktion überprüft soweit möglich die logische Konsistenz der Eingabedateien. Bei den Algorithmen, die Initiallösungen benötigen, muß z.B. die Dimension der Vektoren in der ersten Datei mit der Variablenzahl des Ideals in der zweiten Datei übereinstimmen.

Bei erfolgreicher Reduktion speichert `solve` die Ergebnisse in einer Datei, die nach der ersten Eingabedatei benannt ist.

Zur Variation der Termordnung bei gleichbleibender Matrix dient schließlich die Funktion `change_cost`. Deren Eingabedateien enthalten das bereits bekannte torische Ideal zur Matrix bzw. den neuen Kostenvektor. Die Funktion benutzt keinen speziellen Algorithmus zur Umrechnung von Gröbnerbasen, sondern startet einfach den Buchberger-Algorithmus bezüglich der neuen Termordnung auf den alten Idealerzeugern.

Zusammenfassend zeigt das nachstehende Bild die Beziehungen zwischen allen Dateien des Programmpakets. In eckigen Boxen stehen die Headerdateien (`.h`-Dateien) mit den Deklarationen, in runden die `.cc`-Dateien mit der eigentlichen Implementierung. Ein Pfeil von Datei 1 zu Datei 2 bedeutet, daß Datei 2 Datei 1 mit `#include` einbindet. Gestrichelte Pfeile zeigen von den `.h`-Dateien zu ihren Implementierungen. Die gepunkteten Pfeile gehen zu Dateien, die `main()`-Funktionen enthalten, aus denen also die ausführbaren Programme entstehen; diese werden im folgenden Abschnitt noch genauer erläutert.

Die Binomklasse und die Klasse der monomialen Ordnungen brauchen sich gegenseitig. Da sich die Dateien `binomial.h` und `term_ordering.h` nicht gegenseitig mit `#include` einbinden können, binden wir beide in eine gemeinsame Headerdatei `binomial__term_ordering.h` ein, die ihrereseits von `binomial.cc` und `term_ordering.cc` eingebunden wird. Für Details sei auf den Quellcode in Anhang A verwiesen.

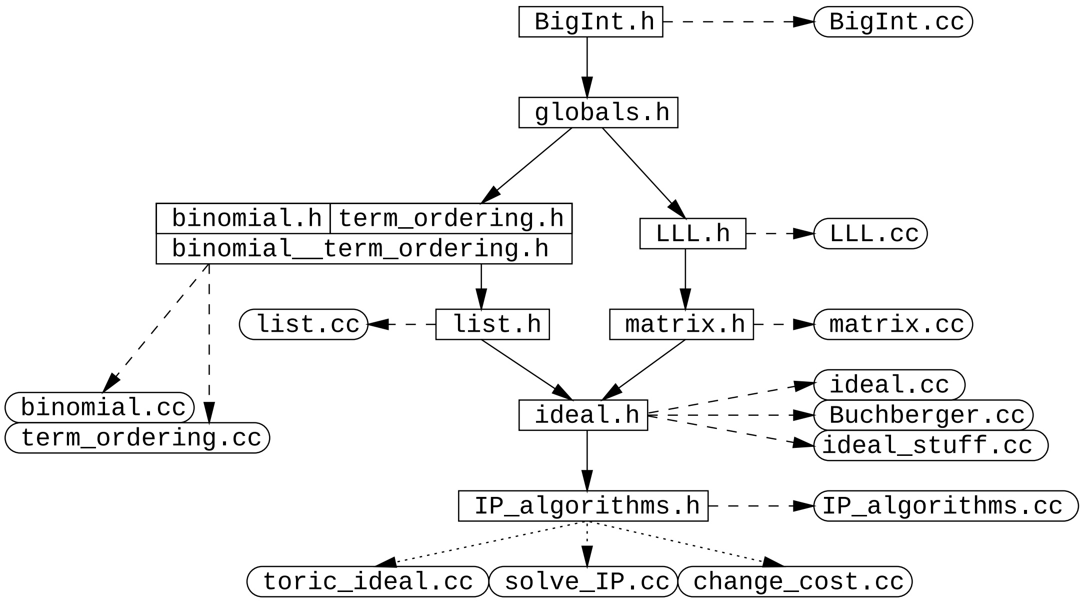


Abbildung 5.2: Struktur des Programmpakets

5.3 Benutzung des Programms

5.3.1 Eine Kommandozeilenversion

Die vorgestellte Implementierung gliedert sich in drei ausführbare Programme:

1. `toric_ideal` zur Berechnung des torischen Ideals zu einer Matrix. Den Quellcode findet man in `toric_ideal.cc`.

2. `change_cost` zur Änderung des Kostenvektors einer IP-Instanz bzw. der monomialen Ordnung für ein bekanntes torisches Ideal. Den Quellcode findet man in `change_cost.cc`.

3. `solve_IP` zur ganzzahligen Optimierung. Es akzeptiert als Eingabe entweder die Matrix des Optimierungsproblems oder deren torisches Ideal. Den Quellcode findet man in `solve_IP.cc`.

Die Programme wurden für **UNIX-ähnliche Betriebssysteme** geschrieben und mit verschiedenen Versionen des **GNU C++ Compilers g++** compiliert. Das Makefile in Anhang A.3 macht es einfach, die Programme auf einer solchen Plattform zu installieren. Dazu kopiert man Quellcode (`.cc`- und `.h`-Dateien), Makefile und Hilfe-Dateien (`.hlp`) in ein Verzeichnis und gibt

`make <Programmname>` bzw. `make all`

ein, um einzelne Programme bzw. das gesamte Paket zu compilieren. Das Makefile geht davon aus, daß die GMP-Bibliothek, siehe Abschnitt 5.2.1, auf dem Rechner zur Verfügung steht! Anschließend erhält man durch Eingabe von

`<Programmname> --help`

einen Hilfetext zur Benutzung des gewünschten Programms; dabei wird der UNIX-Befehl `less` aufgerufen, d.h. man verläßt die Hilfe durch Eingabe von `q`. Neben dem Eingabeformat werden im Hilfetext auch die Programmoptionen erläutert. Beim Aufruf von `toric_ideal` ist z.B. die Angabe eines Algorithmus zwingend, ferner kann man Optionen für den Buchberger-Algorithmus setzen.

5.3.2 Die Anbindung an SINGULAR

Für das Programmpaket ist eine Anbindung an das Computeralgebrasystem SINGULAR ([28]) geplant. Die vorläufige Version der Anbindung besteht aus zwei Bibliotheken `toric.lib` und `IP.lib`. Erstere enthält die Funktionen der Programme `toric_ideal` und `change_cost`, letztere die von `solve_IP`. Hilfe zur Benutzerschnittstelle erhält man durch Eingabe von

`help toric.lib;` bzw. `help IP.lib;`

Die Benutzung der Programme innerhalb von SINGULAR ist einfacher als die der Kommandozeilenversion, da die Oberfläche dieses Systems zur Verfügung steht. Insbesondere muß man sich nicht um geeignet formatierte Eingabedateien bemühen, sondern kann die Funktionen direkt mit den passenden SINGULAR-Datentypen aufrufen (z.B. `matrix` und `ideal`). Die Ausgabe erfolgt ebenfalls nicht über Dateien; stattdessen geben die Funktionen SINGULAR-Vektoren (`intvec`), Listen von Vektoren (`list`) oder Ideale (`ideal`) zurück. Allerdings hat der Benutzer im Moment noch keine Möglichkeit, wie in der Kommandozeilenversion Optionen für den Buchberger-Algorithmus zu setzen; solche Optionen müßten nämlich als Funktionsargumente übergeben und somit geeignet codiert werden. Eventuell wird die Anpassung in Absprache mit den SINGULAR-Entwicklern noch in diese Richtung erweitert.

5.4 Alternative Ideen zur Implementierung

Im Zusammenhang mit dem Buchberger-Algorithmus haben wir bereits darauf hingewiesen, daß es viele Programmiermöglichkeiten gibt. Wir erwähnen nun einige Punkte, in denen noch Verbesserungspotential für unsere Implementierung stecken könnte:

Speicherverwaltung

Im Laufe des Buchberger-Algorithmus kommen durch S-Paar-Bildung ständig neue Binome hinzu, andere verschwinden durch Reduktion. Da man die Größe einer Gröbnerbasis im voraus nicht kennt, muß man den benötigten Speicherplatz dynamisch verwalten, d.h. für neue Binome explizit Speicher anfordern und ihn später wieder freigeben. Das kostet natürlich Zeit, die nichts zur Berechnung beiträgt. Mike Stillman schlug folgende Strategie vor, um die Anzahl der Speicheranforderungen und -freigaben zu verringern:

> Der Speicherplatz von Binomen, die durch Reduktion verschwinden, wird nicht freigegeben, sondern auf einen **Stapel (stack)** gelegt. Benötigt man nun Speicherplatz für ein neues Binom, so nimmt man ihn von diesem Stapel, falls er nicht leer ist.

Diese Strategie wurde versuchsweise in einer frühen Programmversion implementiert. Der Laufzeitvergleich zeigte für die getesteten Beispiele nur geringfügige Abweichungen im Rahmen von etwa 2% nach oben wie nach unten. Dafür kommen zwei Gründe in Frage:

- Die Verwaltung des Stapels braucht in etwa soviel Zeit wie die dynamische Speicherverwaltung.
- Die Erzeugendensysteme der torischen Ideale sind am Ende des Buchberger-Algorithmus in der Regel viel größer als zu Beginn, so daß die Anzahl der Anforderungen neuen Speichers die Zahl der Freigaben alten Speichers weit übersteigt.

Dennoch ist die Idee es wert, weiterhin untersucht zu werden: Vielleicht zahlt sie sich bei regelmäßigen Matrizen aus, deren Erzeugendensysteme während des Buchberger-Algorithmus nicht so stark wachsen; oder es gelingt, durch eine geschicktere Implementierung des Stapels die Verwaltungskosten deutlich zu reduzieren.

Trägernutzung

Die Ergebnisse in Kapitel 6 stellen ihren Nutzen außer Frage. Nur beschränkt unsere Implementierung die Anzahl der betrachteten Variablen auf die Anzahl der Bits einer `long`-Zahl; ob eine höhere oder auch geringere Anzahl vorteilhaft wäre, bleibt ungeklärt (vgl. Bemerkung 4.13). Ebenso das schwer zu formalisierende Problem, welche Variablen in Abhängigkeit vom Ideal und der monomialen Ordnung in den Träger eingehen sollten.

Variablenreihenfolge

Die gewichteten revers-lexikographischen Ordnungen des Algorithmus von Hoşten und Sturmfels betrachten in jedem Saturierungsschritt eine andere Variablenreihenfolge. Dabei ist für die Korrektheit nur wichtig, welche Variable zuerst zum Monomvergleich betrachtet wird – es bleibt also eine ganze Reihe von Möglichkeiten. Unsere Implementierung realisiert die Änderung der Reihenfolge durch Variablentausch: Die Variable, nach der saturiert werden soll, wird mit der letzten Variable vertauscht, und nach der Saturierung wird die Vertauschung wieder rückgängig gemacht.

Das Rücktauschen läßt sich vermeiden, wenn man sich die Permutation merkt, mit der man die Variablen wieder in die ursprüngliche Reihenfolge bringt. Es ist aber auch nicht kritisch für die Effizienz. Suspekt ist der Variablentausch jedoch in folgender Hinsicht: Tauscht man z.B. x_1 und x_n, so wird die vorher wichtigste Variable (d.h. die zuerst betrachtete) plötzlich zur unwichtigsten und umgekehrt.

Die in den Zwischenschritten berechneten Gröbnerbasen unterscheiden sich vermutlich weniger, wenn man derartige Sprünge vermeidet. Gesucht ist also eine Strategie, die uns zu einer besseren Variablenreihenfolge verhilft.

Bemerkung 5.2 Um die Variablenreihenfolge zu ändern, hat man auf der Ebene der Implementierung zwei Möglichkeiten:

1. Man permutiert die Vektorkomponenten der Idealerzeuger und die Gewichte der monomialen Ordnung entsprechend. Dies tut unsere Implementierung.

2. Man speichert mit der Termordnung eine Permutation, die angibt, in welcher Reihenfolge die Variablen zu betrachten sind.

Die zweite Vorgehensweise sieht eleganter aus, macht aber Monomvergleiche langsamer: Man muß die Variablen nämlich indirekt über die Permutation ansprechen, d.h in der Form `variable[permuation[i]]` statt in der Form `variable[i]`. Dies geht zu Lasten des zeitkritischen Buchberger-Algorithmus, wohingegen Permutationen von Vektorkomponenten nur zwischen zwei Saturierungsschritten, also nicht während der Gröbnerbasisberechnung, durchgeführt werden. □

Termordnungen

Das Problem der Variablenreihenfolge läßt sich auf die Frage verallgemeinern, welche Termordnung an welcher Stelle angebracht ist. Der Algorithmus von Di Biase und Urbanke braucht beispielsweise Eliminationsordnungen bezüglich wechselnden Variablen für seinen Variablenflip. Wie die Ordnungen auf den Nicht-Eliminationsvariablen aussehen, ist aus theoretischer Sicht egal, nicht aber aus praktischer. Für die Eliminationsvariablen des Conti-Traverso-Algorithmus erweist sich etwa die verrufene lexikographische Ordnung als günstig. Die Resultate von Versuchen mit unserer Implementierung decken sich in diesem Punkt mit denen in [7].

Gradweises Vorgehen

Unsere Implementierung des Buchberger-Algorithmus hält dazu lediglich eine Variante bereit, in der die S-Paare gradweise abgearbeitet werden. Bigatti, La Scala und Robbiano empfehlen in [3] eine viel dynamischere Strategie, um möglichst schnell Erzeuger mit möglichst kleinem Grad zu finden: Der Buchberger-Algorithmus läuft dort nicht in Phasen, sondern unterbricht die Berechnungsschleifen, um Polynome kleinen Grades bevorzugt zu behandeln.

Neben der Menge der S-Binome wird dann auch die Erzeugermenge gradweise abgearbeitet. Leider beißt sich diese Vorgehensweise mit der erweiterten Trägernutzung: Man kann die Erzeuger nicht gleichzeitig nach dem Grad und nach dem Träger der Initialmonome ordnen.

Bringt eine stärkere Orientierung des Buchberger-Algorithmus am Grad der Erzeuger Vorteile, die die der erweiterten Trägernutzung übertreffen? Wer diese Frage beantworten will, sollte bei einer Implementierung jedenfalls daran denken,

- den Grad eines Binoms explizit zu speichern, denn dieser wird bei gradweiser Abarbeitung der Erzeuger öfters abgefragt. Seine Berechnung kostet Zeit proportional zur Anzahl der Variablen – zu viel, um ihn bei jeder Abfrage neu zu bestimmen.

- die Listen durch eine adäquatere Struktur zu ersetzen. Denn man braucht Zeit proportional zur Anzahl der Elemente, um ein Binom an der richtigen Stelle einer geordneten Liste einzufügen (vgl. 5.1.4). Mit geeigneten Baumstrukturen oder sogenannten **Prioritätswarteschlangen** (vgl. [8]) benötigt Einfügen und Löschen nur logarithmische Zeit.

Parallelisierung

Abschnitt 3.2 in [3] stellt eine parallele Version des Algorithmus von Hoşten und Sturmfels vor, PATI genannt – **p**aralleler **A**lgorithmus für **t**orische **I**deale. Dieser geht davon aus, daß ein Computer mit n Prozessoren zur Verfügung steht, und versucht, die sukzessiven Saturierungsschritte gleichzeitig durchzuführen. Die Grundidee ist, jeden Prozessor das Anfangsideal nach einer der n Variablen saturieren zu lassen. Damit bei allen Prozessoren dasselbe und vor allem richtige Ergebnis herauskommt, müssen diese während der Berechnung natürlich Teilergebnisse austauschen.

Die Autoren betrachten damit das Problem der Berechnung torischer Ideale erstmals auf der Ebene des Maschinenmodells. Und das zugrundeliegende parallele Modell bietet sich für mehr an: Die S-Paar-Berechnung beim Buchberger-Algorithmus etwa ließe sich wunderbar parallelisieren.

Ferner kann der Versuch der Parallelisierung neue Ideen für sequentielle Algorithmen liefern: Übersetzt man PATI nämlich auf das gewöhnliche sequentielle Maschinenmodell (mit nur einem Prozessor) zurück, so unterscheidet er sich in einem Punkt vom Algorithmus von Hoşten und Sturmfels: Dessen brave Schleife in Schritt (3) wird durch ein Hin- und Herwechseln zwischen den verschiedenen Termordnungen bzw. Variablenreihenfolgen ersetzt.

Randomisierung

Und wenn wir schon mal bei scheinbar ungeordneten Organisationsformen angelangt sind: Wer wagt es auszuschließen, daß damit der Buchberger-Algorithmus besonders gut funktioniert? Wenn man nicht weiß, in welcher Reihenfolge man seine Operationen ausführen soll, kann man das ja vielleicht dem Zufall überlasssen. Die Idee der Randomisierung hat sich in der Praxis jedenfalls schon häufig bewährt.

Kapitel 6

Praktische Experimente

6.1 Auswahl des Testmaterials

Wir testen unsere Implementierung an ganzzahligen Optimierungsproblemen in Standardform (2.3). Eine Testinstanz setzt sich zusammen aus einer Matrix A, einem Kostenvektor c und einem Begrenzungsvektor b bzw. einer dazu passenden Initiallösung. Man muß dafür folgendes wählen:

1. die Dimension von A, d.h. die Anzahl der Zeilen und Spalten;
2. die Koeffizienten von A aus dem Bereich der ganzen Zahlen;
3. die Koeffizienten von c aus dem Bereich der nichtnegativen reellen Zahlen;
4. die Koeffizienten von b aus dem Bereich der ganzen Zahlen bzw. die der Initiallösung aus den nichtnegativen ganzen Zahlen.

Zunächst möchten wir zufällige Instanzen betrachten. Nun ist die Wahl zufälliger Elemente aus unendlichen Mengen wie $\mathbb{N}$, $\mathbb{Z}$ oder $\mathbb{R}_{\geq 0}$ technisch problematisch – man denke nur daran, daß der Rechner diese Mengen gar nicht darstellen kann. Die meisten Systeme stellen Funktionen bereit, die Pseudozufallszahlen innerhalb des darstellbaren Bereichs für ganze Zahlen liefern. Deshalb beschränken wir uns bei der Wahl der Koeffizienten von A, b und c auf ganze Zahlen innerhalb eines begrenzten Bereichs. Das ist zum Testen nicht nur ausreichend, sondern auch wünschenswert: Denn die Koeffizientengröße von A beeinflußt stark die Laufzeit unserer Algorithmen (vergleiche Seite 106), und die Testinstanzen sollen schließlich praktikabel sein. Aus gleichem Grund überlassen wir auch die Matrixdimension nicht dem Zufall.

Da viele kombinatorische Probleme zu regelmäßigen Matrizen führen, geben wir uns nicht mit zufällig gewähltem Testmaterial zufrieden und berücksichtigen in unserer Testreihe das Transportproblem.

Bei der Interpretation der Testdaten sollte man vorsichtig bleiben; denn nicht alles, was verglichen wird, ist in jeder Hinsicht vergleichbar. So ist es nicht ganz fair, einen Algorithmus, der eine Initiallösung benötigt, mit einem Algorithmus zu vergleichen, der ohne diese auskommt; ebenso verhält es sich mit Algorithmen, die unterschiedliche Voraussetzungen an die Eingabematrix machen.

6.2 Konfiguration des Buchberger-Algorithmus

Wir suchen über eine Testreihe eine gute Kombination von Optionen für den Buchberger-Algorithmus. Bei der Compilierung können wir wählen

- zwischen einfach und doppelt verketteten Listen;
- zwischen dem Verzicht auf Trägernutzung, der Trägernutzung auf Ebene der Binome und der erweiterten Trägernutzung;
- zwischen vorderen und hinteren Variablen als Trägervariablen (bei Trägernutzung);
- die Anzahl der Erzeugerlisten (bei erweiterter Trägernutzung);
- zwischen der Verwendung von `short`, `int` und `long` als Typ für ganze Zahlen.

Zusätzlich können wir zur Laufzeit regeln,

- welche der fünf S-Paar-Kriterien verwendet werden,
- wie häufig Zwischenreduktionen stattfinden,
- welche der implementierten Versionen des Buchberger-Algorithmus zur Anwendung kommt.

Zusammen gibt dies mehr Kombinationsmöglichkeiten, als wir testen können. Deshalb gehen wir von einer Kombination aus, die sich im Rahmen von Korrektheitstests während der Implementierung als gut herausgestellt hat, und zeigen durch Variation einzelner Einstellungen den Einfluß der Optionen auf das Laufzeitverhalten. Fest bleibt dabei die Verwendung des Typs `long` für ganze Zahlen, da der Algorithmus von Di Biase und Urbanke bei Verwendung von `short` bei einigen Testinstanzen einen Überlauf verursacht.

Im folgenden bezeichne $r(m \times n, l, u)$ eine $m \times n$-Matrix mit zufällig gewählten ganzzahligen Koeffizienten im Bereich von l bis u. $t(m, n)$ sei die Matrix des $m \times n$-Transportproblems, also eine $(m+n) \times (m \cdot n)$-Matrix. Die Koeffizienten des Kostenvektors zur Testmatrix $r(m \times n, l, u)$ werden zufällig aus dem Bereich von $\{0, \dots, u\} \subset \mathbb{N}$ gezogen. Bei den Instanzen des Transportproblems stammen sie aus $\{0, \dots, 10\}$, nur bei der mit $_{50}$ indizierten Instanz aus $\{0, \dots, 50\}$.

Die nachstehende Tabelle zeigt die Laufzeiten einiger Testinstanzen in der **Standardkonfiguration** des Buchberger-Algorithmus mit

- einfach verketteten Listen;
- erweiterter Trägernutzung mit vorderen Variablen als Trägervariablen und 2^8 Erzeugerlisten;
- den S-Paar-Kriterien M, B und teilerfremde Initialterme;
- einer Zwischenreduktionsquote von 12%.

Alle in diesem Kapitel aufgeführten Laufzeiten sind in Sekunden angegeben.

	$\mathcal{GB}$	ct	pct	ect	pt	hs	du	blr
$r(1 \times 11, 0, 1)$	10 (12,11)	0.00	0.00	0.01	0.00		0.01	
$r(1 \times 11, 0, 5)$	17 (23,18)	0.01	0.00	0.01	0.00	0.01	0.02	0.01
$r(1 \times 11, -5, 5)$	11 (13)	0.00		0.01	0.00		0.01	
$r(1 \times 11, 0, 15)$	46 (56,56)	0.03	0.03	0.03	0.02		0.04	
$r(1 \times 11, -15, 15)$	31 (33)	0.01		0.01	0.02		0.03	
$r(3 \times 9, 0, 1)$	6 (10,9)	0.00	0.00	0.00	0.00	0.00	0.01	0.00
$r(3 \times 9, 0, 5)$	163 (311,433)	1.18	2.85	1.18	1.07	0.59	0.67	0.65
$r(3 \times 9, -5, 5)$	208 (212)	0.55		0.56	0.50		0.62	
$r(3 \times 9, 0, 15)$	278 (595,831)	4.62	14.95	4.62	3.75	2.07	2.22	2.77
$r(3 \times 9, -15, 15)$	395 (399)	2.60		2.61	3.01		3.02	
$r(5 \times 7, 0, 1)$	2 (8,12)	0.00	0.00	0.00	0.00			
$r(5 \times 7, 0, 5)$	14 (325,563)	1.73	5.98	1.73	0.01	0.01	0.02	0.01
$r(5 \times 7, -5, 5)$	19 (473)	3.55		3.55	0.02		0.02	
$r(5 \times 7, 0, 15)$	9 (300,1617)	2.07	75.91	2.07	0.00	0.00	0.02	0.03
$r(5 \times 7, -15, 15)$	6 (1821)	181.39		181.32	0.00		0.03	
$t(3, 4)$	18 (28,60)	0.02	0.02	0.01	0.00	0.02	0.01	0.01
$t(5, 7)$	239 (421,449)	16.37	6.91	16.36	139.44	4.60	3.01	26.97
$t(5, 7)_{50}$	230 (383,440)	12.37	6.57	12.37	143.31	4.57	2.99	26.90

Tabelle 6.1: Laufzeiten der Standardkonfiguration des Buchberger-Algorithmus

Die erste Spalte gibt Dimension und Koeffizientengröße der Testmatrizen an; die konkreten Matrizen samt Kostenvektor findet man in Anhang B, ebenso die reduzierten Gröbnerbasen von einigen der zugehörigen torischen Ideale. Die zweite Spalte verrät die Anzahl der Elemente dieser Gröbnerbasen; die Zahlen in Klammern geben die Größe der erweiterten Gröbnerbasen an, die der Conti-Traverso-Algorithmus und gegebenenfalls seine nichtnegative Variante berechnen.

Die anderen Spalten sind mit dem verwendeten Algorithmus markiert und enthalten dessen Laufzeiten zur Berechnung der torischen Ideale. Dabei messen wir nur die Laufzeit der eigentlichen Berechnungen, nicht die für Ein- und Ausgabe. Der Wert 0.00 bedeutet eine Laufzeit von weniger als $\frac{1}{100}$ Sekunden. Hier die Abkürzungen für die Algorithmen:

ct	steht für den Algorithmus von Conti und Traverso,
pct	für dessen nichtnegative Variante,
ect	für die Variante mit Elimination,
pt	für den Algorithmus von Pottier,
hs	für den Algorithmus von Hoşten und Sturmfels,
du	für den Algorithmus von Di Biase und Urbanke,
blr	für den Algorithmus von Bigatti, La Scala und Robbiano.

Fehlende Einträge bedeuten, daß der Algorithmus nicht für die betreffende Instanz geeignet ist – wie etwa die nichtnegative Version des Algorithmus von Conti und Traverso für Instanzen mit negativen Komponenten oder der Algo-

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	16.95	52.90	16.95	14.10	4.67	6.16	7.83
$r(3 \times 9, -15, 15)$	7.57		7.57	7.58		8.03	
$r(5 \times 7, 0, 15)$	11.98	617.50	11.97	0.00	0.00	0.03	0.03
$r(5 \times 7, -15, 15)$	A		A	0.01		0.03	
$t(5, 7)$	90.38	40.56	90.42	A	26.42	16.51	96.78

Tabelle 6.2: Laufzeiten ohne Benutzung von S-Paar-Kriterien

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	14.90	40.86	14.89	11.66	3.85	5.25	6.52
$r(3 \times 9, -15, 15)$	6.77		6.77	6.40		7.14	
$r(5 \times 7, 0, 15)$	9.83	546.26	9.84	0.00	0.01	0.02	0.02
$r(5 \times 7, -15, 15)$	A		A	0.00		0.03	
$t(5, 7)$	10.52	3.20	10.52	A	3.00	1.98	58.57

Tabelle 6.3: Laufzeiten mit dem Kriterium teilerfremder Initialterme

rithmus von Di Biase und Urbanke für Matrizen, deren Kern keinen Vektor ohne Nullkomponenten enthält. Allerdings haben wir die Existenz einer positiven Graduierung für die Algorithmen von Hoşten und Sturmfels bzw. Bigatti, La Scala und Robbiano nicht mit dem Simplex-Algorithmus überprüft, so daß diese nur für Matrizen ausgeführt wurden, aus denen sich eine solche Graduierung unmittelbar ablesen läßt.

Alle Tests wurden auf einem **PC** mit **Intel Celeron Prozessor (266 MHz)** und **32 MB RAM** unter **SuSE Linux 6.0** durchgeführt; die Programme wurden mit dem **GNU C++-Compiler egcs-2.91.60** compiliert und mit der Option **-02** optimiert. Für größere Testinstanzen empfiehlt es sich dringend, etwas mehr Hauptspeicher bereitzustellen, damit der Buchberger-Algorithmus keine langsamen Plattenzugriffe tätigen muß.

Wir zeigen nun anhand einiger Testinstanzen, wie sich abgewandelte Konfigurationen des Buchberger-Algorithmus auf die Laufzeiten der IP-Algorithmen auswirken.

Die S-Paar-Kriterien spielen eine entscheidende Rolle für die Effizienz: Schaltet man sie ab, laufen die Algorithmen etwa zwei- bis achtmal so lang (Tabelle

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	4.73	14.30	4.74	3.75	2.12	2.25	2.82
$r(3 \times 9, -15, 15)$	2.73		2.72	3.01		3.11	
$r(5 \times 7, 0, 15)$	1.88	72.65	1.88	0.00	0.00	0.02	0.03
$r(5 \times 7, -15, 15)$	179.44		179.48	0.01		0.03	
$t(5, 7)$	14.94	7.04	14.94	115.61	4.83	3.23	17.95

Tabelle 6.4: Laufzeiten mit dem Kriterium teilerfremder Initialterme und den Gebauer-Möller-Kriterien

	ct	pct	ect	pt	hs	du	blr
$r(3\times 9, 0, 15)$	5.92	18.13	5.92	5.09	2.44	2.81	3.50
$r(3\times 9, -15, 15)$	3.85		3.85	3.61		3.77	
$r(5\times 7, 0, 15)$	2.09	135.06	2.09	0.00	0.01	0.02	0.02
$r(5\times 7, -15, 15)$	350.14		350.24	0.00		0.03	
$t(5,7)$	9.40	4.58	9.40	101.78	3.28	2.33	42.59

Tabelle 6.5: Laufzeiten mit dem Kriterium teilerfremder Initialterme und dem zweiten Buchberger-Kriterium

	ct	pct	ect	pt	hs	du	blr
$r(3\times 9, 0, 15)$	4.68	14.47	4.69	3.97	2.27	2.31	2.93
$r(3\times 9, -15, 15)$	3.08		3.08	3.00		3.27	
$r(5\times 7, 0, 15)$	1.82	74.19	1.83	0.01	0.00	0.03	0.02
$r(5\times 7, -15, 15)$	199.95		199.91	0.01		0.03	
$t(5,7)$	26.05	8.56	26.09	1590.84	5.06	3.59	20.73

Tabelle 6.6: Laufzeiten mit allen S-Paar-Kriterien

6.2). Dabei unterscheiden sich die Auswirkungen einzelner Kriterien je nach benutztem Algorithmus und nach Art der Instanz (Tabelen 6.2 bis 6.6): Bei der Instanz $t(5,7)$ verträgt sich der Conti-Traverso-Algorithmus z.B. besonders gut mit dem zweiten Buchberger-Kriterium (Tabelle 6.5), der Algorithmus von Bigatti, La Scala und Robbiano hingegen mit den Gebauer-Möller-Kriterien (Tabelle 6.4).

Der Eintrag „A“ in den Tabellen kennzeichnet Instanzen, deren Bearbeitung vom System unterbrochen wurde, vermutlich, weil ein Speicherüberlauf drohte.

Die Tabellen 6.7 bis 6.10 untersuchen, welche Rolle die Zwischenreduktionen für die Effizienz des Buchberger-Algorithmus spielen. Ein kompletter Verzicht darauf (Tabelle 6.7) verursacht für die Instanz $r(3\times 9, -15, 15)$ bei Verwendung des Conti-Traverso-Algorithmus eine mehr als 75-fache Laufzeit; an anderen Stellen tritt kaum eine Verschlechterung ein. Allerdings spricht keines der Testergebnisse gegen Zwischenreduktionen. Auf eine Änderung der Zwischenreduktionsquote reagiert der Buchberger-Algorithmus recht unempfindlich (Tabellen 6.8 bis 6.10).

	ct	pct	ect	pt	hs	du	blr
$r(3\times 9, 0, 15)$	26.65	19.12	26.66	4.71	2.30	2.82	3.02
$r(3\times 9, -15, 15)$	199.64		199.73	4.50		9.11	
$r(5\times 7, 0, 15)$	8.95	82.10	8.95	0.01	0.01	0.02	0.05
$r(5\times 7, -15, 15)$	326.61		326.75	0.00		0.03	
$t(5,7)$	51.58	6.93	51.60	486.32	4.66	3.45	94.24

Tabelle 6.7: Laufzeiten ohne Zwischenreduktionen

	ct	pct	ect	pt	hs	du	blr
$r(3\times 9, 0, 15)$	4.67	15.18	4.66	3.75	2.10	2.24	2.79
$r(3\times 9, -15, 15)$	2.62		2.62	3.02		3.05	
$r(5\times 7, 0, 15)$	2.10	75.55	2.10	0.00	0.01	0.02	0.02
$r(5\times 7, -15, 15)$	182.79		182.87	0.00		0.03	
$t(5, 7)$	17.23	6.92	17.24	147.56	4.60	3.06	27.01

Tabelle 6.8: Laufzeiten mit Zwischenreduktionsquote 25%

	ct	pct	ect	pt	hs	du	blr
$r(3\times 9, 0, 15)$	4.60	14.27	4.60	3.76	2.06	2.22	2.76
$r(3\times 9, -15, 15)$	2.61		2.60	2.94		3.04	
$r(5\times 7, 0, 15)$	2.07	75.51	2.07	0.00	0.00	0.02	0.03
$r(5\times 7, -15, 15)$	181.09		181.17	0.01		0.03	
$t(5, 7)$	15.61	6.91	15.62	139.60	4.60	2.95	26.95

Tabelle 6.9: Laufzeiten mit Zwischenreduktionsquote 5%

	ct	pct	ect	pt	hs	du	blr
$r(3\times 9, 0, 15)$	4.60	14.13	4.61	3.75	2.07	2.22	2.74
$r(3\times 9, -15, 15)$	2.61		2.61	2.93		3.04	
$r(5\times 7, 0, 15)$	2.08	75.51	2.08	0.01	0.01	0.02	0.03
$r(5\times 7, -15, 15)$	181.23		181.27	0.01		0.03	
$t(5, 7)$	15.61	6.91	15.61	139.01	4.61	2.99	26.95

Tabelle 6.10: Laufzeiten mit Zwischenreduktion nach jeder Phase

	ct	pct	ect	pt	hs	du	blr
$r(3\times 9, 0, 15)$	11.30	32.42	11.29	7.03	5.26	6.83	7.85
$r(3\times 9, -15, 15)$	8.06		8.06	11.26		11.18	
$r(5\times 7, 0, 15)$	2.82	111.01	2.82	0.01	0.00	0.03	0.03
$r(5\times 7, -15, 15)$	225.18		225.24	0.00		0.04	
$t(5, 7)$	47.74	22.85	47.74	?	78.84	38.47	276.60

Tabelle 6.11: Laufzeiten der Version 1a

	ct	pct	ect	pt	hs	du	blr
$r(3\times 9, 0, 15)$	4.91	14.89	4.90	4.02	2.09	2.19	2.82
$r(3\times 9, -15, 15)$	2.81		2.82	2.53		2.94	
$r(5\times 7, 0, 15)$	1.93	76.15	1.93	0.00	0.01	0.02	0.02
$r(5\times 7, -15, 15)$	178.31		178.51	0.00		0.03	
$t(5, 7)$	15.61	7.25	15.61	85.50	4.35	2.87	25.78

Tabelle 6.12: Laufzeiten der Version 2

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	4.50	13.82	4.51	3.68	2.04	2.12	2.65
$r(3 \times 9, -15, 15)$	2.68		2.69	2.37		2.93	
$r(5 \times 7, 0, 15)$	1.91	74.59	1.91	0.01	0.01	0.02	0.03
$r(5 \times 7, -15, 15)$	177.31		177.29	0.00		0.04	
$t(5, 7)$	14.54	7.18	14.54	78.01	4.44	2.82	25.09

Tabelle 6.13: Laufzeiten der Version 3

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	4.90	15.97	4.89	3.88	2.23	2.39	2.98
$r(3 \times 9, -15, 15)$	2.81		2.81	3.19		3.27	
$r(5 \times 7, 0, 15)$	2.30	79.51	2.30	0.01	0.01	0.02	0.03
$r(5 \times 7, -15, 15)$	192.79		192.90	0.00		0.04	
$t(5, 7)$	16.49	7.32	16.49	152.91	5.02	3.21	28.31

Tabelle 6.14: Laufzeiten mit doppelt verketteten Listen

Nun vergleichen wir die implementierten Varianten des Buchberger-Algorithmus (Tabellen 6.11 bis 6.13). Die Version mit den sortierten S-Paar-Listen (Tabelle 6.11) benötigt deutlich höhere Laufzeiten als die anderen; der Vorteil der sortierten Liste kompensiert den Zusatzaufwand also längst nicht. Version 3 (Tabelle 6.13) schneidet stets ein bißchen besser ab als Version 2 (Tabelle 6.12) und meistens auch als die Standardversion 1 (Tabelle 6.1). Die Unterschiede bewegen sich hier in einem Rahmen von unter 10%, Ausnahme: $t(5, 7)$ mit dem Algorithmus von Pottier. Für die mit „?“ markierte Instanz in Tabelle 6.11 teilte uns der Computer die hübsche Laufzeit von -1224.58 Sekunden mit: Er rechnete wohl so lange, daß der für die Messung verwendete Typ `clock_t` der GNU C Library nicht ausreichte.

Ebenfalls in der Größenordnung von bis zu 10% liegen die Gewinne, die man durch Verwendung von einfach verketteten Listen statt doppelt verketteten erzielt (Tabelle 6.14).

Verwirrung stiftet Tabelle 6.15 mit den Laufzeiten, die man erhält, wenn man statt den vorderen die hinteren Variablen als Trägervariablen wählt. Bei den Versionen des Conti-Traverso-Algorithmus sind (in unserer Implementie-

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	6.28	13.37	6.28	4.84	2.35	2.58	3.56
$r(3 \times 9, -15, 15)$	4.21		4.22	3.63		3.40	
$r(5 \times 7, 0, 15)$	1.73	69.90	1.73	0.01	0.01	0.02	0.03
$r(5 \times 7, -15, 15)$	111.18		111.15	0.01		0.03	
$t(5, 7)$	11.55	4.38	11.55	160.55	5.23	2.75	27.18

Tabelle 6.15: Laufzeiten mit hinteren Variablen als Trägervariablen

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	5.29	20.57	5.28	4.25	1.98	2.19	3.02
$r(3 \times 9, -15, 15)$	3.00		3.00	3.65		3.32	
$r(5 \times 7, 0, 15)$	2.15	105.85	2.21	0.00	0.01	0.00	0.01
$r(5 \times 7, -15, 15)$	268.25		268.05	0.00		0.01	
$t(5, 7)$	15.71	6.85	15.71	154.74	4.71	3.10	30.70

Tabelle 6.16: Laufzeiten mit 2^6 Erzeugerlisten

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	5.21	13.06	5.22	5.12	2.94	2.96	3.63
$r(3 \times 9, -15, 15)$	3.78		3.80	3.33		4.19	
$r(5 \times 7, 0, 15)$	2.44	64.59	2.44	0.02	0.03	0.15	0.15
$r(5 \times 7, -15, 15)$	185.48		185.66	0.01		0.18	
$t(5, 7)$	16.76	7.13	16.77	179.26	4.51	3.71	24.06

Tabelle 6.17: Laufzeiten mit 2^{10} Erzeugerlisten

rung) die vorderen Variablen gewichtete Variablen und die hinteren Eliminationsvariablen; aufgrund ihrer unterschiedlichen Funktion könnte man daher erwarten, daß sich eine der Einstellungen `SUPPORT_VARIABLES_FIRST` bzw. `SUPPORT_VARIABLES_LAST` als die bessere Alternative herausstellt. Aber weder bei den Conti-Traverso-Versionen noch bei den Algorithmen ohne Eliminationsvariablen ergeben die Testdaten ein eindeutiges Bild, welche Einstellung zu bevorzugen ist. Als Schlußfolgerung bleibt die Erkenntnis, daß die Laufzeit des Buchberger-Algorithmus nicht unerheblich von Details bestimmt wird, die auf den ersten Blick völlig unwichtig erscheinen – eine unbefriedigende Situation angesichts der großen offenen Frage, wie man sie in den Griff bekommt.

Die Einträge der Tabellen 6.16 bis 6.18 zeigen den Einfluß, den die Anzahl der Erzeugerlisten bei erweiterter Trägernutzung auf die Laufzeit der Algorithmen hat. Für unsere Beispielinstanzen erweist sich die Standardeinstellung von 2^8 Listen tatsächlich als gute Wahl. Der Trend geht erwartungsgemäß dahin, daß sich kleinere Instanzen besser mit wenigen Listen vertragen; bei größeren Instanzen dürfen es ein paar mehr sein.

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	8.29	16.49	8.31	9.36	5.96	6.32	7.21
$r(3 \times 9, -15, 15)$	4.44		4.44	4.55		9.12	
$r(5 \times 7, 0, 15)$	4.30	65.50	4.29	0.09	0.12	1.15	0.64
$r(5 \times 7, -15, 15)$	176.70		176.89	0.06		1.28	
$t(5, 7)$	20.75	8.11	20.77	173.04	6.82	6.94	27.79

Tabelle 6.18: Laufzeiten mit 2^{12} Erzeugerlisten

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	18.10	47.52	18.10	15.72	4.84	6.94	9.20
$r(3 \times 9, -15, 15)$	7.60		7.60	11.07		9.21	
$r(5 \times 7, 0, 15)$	2.74	258.54	2.75	0.00	0.00	0.00	0.00
$r(5 \times 7, -15, 15)$	618.75		618.96	0.00		0.01	
$t(5, 7)$	14.52	9.16	14.52	210.36	6.95	4.52	78.67

Tabelle 6.19: Laufzeiten ohne erweiterte Trägernutzung

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	28.53	77.02	28.53	23.44	7.20	10.83	13.99
$r(3 \times 9, -15, 15)$	11.23		11.23	16.46		13.45	
$r(5 \times 7, 0, 15)$	4.64	401.26	4.65	0.00	0.00	0.01	0.01
$r(5 \times 7, -15, 15)$	726.79		727.36	0.00		0.00	
$t(5, 7)$	27.35	17.07	27.35	397.04	22.40	13.24	165.50

Tabelle 6.20: Laufzeiten ohne Trägernutzung

Über die Vorzüge der erweiterten Trägernutzung an sich gibt Tabelle 6.19 Auskunft: Ein Verzicht darauf verursacht oft drei- bis viermal so hohe Laufzeiten. Schaltet man auch auf der Ebene der Binome die Trägernutzung ab, so erhöhen sich die Laufzeiten der meisten Instanzen um weitere 60%, beim Transportproblem sogar um etwa 100% (Tabelle 6.20). Insgesamt drückt die konsequente Trägernutzung die Laufzeit auf etwa ein Fünftel!

Für Statistikfreunde veranschaulichen wir in Tabelle 6.21 noch die Auswirkungen einer sehr schlechten Konfiguration: ohne Trägernutzung, mit doppelt verketteten Listen, ohne Benutzung von S-Paar-Kriterien und ohne Zwischenreduktionen. Diese Konfiguration kommt sehr kleinen Instanzen sogar entgegen, da sie jeden Zusatzaufwand vermeidet; ansonsten sprechen die gegenüber der Standardkonfiguration verzwanzigfachten Laufzeiten für sich.

Sensitivität des Buchberger-Algorithmus

Unsere wenigen Beispielinstanzen lassen bereits Tendenzen erkennen, wie der Buchberger-Algorithmus auf Faktoren wie Dimension und Koeffizientengröße der Matrizen reagiert. Auf 0-1-Matrizen erzielt er wesentlich bessere Laufzeiten

	ct	pct	ect	pt	hs	du	blr
$r(3 \times 9, 0, 15)$	817.64	660.11	815.35	204.28	37.57	67.09	80.91
$r(3 \times 9, -15, 15)$	1445.70		1446.32	129.98		495.99	
$r(5 \times 7, 0, 15)$	251.91	-823.48	251.65	0.01	0.00	0.01	0.04
$r(5 \times 7, -15, 15)$	A		A	0.00		0.01	
$t(5, 7)$	345.63	121.22	346.14	A	99.95	70.87	917.02

Tabelle 6.21: Laufzeiten einer sehr schlechten Konfiguration

als auf Matrizen mit größeren Einträgen. Unter der Annahme, daß bei Matrizen mit kleinen Koeffizienten in der Regel auch die LLL-reduzierten Gitterbasen des Kerns kleine Koeffizienten haben, entsprechen die Ergebnisse im torischen Fall den Erfahrungen für allgemeinere Ideale.

Ebenso verursachen Matrizen mit vielen Einträgen in der Regel mehr Arbeit als Matrizen mit wenigen. Die Ergebnisse für die Matrizen des Transportproblems deuten allerdings darauf hin, daß der Buchberger-Algorithmus regelmäßige Strukturen besser verarbeitet als beliebige.

Im Zusammenhang mit unseren Betrachtungen zur Koeffizientengröße überrascht der Vergleich der Instanzen des Transportproblems mit verschiedenen Kostenvektoren: Der Vektor mit größeren Koeffizienten führt zu kleineren Gröbnerbasen und verkürzter Laufzeit. Vermutlich ist das Zufall. Dennoch kann man nicht davon ausgehen, daß die Koeffizientengröße des Kostenvektors die Laufzeit auch nur annähernd so stark beeinflußt wie die Koeffizientengröße der Matrix: In den Kapiteln 1 und 2 von [30] wird gezeigt, daß sich die unendlich vielen nichtnegativen Gewichtsvektoren des $\mathbb{R}^n$ in endlich viele Äquivalenzklassen einteilen lassen, wenn es um die Berechnung von Gröbnerbasen eines festen Ideals in $K[x_1, \ldots, x_n]$ geht.

Die Anzahl der Erzeuger des Anfangsideals gibt keine Hinweise auf die Laufzeit: 1×11-Matrizen mit 10-dimensionalem Kern werden viel schneller abgehandelt als 5×7-Matrizen mit 2-dimensionalem Kern. Und das, obwohl bei den 1×11-Matrizen auch noch die Anzahl der Variablen höher ist, falls man nicht die Conti-Traverso-Algorithmen benutzt. Vermutlich sehen die Gitterbasen der 1×11-Matrizen etwas netter aus als die der 5×7-Matrizen. Unsere Algorithmen bieten sich angesichts dieser Ergebnisse zur Lösung von Rucksackproblemen (Beispiel 2.6) an.

Obwohl gewisse Trends nicht zu leugnen sind, tauchen in den Tabellen immer wieder auffällig hohe oder niedrige Werte auf; Laufzeitschätzungen für den Buchberger-Algorithmus sind eben sehr schwierig.

6.3 Ein Algorithmenvergleich

Ein Blick auf die Testdaten in Tabelle 6.1 zeigt, daß die Versionen des Conti-Traverso-Algorithmus auf größeren Instanzen deutlich länger laufen als die anderen Algorithmen. Das entspricht unseren in Abschnitt 3.7.1 formulierten Erwartungen und rechtfertigt sich durch die Tatsache, daß die Conti-Traverso-Algorithmen bei der Lösung von IP-Instanzen keine Initiallösung brauchen – bis auf die Variante mit Elimination, die ohnehin nur zu Testzwecken implementiert wurde. Hierzu bemerkt man, daß der eigentliche Eliminationsschritt an der Laufzeit fast gar nichts ändert: Um dessen Laufzeit unterscheiden sich nämlich die Spalten `ct` und `ect`. Kürzere Laufzeiten in der `ect`-Spalte rühren daher, daß man die Systemauslastung nicht ganz konstant halten kann.

Um wieviel höher die Laufzeiten der Conti-Traverso-Algorithmen im Vergleich zu denen der anderen Algorithmen liegen, hängt augenscheinlich davon ab, wie der Größenunterschied zwischen den Gröbnerbasen ausfällt. Bei den Instanzen mit einzeiligen Matrizen sind die erweiterten Gröbnerbasen der Conti-

Traverso-Algorithmen nur geringfügig größer als die Gröbnerbasen der eigentlichen torischen Ideale; entsprechend zeigen sich die Conti-Traverso-Algorithmen dort wettbewerbsfähig, soweit man das bei den kleinen Laufzeiten beurteilen kann. Auch beim Transportproblem schneiden sie nicht allzu schlecht ab; allerdings lohnt sich dort der Zusatzaufwand z.B. im Vergleich zum Di-Biase-Urbanke-Algorithmus nicht, weil die Bestimmung einer Initiallösung einfach ist.

Überraschenderweise erweist sich der nichtnegative Conti-Traverso-Algorithmus, der gegenüber der allgemeinen Variante eine Variable einspart, keinesfalls als überlegen; seine Gröbnerbasen sind meist größer, seine Laufzeiten höher als die der allgemeinen Version. Ausnahmen bilden wiederum die Instanzen des Transportproblems.

Der Algorithmus von Pottier hinterläßt sehr unterschiedliche Eindrücke: Er ist universell einsetzbar, erzielt bei Matrizen mit negativen Koeffizienten oft die besten Laufzeiten und ist bei der Instanz des 5×7-Transportproblems hoffnungslos langsam. Insgesamt empfiehlt er sich als Alternative für Instanzen, bei denen der Algorithmus von Hoşten und Sturmfels nicht eingesetzt werden kann.

Letzterer stellt sich als der schnellste Algorithmus für zufällige Matrizen heraus – unabhängig von der Konfiguration des Buchberger-Algorithmus. Nur bei den Transportmatrizen ist er dem Algorithmus von Di Biase und Urbanke unterlegen. Größter Kritikpunkt bleibt die eingeschränkte Verwendbarkeit.

Der Algorithmus von Di Biase und Urbanke löst am schnellstens Transportprobleme. Ansonsten ist er fast immer der zweitschnellste Algorithmus, bei Matrizen mit negativen Koeffizienten nicht viel langsamer als der Algorithmus von Pottier, bei Matrizen ohne solche nicht viel langsamer als der Algorithmus von Hoşten und Sturmfels. Da er beständig niedrige Laufzeiten produziert, empfiehlt er sich in vielen Fällen als Alternative – unter unseren Testinstanzen befindet sich lediglich eine Instanz, für die er nicht eingesetzt werden kann, nämlich $r(5 \times 7, 0, 1)$. Vorsicht ist bei der Wahl des Datentyps für ganze Zahlen geboten: Durch die wechselnden Eliminationsordnungen produziert der Algorithmus Exponentenvektoren mit sehr großen Komponenten; bei Verwendung von `short` führen schon kleine Instanzen zu einem Überlauf, z.B. unsere Testinstanz $r(5 \times 7, -15, 15)$.

Der Algorithmus von Bigatti, La Scala und Robbiano kann in denselben Fällen eingesetzt werden wie der Algorithmus von Hoşten und Sturmfels, ist aber stets langsamer. Besonders ungeeignet erweist er sich für das Transportproblem. Seine Beurteilung ist allerdings nicht ganz fair, da er für einen Buchberger-Algorithmus konzipiert wurde, der gradweise vorgeht – unsere Implementierung tut das nicht, auch wenn man die Trägernutzung abschaltet.

	SINGULAR		toric_ideal	
	ct	pct	ct	pct
$r(1\times 11,0,1)$	0.00	0.00	0.00	0.00
$r(1\times 11,0,5)$	0.01	0.00	0.01	0.00
$r(1\times 11,-5,5)$	0.01		0.00	
$r(1\times 11,0,15)$	0.07	0.06	0.03	0.03
$r(1\times 11,-15,15)$	0.02		0.01	
$r(3\times 9,0,1)$	0.00	0.01	0.00	0.00
$r(3\times 9,0,5)$	1.77	2.83	1.18	2.85
$r(3\times 9,-5,5)$	3.06		0.55	
$r(3\times 9,0,15)$	7.98	10.05	4.62	14.95
$r(3\times 9,-15,15)$	48.70		2.60	
$r(5\times 7,0,1)$	0.00	0.00	0.00	0.00
$r(5\times 7,0,5)$	2.05	3.78	1.73	5.98
$r(5\times 7,-5,5)$	54.77		3.55	
$r(5\times 7,0,15)$	12.53	48.80	2.07	75.91
$r(5\times 7,-15,15)$	275.22		181.39	
$t(3,4)$	0.06	0.06	0.02	0.02
$t(5,7)$	35.83	22.99	16.37	6.91
$t(5,7)_{50}$	40.78	19.26	12.37	6.57

Tabelle 6.22: Vergleich mit SINGULAR

6.4 Tests gegen SINGULAR

Das Computeralgebrasystem SINGULAR stellt eine Implementierung des Buchberger-Algorithmus für allgemeine Ideale zur Verfügung. Diese bietet sich dafür an, die Wirkung zu untersuchen, die die Spezialisierung auf torische Ideale auf die Laufzeiten hat.

Da in SINGULAR Dinge wie die Variation von monomialen Ordnungen sehr schwer zu realisieren sind, implementieren wir die Algorithmen nicht vollständig, sondern beschränken den Laufzeitvergleich auf den Buchberger-Algorithmus. Wir rufen dessen Implementierung in SINGULAR also mit den einfach zu bestimmenden Startidealen der Conti-Traverso-Algorithmen auf, und zwar für dieselben Testinstanzen wie in Abschnitt 6.2. Dabei messen wir die Zeiten zur Berechnung der Gröbnerbasen der torischen Ideale. Tabelle 6.22 zeigt die Laufzeiten von SINGULAR und zum Vergleich nochmals die Laufzeiten unserer Implementierung.

In den meisten Fällen ist unsere Implementierung deutlich schneller als SINGULAR. Ausnahmen bilden die Instanzen mit Zufallsmatrizen, die mit dem nichtnegativen Conti-Traverso-Algorithmus gelöst werden – auf diesen erzielt unsere Implementierung ohnehin schlechte Resultate. Die Gewinne durch die Spezialisierung auf torische Ideale sind wahrscheinlich noch höher, als die Tabelle suggeriert; denn die grundlegenden Bestandteile von SINGULAR sind in sehr maschinennahem C implementiert (vgl. [1],[14]) und somit effizienter als eine vergleichbare Implementierung in C++. Zudem berechnet SINGULAR in

	1	2	3	4	5
$r(1 \times 11, 0, 1)$	0.000	0.000	0.000	0.000	0.000
$r(1 \times 11, 0, 5)$	0.000	0.000	0.000	0.000	0.000
$r(1 \times 11, -5, 5)$	0.000	0.000	0.000	0.000	0.000
$r(1 \times 11, 0, 15)$	0.000	0.000	0.010	0.000	0.000
$r(1 \times 11, -15, 15)$	0.000	0.000	0.000	0.000	0.000
$r(3 \times 9, 0, 1)$	0.000	0.000	0.000	0.000	0.000
$r(3 \times 9, 0, 5)$	0.000	0.000	0.000	0.010	0.000
$r(3 \times 9, -5, 5)$	0.000	0.000	0.000	0.000	0.010
$r(3 \times 9, 0, 15)$	0.000	0.000	0.000	0.010	0.000
$r(3 \times 9, -15, 15)$	0.010	0.000	0.010	0.000	0.010
$r(5 \times 7, 0, 1)$	0.000	0.000	0.000	0.000	0.000
$r(5 \times 7, 0, 5)$	0.000	0.000	0.000	0.000	0.000
$r(5 \times 7, -5, 5)$	0.000	0.000	0.000	0.000	0.000
$r(5 \times 7, 0, 15)$	0.000	0.000	0.000	0.000	0.000
$r(5 \times 7, -15, 15)$	0.000	0.000	0.000	0.000	0.000
$t(3, 4)$	0.000	0.000	0.000	0.000	0.000
$t(5, 7)$	0.000	0.000	0.010	0.000	0.010
$t(5, 7)_{50}$	0.010	0.000	0.000	0.010	0.000

Tabelle 6.23: Reduktionszeiten

der Regel nur eine minimale, nicht aber reduzierte Gröbnerbasis (siehe Anhang B.5).

Sowohl bei unserer Implementierung als auch bei SINGULAR wurden nur CPU-Zeiten, d.h. reine Rechenzeiten, gemessen.

6.5 Die Reduktionsphase

Um die Zeit zu messen, die die IP-Algorithmen für die Reduktion brauchen, müssen wir zunächst Begrenzungsvektoren und Initiallösungen für unsere Testmatrizen generieren: Zu jeder Testmatrix wählen wir zufällig fünf Initiallösungen mit Koeffizienten aus dem Bereich zwischen 0 und der hundertfachen oberen Schranke für die Matrizeneinträge. Die Koeffizienten sollen so groß werden, damit bei der Reduktion auch etwas zu tun ist; denn Initiallösungen mit kleinen Koeffizienten haben meist geringes Gewicht und liegen schon nahe an der Optimallösung.

Aus jeweils zwei der fünf Initiallösungen bestimmen wir durch Multiplikation an die Matrix die zugehörigen Begrenzungsvektoren, um garantiert lösbare Instanzen für die Conti-Traverso-Algorithmen zu erhalten. Für diese wählen wir weiterhin pro Instanz drei zufällige Begrenzungsvektoren, um zu sehen, wie Lösbarkeitstests auf eventuell unlösbaren Instanzen aussehen. Die Koeffizienten der Vektoren für die nichtnegative Version des Conti-Traverso-Algorithmus stammen aus dem gleichen Bereich wie die der Initiallösungen; für die allgemeinen Version erlauben wir auch negative Koeffizienten.

	1	2	3	4	5
$r(1\times 11,0,1)$	0.000	0.000	u 0.000	0.000	0.000
$r(1\times 11,0,5)$	0.000	0.000	u 0.000	u 0.000	u 0.000
$r(1\times 11,-5,5)$	0.000	0.000	0.000	0.000	0.000
$r(1\times 11,0,15)$	0.000	0.000	0.000	u 0.000	0.000
$r(1\times 11,-15,15)$	0.000	0.000	0.000	0.000	0.000
$r(3\times 9,0,1)$	0.000	0.000	u 0.000	u 0.000	u 0.000
$r(3\times 9,0,5)$	0.010	0.000	u 0.000	u 0.000	u 0.000
$r(3\times 9,-5,5)$	0.030	0.020	0.020	0.010	0.010
$r(3\times 9,0,15)$	0.010	0.000	u 0.000	u 0.010	u 0.000
$r(3\times 9,-15,15)$	0.100	0.100	0.040	0.040	0.040
$r(5\times 7,0,1)$	0.000	0.000	u 0.000	u 0.000	u 0.000
$r(5\times 7,0,5)$	0.020	0.000	u 0.010	u 0.000	u 0.000
$r(5\times 7,-5,5)$	0.010	0.020	u 0.020	u 0.020	u 0.010
$r(5\times 7,0,15)$	0.000	0.000	u 0.000	u 0.000	u 0.000
$r(5\times 7,-15,15)$	0.050	0.080	u 0.050	u 0.030	u 0.040
$t(3,4)$	0.000	0.000	u 0.000	u 0.000	u 0.000
$t(5,7)$	0.000	0.050	u 0.000	u 0.010	u 0.010
$t(5,7)_{50}$	0.010	0.020	u 0.010	u 0.000	u 0.020

Tabelle 6.24: Reduktionszeiten des Conti-Traverso-Algorithmus

Bemerkung 6.1 Bei unserer Interpretation des Transportproblems ergeben negative Koeffizienten des Begrenzungsvektors keinen Sinn. Außerdem gehört nicht jeder Begrenzungsvektor zu einem Transportproblem wie in 2.8 formuliert; denn mit den Bezeichnungen aus 2.8 müssen die Summen der Koeffizienten f_i bzw. l_j übereinstimmen. Zu Testzwecken dürfen wir die Interpretation aber ignorieren. □

Der allgemeine und der nichtnegative Conti-Traverso-Algorithmus berechnen torische Ideale zu erweiterten Matrizen; alle anderen Algorithmen berechnen das torische Ideal zur Eingabematrix selbst. Somit berechnen die letzteren auch die gleiche reduzierte Gröbnerbasis, denn die Implementierung verfeinert den Kostenvektor bei all diesen Algorithmen zur gleichen Termordnung. Für die Reduktion spielt es keine Rolle, mit welchem Algorithmus das Ideal berechnet wurde; wir müssen sie also nur einmal und nicht für jeden Algorithmus getrennt durchführen.

Tabelle 6.23 zeigt die Reduktionszeiten der gewählten Initiallösungen durch die torischen Ideale, die zu den Testinstanzen aus Abschnitt 6.2 gehören. Analog enthalten die Tabellen 6.24 und 6.25 die Reduktionszeiten der Begrenzungsvektoren durch die erweiterten Ideale der Conti-Traverso-Algorithmen. Unlösbare Instanzen sind mit „u" markiert; die ersten beiden Spalten zeigen nur lösbare Instanzen, nämlich die mit den Begrenzungsvektoren, die wie oben beschrieben aus Initiallösungen bestimmt wurden. Die konkreten Vektoren zu allen Instanzen findet man in Anhang B.3.

	1	2	3	4	5
$r(1 \times 11, 0, 1)$	0.000	0.000	0.000	0.000	0.000
$r(1 \times 11, 0, 5)$	0.000	0.000	0.000	0.000	0.000
$r(1 \times 11, 0, 15)$	0.000	0.000	0.000	0.000	0.000
$r(3 \times 9, 0, 1)$	0.000	0.000	0.000	0.000	0.000
$r(3 \times 9, 0, 5)$	0.010	0.000	u 0.000	u 0.000	0.000
$r(3 \times 9, 0, 15)$	0.000	0.000	u 0.000	u 0.000	0.000
$r(5 \times 7, 0, 1)$	0.000	0.000	u 0.000	u 0.000	u 0.000
$r(5 \times 7, 0, 5)$	0.000	0.010	u 0.000	u 0.000	u 0.000
$r(5 \times 7, 0, 15)$	0.690	0.150	u 0.000	u 0.000	u 0.000
$t(3, 4)$	0.000	0.000	u 0.000	u 0.000	u 0.000
$t(5, 7)$	0.010	0.030	u 0.020	u 0.000	u 0.010
$t(5, 7)_{50}$	0.010	0.020	u 0.020	u 0.010	u 0.000

Tabelle 6.25: Reduktionszeiten des nichtnegativen Conti-Traverso-Algorithmus

Bemerkung 6.2 Betrachtet man die Spalten 3,4 und 5 in 6.24 und 6.25, die zu den zufälligen Begrenzungsvektoren gehören, so trifft man beim Transportproblem nur unlösbare Instanzen an. Das liegt daran, daß die Zeilen der zugehörigen Matrizen linear abhängig sind – mit den Bezeichnungen aus 2.8 sind die Summen der f_i-Zeilen und der l_j-Zeilen gleich. □

Wenig überraschend wächst bei zufälligen Matrizen die Chance, eine Lösung zu finden, mit der Dimension des Kerns: Unlösbare Instanzen sind bei den 1×11-Matrizen deutlich seltener als bei den 5×7-Matrizen.

Im Vergleich zur Gröbnerbasisberechnung verbraucht die Reduktionsphase sehr wenig Zeit. Anscheinend messen die benutzten Funktionen der GNU C Library die CPU-Zeit nur auf eine hunderstel Sekunde genau, so daß man die exakte Rechenzeit überhaupt nicht feststellen kann. Übrigens unterscheiden sich die Optimallösungen und Initiallösungen in der großen Mehrheit der Fälle, so daß tatsächlich Reduktionen stattgefunden haben (vgl. die `.sol.`-Dateien in Anhang B.3).

6.6 Variation der Kostenfunktion

Der vorangehende Abschnitt zeigt, daß die Variation der rechten Seite einer IP-Instanz kein Problem für unsere Algorithmen darstellt: Eine solche verursacht lediglich eine zusätzliche Reduktion. Wir untersuchen nun, wie es sich mit der Variation des Kostenvektors verhält, d.h. wieviel Zeit die Umrechnung einer Gröbnerbasis benötigt. Dazu wählen wir zu jeder Testinstanz einen neuen zufälligen Kostenvektor, dessen Koeffizienten im selben Zahlenbereich wie zuvor liegen, und rufen das Programm `change_cost` mit der alten Gröbnerbasis auf. Wir unterscheiden dabei wieder zwischen dem allgemeinen und dem nichtnegativen Conti-Traverso-Algorithmus sowie den restlichen Algorithmen, weil diese drei Gruppen unterschiedliche Ideale berechnen.

	$\mathcal{GB}$	ct	pct	andere
$r(1 \times 11, 0, 1)$	10 (12,11)	0.00	0.00	0.01
$r(1 \times 11, 0, 5)$	12 (16,13)	0.01	0.00	0.00
$r(1 \times 11, -5, 5)$	11 (13)	0.00		0.00
$r(1 \times 11, 0, 15)$	38 (48,48)	0.01	0.01	0.01
$r(1 \times 11, -15, 15)$	13 (15)	0.00		0.00
$r(3 \times 9, 0, 1)$	7 (12,10)	0.00	0.00	0.00
$r(3 \times 9, 0, 5)$	92 (233,353)	0.69	2.20	0.08
$r(3 \times 9, -5, 5)$	257 (261)	1.00		0.89
$r(3 \times 9, 0, 15)$	324 (617,873)	6.52	17.34	1.46
$r(3 \times 9, -15, 15)$	443 (447)	3.68		3.49
$r(5 \times 7, 0, 1)$	2 (8,12)	0.00	0.00	0.00
$r(5 \times 7, 0, 5)$	14 (325,563)	2.11	7.65	0.01
$r(5 \times 7, -5, 5)$	22 (476)	4.23		0.00
$r(5 \times 7, 0, 15)$	9 (300,1616)	1.76	96.07	0.00
$r(5 \times 7, -15, 15)$	9 (1826)	355.04		0.01
$t(3, 4)$	18 (31,60)	0.00	0.02	0.00
$t(5, 7)$	250 (432,460)	15.15	9.01	0.70
$t(5, 7)_{50}$	241 (405,451)	10.76	8.51	0.64

Tabelle 6.26: Variation des Kostenvektors

Tabelle 6.26 zeigt die Laufzeiten für die Umrechnung der Gröbnerbasen. In der mit $\mathcal{GB}$ markierten Spalte sind die Größen der neuen Gröbnerbasen eingetragen. Ein Vergleich der Werte mit denen in Tabelle 6.1 ergibt, daß sich für den allgemeinen Conti-Traverso-Algorithmus die Umrechnung einer Gröbnerbasis auf eine neue Kostenfunktion nicht lohnt: Wenn die Gröbnerbasis durch die Umrechnung wächst, ist die Laufzeit größer als die in Tabelle 6.1; wenn die Gröbnerbasis schrumpft, ist die Laufzeit meistens kleiner. Eine Ausnahme bilden die Transportprobleme, für die die Umrechnung der Gröbnerbasis etwas weniger Zeit braucht als die Neuberechnung.

Für die nichtnegative Conti-Traverso-Version präsentiert sich ein ähnliches Bild; diesmal scheint die Umrechnung für die Transportprobleme allerdings die schlechtere Alternative zu sein.

Für alle anderen Algorithmen lohnt sich eine Umrechnung gegenüber einem Neustart: Außer bei den Instanzen $r(3 \times 9, -5, 5)$ und $r(3 \times 9, -15, 15)$, wo die Gröbnerbasen durch die neue Kostenfunktion stark wachsen, sind die Zeiten in Tabelle 6.26 wesentlich besser als die in Tabelle 6.1; besonders hoch fallen die Gewinne beim Transportproblem aus.

Die guten Resultate erklären sich daraus, daß diese Algorithmen viel Arbeit in die Berechnung des torischen Ideals ihrer Eingabematrix stecken; bei einer Umrechnung der Gröbnerbasis fällt diese Arbeit weg. Die Conti-Traverso-Algorithmen hingegen bestehen praktisch nur aus einer Gröbnerbasisberechnung, so daß eine Umrechnung nichts einspart. Wenn man den Kostenvektor nur gering variiert, gilt diese Argumentation allerdings nicht, und man darf auf bessere Ergebnisse auch für die Conti-Traverso-Algorithmen hoffen.

Ansonsten braucht man einen kleinen Trick, um die Variation des Kostenvektors für diese effektiv zu gestalten: Anstatt die gesamte Gröbnerbasis umzurechen, tut man dies nur für den Teil, der keine Eliminationsvariablen enthält. Das so erhaltene Erzeugendensystem eignet sich für einen Zweiphasen-Algorithmus (vgl. 3.3).

6.7 Tests gegen andere Verfahren zur ganzzahligen Optimierung

Wir testen unsere Algorithmen nun gegen professionelle Software zur ganzzahligen Optimierung. Zum Modellieren der Testinstanzen benutzen wir das **MPL Modeling System,**Release 4.1, von **Maximal Software** (siehe [24]), zum Lösen den dort integrierten **CPLEX 300 Solver.**Dieses Programm läuft unter Windows 95. Beim Vergleich von Laufzeiten ist angesichts der unterschiedlichen Betriebssysteme also Vorsicht geboten.

Zum Testen verwenden wir die gleichen Matrizen wie zuvor. Zu jeder Matrix lösen wir zwei Probleme, nämlich die mit den rechten Seiten, die den Spalten 1 und 2 der Tabelle 6.24 zugrundeliegen. Wir testen somit nur lösbare Instanzen; das oben angegebene Programm kann zwar Unlösbarkeit feststellen, liefert in diesem Fall jedoch keine Angaben zur Rechenzeit.

In Anhang B.6 findet man ein Beispiel für eine Modellierungs- und eine Lösungsdatei. Bei Matrizen mit kleinen Koeffizienten findet CPLEX oft Lösungen, die sich von denen unserer Algorithmen unterscheiden – allerdings nicht im Zielfunktionswert. In diesem Fall handelt es sich um Instanzen, die nicht eindeutig lösbar sind.

Tabelle 6.27 zeigt die Laufzeiten der Testinstanzen mit CPLEX. Die geringen Laufzeiten der Transportinstanzen lassen vermuten, daß die Optimallösungen dafür bereits über Preprocessing gefunden werden. In der Tat benutzt CPLEX für ganzzahlige Optimierungsprobleme ein sogenanntes **Branch-and-Bound-Verfahren**. Solche Verfahren lösen bei IP-Problemen in der Regel zuerst die lineare Relaxierung. Diese wiederum liefert bei Transportproblemen bereits eine ganzzahlige Lösung (vgl. Seite 48).

Im Vergleich zu den Werten in Tabelle 6.1 ist CPLEX nicht nur bei der Lösung von Transportproblemen überlegen, sondern auch bei den meisten anderen Instanzen – jedenfalls, wenn man fairerweise die Conti-Traverso-Algorithmen zum Vergleich heranzieht, die keine Initiallösung brauchen. Die anderen Algorithmen benötigen auf den meisten zufälligen Instanzen weniger Zeit als CPLEX.

Dennoch machen auch die Conti-Traverso-Algorithmen in manchen Fällen CPLEX Konkurrenz: Auf den Instanzen $r(3 \times 9, -15, 15)$, $r(5 \times 7, 0, 15)$ und $r(5 \times 7, -15, 15)$ erzielen sie zum Teil bessere Werte. Als generell überlegen erweisen sie sich auf Instanzen mit einzeiligen Matrizen, deren Laufzeiten uns bereits zuvor positiv aufgefallen sind.

Zulässig ist der Vergleich der Tabellen 6.1 und 6.27 nur, weil die Reduktionsphase der Algorithmen aus Kapitel 3 von der Laufzeit her nicht ins Gewicht

	1	2
$r(1 \times 11, 0, 1)$	0.05	0.05
$r(1 \times 11, 0, 5)$	0.05	0.05
$r(1 \times 11, -5, 5)$	0.39	0.00
$r(1 \times 11, 0, 15)$	0.44	3.46
$r(1 \times 11, -15, 15)$	0.05	0.00
$r(3 \times 9, 0, 1)$	0.00	0.00
$r(3 \times 9, 0, 5)$	0.11	2.03
$r(3 \times 9, -5, 5)$	0.33	0.22
$r(3 \times 9, 0, 15)$	0.27	0.38
$r(3 \times 9, -15, 15)$	7.25	3.95
$r(5 \times 7, 0, 1)$	0.00	0.00
$r(5 \times 7, 0, 5)$	0.77	0.16
$r(5 \times 7, -5, 5)$	0.05	0.44
$r(5 \times 7, 0, 15)$	3.79	8.95
$r(5 \times 7, -15, 15)$	229	98
$t(3, 4)$	0.00	0.00
$t(5, 7)$	0.00	0.16
$t(5, 7)_{50}$	0.11	0.00

Tabelle 6.27: Laufzeiten der Testinstanzen mit CPLEX

fällt. Möchte man nun mehrere Probleme mit gleicher Matrix lösen, darf man aufgrund der Ergebnisse der Abschnitte 6.5 und 6.6 von unseren Algorithmen eine niedrigere Gesamtlaufzeit als von CPLEX erwarten (außer in Spezialfällen wie dem des Transportproblems). Ab welcher Anzahl von Problemen unsere Algorithmen CPLEX überholen, hängt von den konkreten Instanzen ab; bei den Testinstanzen liegt diese Zahl in einer moderaten Größenordnung zwischen 1 und 10.

Wie sieht es also mit praktischen Einsatzmöglichkeiten für unsere Optimierungsalgorithmen aus? Dazu ein kleines Beispiel:

Wir betrachten wieder die Produktionsplanung 2.2. Solange die Fabrikleitung keine Änderung des Produktionsprozesses beschließt, bleiben die Koefizienten der Modellierungsmatrix gleich. Nicht in den Händen der Chefs, weil vom Markt abhängig, liegt hingegen der Gewinn, den eine Einheit eines Produktes bringt. Marktschwankungen verursachen also Änderungen der Kostenfunktion. Ebenso kann es zu einem veränderten Angebot an Ressourcen kommen (etwa saisonal bedingt), d.h. zu einer Änderung des Begrenzungsvektors. Im Laufe der Zeit kommen so mehrere Planungsprobleme zustande; löst man das erste Problem mit dem Conti-Traverso-Algorithmus, so kann man in Zukunft immer wieder von dieser Lösung profitieren.

Die Lösung leicht veränderter Instanzen unter der Verwendung ursprünglicher Lösungen nennt man **Postoptimierung.** Und für diese Anwendung sollte man unsere Algorithmen im Auge behalten.

Anhang A

Quellcode

A.1 Header-Dateien

Dieser Abschnitt führt die `.h`-Dateien unseres Programmpakets auf. Diese enthalten die Deklarationen der Klassen und Funktionen, also die Schnittstelle für eine Erweiterung des Pakets. Die Dateien sind nach Namen in alphabetischer Reihenfolge geordnet. Für die Programmstruktur siehe Abbildung 5.2.

A.1.1 BigInt.h

```
// BigInt.h

// A BigInt wrapper class for the GNU MP library. Only functions
// needed by the LLL-algorithm.

#ifndef BIG_INT_H
#define BIG_INT_H

#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <ctype.h>
#include <string.h>
#include <gmp.h>

class BigInt
{
    mpz_t value;

    public:

    BigInt( );
    BigInt( long int );
    BigInt( unsigned long int );
    BigInt( int );
```

```
BigInt( unsigned int );
BigInt( short int );
BigInt( unsigned short int );
BigInt( char );
BigInt( unsigned char );
BigInt( const BigInt& );
~BigInt( );

BigInt& operator = ( long int );
BigInt& operator = ( unsigned long int );
BigInt& operator = ( int );
BigInt& operator = ( unsigned int );
BigInt& operator = ( short int );
BigInt& operator = ( unsigned short int );
BigInt& operator = ( char );
BigInt& operator = ( unsigned char );
BigInt& operator = ( const BigInt& );

operator bool( );
operator long int( );
operator unsigned long int( );
operator int( );
operator unsigned int( );
operator short int( );
operator unsigned short int( );
operator char( );
operator unsigned char( );

BigInt  operator - ( );
BigInt& operator += ( const BigInt& );
BigInt& operator -= ( const BigInt& );
BigInt& operator *= ( const BigInt& );
BigInt& operator /= ( const BigInt& );
BigInt& operator ++ ( );
BigInt  operator ++ ( int );
BigInt& operator -- ( );
BigInt  operator -- ( int );

friend BigInt operator - ( const BigInt& );

friend bool operator <  ( const BigInt&, const BigInt& );
friend bool operator <= ( const BigInt&, const BigInt& );
friend bool operator >  ( const BigInt&, const BigInt& );
friend bool operator >= ( const BigInt&, const BigInt& );
friend bool operator == ( const BigInt&, const BigInt& );
friend bool operator != ( const BigInt&, const BigInt& );
friend bool operator <  ( const long int&, const BigInt& );
```

```
    friend bool operator <= ( const long int&, const BigInt& );
    friend bool operator >  ( const long int&, const BigInt& );
    friend bool operator >= ( const long int&, const BigInt& );
    friend bool operator == ( const long int&, const BigInt& );
    friend bool operator != ( const long int&, const BigInt& );
    friend bool operator <  ( const BigInt&, const long int& );
    friend bool operator <= ( const BigInt&, const long int& );
    friend bool operator >  ( const BigInt&, const long int& );
    friend bool operator >= ( const BigInt&, const long int& );
    friend bool operator == ( const BigInt&, const long int& );
    friend bool operator != ( const BigInt&, const long int& );

    friend int    sgn ( const BigInt& );
    friend BigInt abs ( const BigInt& );
};

BigInt operator + ( const BigInt&, const BigInt& );
BigInt operator - ( const BigInt&, const BigInt& );
BigInt operator * ( const BigInt&, const BigInt& );
BigInt operator / ( const BigInt&, const BigInt& );
BigInt operator + ( const long int&, const BigInt& );
BigInt operator - ( const long int&, const BigInt& );
BigInt operator * ( const long int&, const BigInt& );
BigInt operator / ( const long int&, const BigInt& );
BigInt operator + ( const BigInt&, const long int& );
BigInt operator - ( const BigInt&, const long int& );
BigInt operator * ( const BigInt&, const long int& );
BigInt operator / ( const BigInt&, const long int& );

#endif  // BIG_INT_H
```

A.1.2 binomial.h

```
// binomial.h

// The binomials considered here are differences of power-products of
// a special kind. The initial monomial or head (with respect to a
// given term ordering as described in term_ordering.h) has
// coefficient 1, the other monomial (the tail) has coefficient
// -1. The head and the tail of a binomial are always relatively
// prime, i.e., they do not involve common variables.

// Such a binomial in n variables is represented as an Integer vector
// of size n: If the i-th variable is involved in the head with
// exponent k, the i-th component of the vector is set to k. If the
// i-th variable is involved in the tail with exponent k, the i-th
// component of the vector is set to -k.
```

```
// The members head_support and tail_support of type unsigned long
// have to be understood as bit-vectors with m components, where m is
// the number of bits in a long integer. The i-th bit of head_support
// tells us if the i-th variable is involved in the head of the
// binomial (for i<=m): it is 1 iff this variable appears in the
// head. Analogously for tail_support. Using these vectors, the
// question if a binomial can be reduced by another can often be
// answered by performing a simple bitwise operation.

// To avoid global variables, the number of variables is taken by the
// constructors. But as it is the same for all appearing binomials
// during the run of our algorithms (unless some elimination is done),
// the member functions do not perform any range checks. This is for
// efficiency reasons.

// Most of the constructors do not perform sign checks on their
// arguments. The reason is simple: Unlike the matrix or term ordering
// constructors, some binomial constructors are frequently called
// during computation. As the new binomials are build from the input
// binomials, it is sufficient to do these checks on the input.

#ifndef BINOMIAL_H
#define BINOMIAL_H

class term_ordering;

class binomial
{
private:

  Integer* exponent_vector;

  short _number_of_variables;

#ifdef SUPPORT_DRIVEN_METHODS

  unsigned long head_support;
  unsigned long tail_support;

#endif  // SUPPORT_DRIVEN_METHODS

public:

// constructors and destructor

  binomial(const short& number_of_variables);
```

```
  // Allocates memory for a binomial of size number_of_variables
  // without initialization.
  // No range check on the argument!

  binomial(const short& number_of_variables, const Integer* exponents);
  binomial(const short& number_of_variables, const Integer* exponents,
           const term_ordering&);
  // These constructors build a binomial from its exponent
  // vector/array.
  // The first one takes the positive components as its head, the
  // negative ones as its tail; the second one determines its head and
  // tail with respect to the given term ordering.
  // The array "exponents" is always copied. It would be faster to set
  // only pointers on it. However, this is very dangerous, and as
  // these constructors are only used at the beginning of the
  // algorithm, it would not really improve performance.
  // The Integer pointer exponents is declared as "const" to suggest
  // that the referenced array is not changed (although the
  // "const"-declaration does not assert this).
  // These constructors check the range (sign) of their argument
  // number_of_variables because they are not frequently called, but
  // at a critical point (ideal constructors).

  binomial(const binomial&);
  // copy-constructor
  // No check if the copied binomial is corrupt!

  ~binomial();
  // destructor

// object information

  short number_of_variables() const;
  // Returns the number of variables.

  short error_status() const;
  // Returns _number_of_variables iff _number_of_variables<0, else 0
  // (this is the "errror flag").

// assigment and access operators

  binomial& operator=(const binomial&);
  // assignment operator with memory control

  Integer operator[](const short& i) const;
  // access operator for reading exponent_vector[i]
  // (cannot be used to overwrite exponent_vector[i])
```

```
  // no range check on i!

// comparison operators

  BOOLEAN operator==(const binomial& v) const;
  BOOLEAN operator!=(const binomial& v) const;
  // comparison of binomials

  BOOLEAN operator==(const Integer comp_value) const;
  BOOLEAN operator!=(const Integer comp_value) const;
  BOOLEAN operator<=(const Integer comp_value) const;
  BOOLEAN operator>=(const Integer comp_value) const;
  // operators for an efficient comparison with the zero binomial
  // (comp_value=0)

// basic routines for Buchbergers's algorithm

  Integer head_reductions_by(const binomial& b) const;
  // Returns the number of possible reductions of the actual
  // binomial's head by the binomial b.
  // The return value -1 means that b==0 or head(b)==1; one should not
  // try to reduce by such a binomial (reduction is undefined or does
  // not terminate). The procedure reduce_tail_by(...) returns the
  // unchanged binomial in such a case without a warning. This is done
  // in view of the application: If a generator of an ideal is reduced
  // to zero during a Groebner basis calculation and one forgets to
  // delete it, this generator is ignored (else it would probably
  // cause a fatal error).

  Integer tail_reductions_by(const binomial& b) const;
  // Returns the number of possible reductions of the actual
  // binomial's tail by the binomial b.
  // For the return value -1 see above.

  int reduce_head_by(const binomial& b, const term_ordering& w);
  // Performs a multiple reduction of the actual binomial's head by
  // the binomial b and computes the new head and tail with respect to
  // the term ordering w.
  // The return value is 1 if the binomial has really been reduced, 2
  // if the binomial has been reduced to zero, 0 if the binomial has
  // not been changed.

  int reduce_tail_by(const binomial& b, const term_ordering& w);
  // Performs a multiple reduction of the actual binomial's tail by
  // the binomial b and computes the new head and tail with respect to
  // the term ordering w.
  // The return value is 1 if the binomial has really been reduced,
```

```
  // else 0. (By a tail reduction, the binomial cannot be reduced to
  // zero if the head and the tail of the binomial itself and of b are
  // correct with respect to w.)

  friend binomial& S_binomial(const binomial& a, const binomial& b,
                              const term_ordering& w);
  // Computes the S-binomial of a and b with respect to the term
  // ordering w and returns a reference on it (the necessary memory is
  // allocated during the computation).

// criteria for unnecessary S-Pairs

  friend BOOLEAN relatively_prime(const binomial& a,
                                  const binomial& b);
  // Returns TRUE iff the leading terms of a and b are relatively
  // prime.

  friend BOOLEAN M(const binomial& a, const binomial& b,
                   const binomial& c);
  // Verifies if lcm(head(a),head(c)) divides properly
  // lcm(head(b),head(c)).

  friend BOOLEAN F(const binomial& a, const binomial& b,
                   const binomial& c);
  // Verifies if lcm(head(a),head(c))=lcm(head(b),head(c)).

  friend BOOLEAN B(const binomial& a, const binomial& b,
                   const binomial& c);
  // Verifies if head(a) divides lcm(head(b),head(c)) and
  // lcm(head(a),head(b))!=lcm(head(b),head(c))!=lcm(head(a),head(c)).

  friend BOOLEAN second_crit(const binomial& a, const binomial& b,
                             const binomial& c);
  // Verifies if head(a) divides lcm(head(b),head(c)).

// special routines needed by the IP-algorithms

// All this routines are not very frequently used (especially not in
// Buchberger's algorithm), so I haven't spent much time in
// optimization. None of them performs range checks on their
// arguments.

  BOOLEAN involves_elimination_variables(const term_ordering& w);
  // Verifies if the binomial involves elimination variables with
  // respect to w.
  // UNCHECKED PRECONDITION: w and the binomial are compatible, i.e.
  // involve the same number of variables.
```

```
  binomial& drop_elimination_variables(const term_ordering& w);
  // Cuts the elimination variables (with respect to w) from the
  // binomial and adapts the member _number_of_variables as well as
  // the head and the tail.
  // UNCHECKED PRECONDITION: w and the binomial are compatible.
  // The interesting part of the binomial (ot its additive inverse) is
  // copied hereby to be able to free the memory needed by the dropped
  // components.

  binomial& drop_last_weighted_variable(const term_ordering& w);
  // Cuts the last variable from the binomial and adapts the member
  // _number_of_variables as well as head and tail to the given
  // term_ordering.
  // UNCHECKED PRECONDITION: w is a weighted termordering (without
  // elimination variables) that is compatible with the actual
  // binomial.
  // The interesting part of the binomial (ot its additive inverse) is
  // copied hereby to be able to free the memory needed by the dropped
  // component.

  int adapt_to_term_ordering(const term_ordering&);
  // Determines head and tail of the binomial with respect to the
  // given term ordering; i.e. multiplies the exponent vector with -1
  // and exchanges head_support and tail_support, if necessary.
  // Returns -1 if head and tail were exchanged, 1 else.
  // UNCHECKED PRECONDITION: w and the binomial are compatible.

  binomial& swap_variables(const short& i,const short& j);
  // Swaps the components i and j of the exponent vector and
  // actualizes head_support and tail_support.
  // No range check on the arguments!

  binomial& flip_variable(const short& i);
  // Inverts component i of the exponent vector and actualizes
  // head_support and tail_support.
  // No range check on i!

// output

  void print() const;
  void print_all() const;
  // Writes the actual binomial to the standard output medium.
  // The first routine only prints the exponent vector, the second
  // prints the head and tail support, too (if SUPPORT_DRIVEN_METHODS
  // are used).
```

```
  void print(FILE* output) const;
  void print_all(FILE* output) const;
  // Writes the actual binomial to the referenced file which must have
  // been opened for writing before.

  void print(ofstream&) const;
  void print_all(ofstream&) const;
  // Writes the actual binomial to the given ofstream.

  void format_print(ofstream&) const;
  // Writes the given binomial to the actual ofstream in the format
  // needed by the IP-algorithms.

  friend class ideal;
  // The class ideal is declared as a friend for efficiency reasons:
  // This avoids many unnecessary function calls for inspecting
  // members like head_support and tail_support.
  // The class ideal does not change any members of the binomial
  // class.

  friend short term_ordering::compare(const binomial&,
                                      const binomial&) const;
  // This function is declared as a friend for efficiency reasons to
  // avoid  calls of the access operator [].
};
#endif  // BINOMIAL_H
```

A.1.3 binomial__term_ordering.h

```
// binomial__term_ordering.h

// The class binomial needs to know the class term_ordering and vice
// versa. But for a  beautiful design, we did not want to write both
// declarations into the same file. This file simply puts the two
// corresponding header files together.

#ifndef BINOMIAL__TERM_ORDERING_H
#define BINONIAL__TERM_ORDERING_H

#include "term_ordering.h"
#include "binomial.h"

#endif  // BINOMIAL__TERM_ORDERING_H
```

A.1.4 globals.h

```
// globals.h
```

```
// In this file
// - some facilities as the standard input/output routines are
//   included,
// - the general data type can be chosen (the people of Singular do
//   not like templates),
// - some makros are defined for an easier handling of the program,
// - some flags and constants can be set to observe the behaviour of
//   the algorithms when using different strategies.

#ifndef GLOBALS_H
#define GLOBALS_H

// Include facilities needed by several files:

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stddef.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <string.h>

// Include a BigInt class from a foreign library:
// This is needed by the LLL algorithm. If no such class is included,
// it computes with long ints; this results in overflows and
// unexpected behaviour even when computing relatively small
// instances.

#define GMP

#ifdef GMP

#include "BigInt.h"

#else   // GMP

#ifdef LEDA

#include <LEDA/integer.h>
#include <LEDA/rational.h>

typedef integer BigInt;

#else   // LEDA
```

```
typedef long BigInt;

#endif  // LEDA

#endif // GMP

// Define the general data type:
// short seems to be sufficient for most practicable instances, but
// not when using the algorithm of Di Biase and Urbanke

typedef long Integer;
// The general data type to deal with can be short, long or int...

#define _LONG_
// For an overflow control for the result of the LLL algorithm, we
// have to know the data type used.
// Possibilities: _SHORT_, _INT_, _LONG_

// Define a BOOLEAN data type etc. for an easy-to-read code:

typedef char BOOLEAN;
#define TRUE  1
#define FALSE 0

#define MAXIMUM(a,b) (a>b? a:b)

#define INHOMOGENEOUS 0
#define HOMOGENEOUS   1
// For a more comfortable call of the term_ordering constructors.

// Enumerate the algorithms:
// The ideal constructor which builds an ideal from the input matrix
// depends on the algorithm. The constants are defined to make
// constructor calls easier (you do not need to remember a number).

#define CONTI_TRAVERSO           1
#define POSITIVE_CONTI_TRAVERSO  2
#define POTTIER                  3
#define HOSTEN_STURMFELS         4
#define DIBIASE_URBANKE          5
#define BIGATTI_LASCALA_ROBBIANO 6

// Enumerate the criteria to discard unnecessary S-pairs:
// The constants are defined to allow an easier call of Buchberger's
// algorithm with different criteria. This algorithm takes as an
// argument a short int which represents the combination of criteria
```

```
// to be used (with a default combination if no argument is
// given). The method to compute the argument for a certain
// combination is simple: Simply add the constants that belong to
// criteria you want to use. As the constants are chosen to be
// different powers of 2, each short int (in the range of 0 to 31)
// gives a unique combination (cf. the read-write-execute rights for
// files in UNIX systems).
// EXAMPLE:
// If you want to call Buchberger's algorithm with the criterion
// "relatively prime leading terms" and the second criterion for the
// ideal I, write:
//
//      I.reduced_Groebner_basis_1(REL_PRIMENESS + SECOND_CRITERION);
// or
//      I.reduced_Groebner_basis_1(17);
//
// The argument 0 means that no criteria are used, the argument 31
// leads to the use of all criteria.

#define REL_PRIMENESS      1
#define M_CRITERION        2
#define F_CRITERION        4
#define B_CRITERION        8
#define SECOND_CRITERION  16

// The names of these criteria are chosen according to the criteria
// described in Gebauer/Moeller (except from the last). The first one
// (relatively prime leading terms) is a local criterion and the most
// effective.
// The last four criteria are global ones involving searches over the
// generator lists. But although the lcm-computations performed by
// these checks are not much cheaper than a reduction, most of the
// criteria seem to accelerate the computation.
// REL_PRIMENESS should always be used.
// The Gebauer/Moeller-criteria (M,F,B) seem to improve computation
// time in the general case; Buchberger's second criterion seems to
// improve it in more special cases (for example the transportation
// problem). They should be switched on and off according to these
// results, but never be enabled together; the overhead of too many
// criteria leads to a rather bad performance in all cases.
// The relative primeness criterion is verified first because it is
// very easy to check. The order of the Gebauer/Moeller-criteria does
// not really affect computation. The relative order of the
// Gebauer/Moeller criteria and the second criterion (if they are
// enabled together) has almost the same effect as switching off the
// last criterion in this order.
// There is no possibility to change the order in which the criteria
```

```
// are tested without changing the program!

// Enumerate the term orderings that can be used to refine the weight:
// W_REV_LEX will give a term ordering in the classical sense if and
// only if all weights are positive. It has been implemented because
// several algorithms need a reverse lexicographical refinement. (Of
// course, these algorithms control if they have a positive grading.)
// A warning will be output when using W_REV_LEX in an insecure case.

#define W_LEX          4
#define W_REV_LEX      5
#define W_DEG_LEX      6
#define W_DEG_REV_LEX  7

// Enumerate the term ordering for the block of elimination variables:

#define LEX          1
#define DEG_LEX      2
#define DEG_REV_LEX  3

//////////////////////////////////////////////////////////////////////
///////////////     TEST PARAMETER SECTION    ////////////////////////
//////////////////////////////////////////////////////////////////////

// Set flags concerning the basic operations and the course of
// Buchberger's algorithm:

#define SUPPORT_DRIVEN_METHODS
// possibilities:
// SUPPORT_DRIVEN_METHODS
// NO_SUPPORT_DRIVEN_METHODS

// This flag allows to switch of and on the use of the support vectors
// in the binomial class; these are used to speed up tests for
// reducibility of binomials, relative primeness etc.

#define SUPPORT_DRIVEN_METHODS_EXTENDED
// possibilities:
// SUPPORT_DRIVEN_METHODS_EXTENDED
// NO_SUPPORT_DRIVEN_METHODS_EXTENDED

// This flag allows to switch of and on the extended use of the
// support information. This includes the splitting of the ideal
// generators into several lists according to their head support. This
// discards many unnecessary tests for divisibility etc.
// SUPPORT_DRIVEN_METHODS_EXTENDED should only be enabled when
// SUPPORT_DRIVEN_METHODS is set! The combination
```

```
// NO_SUPPORT_DRIVEN_METHODS and  SUPPORT_DRIVEN_METHODS_EXTENDED is
// quite senseless and will only work if your compiler automatically
// initializes integers to zero.

const unsigned short List_Support_Variables=8;

// This is the number of variables considered to create and maintain
// the generator lists of an ideal if SUPPORT_DRIVEN_METHODS_EXTENDED
// is enabled. As there are 2^List_Support_Variables lists, this
// number should not be too big (to avoid the overhead of many empty
// lists).  The best number may depend on the problem size and
// structure...

#define SUPPORT_VARIABLES_FIRST
// possibilities:
// SUPPORT_VARIABLES_LAST
// SUPPORT_VARIABLES_FIRST

// This flag determines whether the first or the last variables are
// considered  by the support methods. So this setting will only
// affect the computation if such methods are enabled.
// The reason for the introduction of this flag is the following:
// The Conti-Traverso algorithm for solving an integer program works
// in two phases: Given an artificial solution in auxiliary variables,
// these variables are first eliminated to get a feasible
// solution. The elimination variables are for technical reasons
// always the last. In a second phase, this solution is reduced to a
// minimal solution with respect to a given term ordering.
// If SUPPORT_VARIABLES_LAST is set, the first phase will probably
// take less time as if SUPPORT_VARIABLES_FIRST is set. If one is only
// interested in finding a feasible solution (not necessary  minimal),
// SUPPORT_VARIABLES_LAST might be the better choice.

#endif  // GLOBALS_H
```

A.1.5 ideal.h

```
// ideal.h

// This class implements a saturated binomial ideal. Such an ideal is
// given by its generators.
// If SUPPORT_DRIVEN_METHODS_EXTENDED are enabled, the generators are
// not stored in a single list, but classified according to their
// support. This representation, together with the subset_tree data
// structure, allows a faster search for reducers.

// The entries of the two-dimensional array "subsets_of_support" are
```

```
// to be read as bit vectors with List_Support_Variables components
// (the constant List_Support_Variables is defined in globals.h). So
// all entries are in the range between 0 and 2^List_Support_Variables
// -1. subsets_of_support[i] is an array that contains all binary
// vectors whose support is a subset of that of i, i read as a binary
// vector. number_of_subsets[i] is the number of these subsets,
// i.e. the length of the array subsets_of_support[i].

// If, for example, List_Support_Variables=8, the datastructure
// subset_tree has 6561=8^3 entries.

#ifndef IDEAL_H
#define IDEAL_H

#include "list.h"
#include "matrix.h"

const short Number_of_Lists=1<<List_Support_Variables;
// =2^List_Support_Variables

///////////////////// struct subset_tree /////////////////////////

typedef struct
{
  short* subsets_of_support[Number_of_Lists];
  short number_of_subsets[Number_of_Lists];
} subset_tree;

/////////////////////// class ideal //////////////////////////////

class ideal
{
private:

// generator lists

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  subset_tree S;

  list generators[Number_of_Lists];

  list new_generators[Number_of_Lists];
  // Only needed in some special versions of Buchberger's algorithm to
  // store newly found generators.

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED
```

```
#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  list generators;

  list new_generators;
  // Only needed in some special versions of Buchberger's algorithm to
  // store newly found generators.

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

// flags for the use of the S-pair criteria and the autoreduction
// (=interreduction)

  short rel_primeness;
  short M_criterion;
  short F_criterion;
  short B_criterion;
  short second_criterion;
  // When Buchberger's algorithm is called, we only use one argument
  // which describes the combination of the criteria to be used (see
  // in globals.h). But we use five flags instead of one here because
  // this is a little more efficient.
  // The standard setting enables the relative primeness criterion,
  // the M- and the B-criterion.

  float interreduction_percentage;
  // To determine the autoreduction frequency, i.e. the percentage of
  // new generators (i.e. generators found since the last
  // autoreduction) with respect to the total size of an ideal that
  // must be reached to cause an interreduction.
  // If Interreduction_Percentage==0, interreduction is done after
  // each S-Pair computation. If Interreduction_Percentage<0,
  // interreduction is only done once at the end of the algorithm.
  // The standard setting is 12 (=12%).

// further members

  term_ordering w;
  // For technical reasons, the term ordering is taken as a member.
  // So we do not need to pass it as an argument to each ideal
  // function, and the management of the generator lists is easier and
  // safer when using SUPPORT_DRIVEN_METHODS_EXTENDED.

  list aux_list;
  // an auxiliary list for keeping (for example) S-pairs
```

```
  long size;
  // the actual number of generators of the ideal
  // Also used as an "error flag"; a negative size means that an error
  // has occurred:
  // -1 indicates a "semantic" error (which occurs e.g. if some
  //    constructor argument is out of range)
  // -2 indicates an error occured when reading from a file
  // -3 indicates an overflow of an integer type variable

  long number_of_new_binomials;
  // the number of newly found generators

// private member functions

// subroutines for building and deleting the subset_tree data structure
// (implemented in ideal.cc)

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  create_subset_tree();
  destroy_subset_tree();

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

// subroutines for Buchberger's algorithm (implemented in
// Buchberger.cc, except from the first two that are implemeted in
// ideal.cc)

// Some of these procedures do not interact with all versions of
// Buchberger's algorithm. Generally, they cannot be combined as one
// likes. This header file only gives a brief overview of their
// features. For more detailed comments see the implementation.

  ideal& add_new_generator(binomial& bin);
  ideal& add_generator(binomial& bin);
  // Inserts a (new) generator in the appropriate list.

  short add_new_generators();
  // Moves the new generators to the generator lists.

// S-pair computation

  BOOLEAN unnecessary_S_pair(list_iterator&, list_iterator&) const;
  // Checks if the S-pair of the binomials referenced by the iterators
  // can be discarded (by the chosen criteria).

  ideal& compute_actual_S_pairs_1();
```

```
  ideal& compute_actual_S_pairs_1a();
  ideal& compute_actual_S_pairs_2();
  ideal& compute_actual_S_pairs_3();
  // different versions for computing the S-binomials of the actual
  // generators
  // They differ for example in the following points:
  // - They insert the S-binomials in the order in which they are
  //   computed, in the order given by the term ordering w or
  //   according to their support.
  // - Some reduce an S-binomial immediately after its computation by
  //   the ideal generators, others reduce it later (after having
  //   computed more S-binomials, hence more possible reducers).
  // - Some reduce the ideal generators immediately by a newly
  //   computed S-binomial, others don't.
  // In each case, the use of the list flag "done" guarantees that we
  // perform the S-pair computation only once for each binomial pair.

// minimalization / autoreduction

  ideal& reduce_by(const binomial&, list_iterator&, list_iterator&);
  // Reduces the heads of the ideal generators by the given binomial
  // (used by some versions of the S-pair computation).

  ideal& minimalize_S_pairs();
  ideal& minimalize_new_generators();
  ideal& minimalize();
  // Performs an autoreduction of the binomials stored in aux_list
  // respectively in the list(s) new_generators respectively in the
  // generators.
  // The respective list(s) is (are) reduced to a minimal set (not
  // necessarily to a reduced set); this means that only the heads of
  // the binomials are reduced, not their tails. This strategy showed
  // to be a little more efficient than a complete autoreduction.
  // The last two procedures use the "head_reduced" flag of the list
  // class to avoid unnecessary tests for interreduction.

  ideal& final_reduce();
  // final interreduction of the ideal generators
  // It seems to be slightly more efficient to perform a complete
  // interreduction only at the end of Buchberger's algorithm and
  // to replace the intermediate interreductions by minimalizations.
  // For efficiency reasons, this routine is designed for reducing a
  // Groebner basis of an saturated ideal. Reducing another generating
  // set with it may cause inconsistencies (cf. comment in the
  // implementation).

// constructor subroutines for the handling of the different
```

```
// algorithms (implemented in ideal.cc)
// The "Conti-Traverso constructors" create a toric ideal directly
// from the input matrix and the term ordering given by the input
// weight vector. The other constructors create a toric ideal from a
// lattice basis of the kernel of the input matrix.

  ideal& Conti_Traverso_ideal(matrix&, const term_ordering&);
  ideal& Positive_Conti_Traverso_ideal(matrix&, const term_ordering&);
  ideal& Pottier_ideal(matrix&, const term_ordering&);
  ideal& Hosten_Sturmfels_ideal(matrix&, const term_ordering&);
  ideal& DiBiase_Urbanke_ideal(matrix&, const term_ordering&);
  ideal& Bigatti_LaScala_Robbiano_ideal(matrix&, const term_ordering&);

public:

// constructors and destructor (implemented in ideal.cc)

  ideal(matrix&, const term_ordering&, const short& algorithm);
  // Creates a binomial ideal from the given matrix using the given
  // algorithm (see in globals.h).
  // The arguments are checked for consistency as far as possible.
  // The term ordering is needed to determine the leading terms of the
  // binomials in the resulting generating set. Neither is the
  // generated ideal saturated nor is it given in form of a Groebner
  // basis: Such computations must be explicitly demanded.
  // The argument matrix cannot be declared as const because the
  // constructor may call the LLL-algorithm to compute the matrix
  // kernel (if this hasn't been done yet). This algorithm changes
  // some matrix members. But the real matrix remains, of course,
  // unchanged.

  ideal(const ideal&);
  // copy-constructor
  // It might be useful to keep several Groebner bases of the same
  // ideal (or of an ideal and some of its elimination ideals).

  ideal(ifstream&, const term_ordering&,
        const short& number_of_generators);
  // Reads an ideal from a given ifstream in the following way:
  // A block of integers is converted into a binomial
  // that is stored in the generator list(s) with respect to the given
  // term ordering. The size of such a block is the size of the given
  // term_ordering, i.e. the number of variables for which it was
  // constructed. This is done number_of_generators times.

  ~ideal();
  // destructor
```

```
// object information (implemented in ideal.cc)

  long number_of_generators() const;
  // Returns the actual number of generators.

  short error_status() const;
  // Returns -1 if an error has occurred (i.e. size<0), else 0.

// Buchberger stuff (implemented in Buchberger.cc)

  ideal& reduced_Groebner_basis_1(const short& S_pair_criteria=11,
                             const float& interred_percentage=12.0);
  ideal& reduced_Groebner_basis_1a(const short& S_pair_criteria=11,
                              const float& interred_percentage=12.0);
  ideal& reduced_Groebner_basis_2(const short& S_pair_criteria=11,
                              const float& interred_percentage=12.0);
  ideal& reduced_Groebner_basis_3(const short& S_pair_criteria=11,
                              const float& interred_percentage=12.0);
  // Several different versions of Buchberger's algorithm for
  // computing the reduced Groebner basis of the actual ideal. They
  // differ in the steering of the algorithm (i.e. in the way in which
  // the S-pair computation and the reduction is organized).
  // Except from the variant 1a (which orders the S-pairs with respect
  // to w and which shows to be quite slow), the running times of
  // these algorithms do not differ too much.
  // The first argument indicates which criteria will be used to
  // discard unnecessary S-pairs. For an explaination of how this
  // works, see in globals.h. The default value 11 means that we use
  // the criterion of relatively prime leading terms as well as the M-
  // and the B-criterion.
  // The second argument specifies the interreduction frequency; see
  // the comment for the member interreduction_percentage above. The
  // standard value is 12%.
  // ATTENTION: In spite of the different argument types you should
  // never try to use these functions with one default argument
  // (either two or none). When writing
  //    reduced_Groebner_basis_1(10.5),
  // the GNU C++ compiler casts 10.5 into an integer and takes it as
  // the argument S_pair_criteria!
  // ATTENTION: If the input ideal is not saturated, the computed
  // reduced Groebner basis is not that of the input ideal, but that
  // of an ideal "between" the input ideal and its saturation.

  ideal& reduced_Groebner_basis(const short& version=1,
                         const short& S_pair_criteria=11,
                     const float& interred_percentage=12.0);
```

```
  // Takes the version of Buchberger's algorithm as above as an
  // argument (to allow a call of different versions of our
  // IP-algorithms from the command line). To call version 1a, the
  // first argument has to be zero.

  binomial& reduce(binomial& bin, BOOLEAN complete=TRUE) const;
  // Reduces a binomial by the actual generators.
  // To reduce a binomial by the ideal (outside Buchberger's
  // algorithm), be sure to have computed the reduced Groebner basis
  // before.
  // The flag "complete" indicates if the binomial is reduced
  // completely (head and tail); if it is set to FALSE, only the head
  // will be reduced. Using this flag in a clever manner allows to
  // improve the performance of Buchberger's algorithm by up to three
  // percent.

// special features needed by our IP-algorithms
// (implemented in ideal_stuff.cc)

  ideal& eliminate();
  // Eliminates the generators involving elimination variables
  // (with respect to the term ordering w).
  // The reduced Groebner basis with respect to w should have been
  // computed before calling this routine.

  ideal& pseudo_eliminate();
  // Eliminates the generators involving the last weighted variable -
  // a special routine needed by the algorithm of Bigatti, La Scala
  // and Robbiano.

  ideal& change_term_ordering_to(const term_ordering& _w);
  // Replaces the actual term ordering by the argument ordering.
  // Their "compatibility" (number of variables) is checked, head and
  // tail of the generators are adapted to the new ordering.
  // This may especially involve a rebuilding of the list structure if
  // SUPPORT_DRIVEN_METHODS_EXTENDED are enabled.

  ideal& swap_variables(const short& i, const short& j);
  // Swaps the i-th and the j-th variable in all generators as well as
  // the corresponding components of the term ordering's weight
  // vector.
  // If SUPPORT_DRIVEN_METHODS are enabled, the list structure is
  // rebuilt according to the new order on the variables.

  ideal& swap_variables_unsafe(const short& i, const short& j);
  // Swaps the i-th and the j-th variable in all generators as well as
  // the corresponding components of the term ordering's weight
```

```
  // vector.
  // DANGER: The head/tail structure is not rebuilt!

  ideal& flip_variable_unsafe(const short& i);
  // Inverts the sign of the i-th variable in all generators.
  // DANGER: The list structure is not rebuilt!

// output (implemented in ideal.cc)

  void print() const;
  void print_all() const;
  // Writes the ideal to the standard output medium.
  // The second routine also prints the S-pair flags.

  void print(FILE*) const;
  void print_all(FILE*) const;
  // Writes the ideal to the referenced file which has to be opened
  // for writing before.

  void print(ofstream&) const;
  void print_all(ofstream &) const;
  // Writes the ideal to the given ofstream.

  void format_print(ofstream&) const;
  // Writes the ideal generators to the given ofstream in a format
  // that allows them to be reread by the ideal constructor from an
  // ifstream.
};
#endif  // IDEAL_H
```

A.1.6 IP_algorithms.h

```
// IP_algorithms.h

#ifndef IP_ALGORITHMS_H
#define IP_ALGORITHMS_H

#include "ideal.h"

typedef char* INPUT_FILE;
typedef char* OUTPUT_FILE;

// This file contains the declaration of all implemented
// IP-algorithms.
// They only solve problems for nonnegative weight vectors.

// Let A an integral (mxn)-matrix, b a vector with m integral
```

```
// coefficients and c a vector with n nonnegative real coefficients.
// The solution of the IP-problem
//
//    minimize cx
//    under the conditions
//             Ax=b and all components of x are nonnegative integers
//
// proceeds in two steps:

// First, we have to compute the toric ideal of A and its Groebner
// basis with respect to a term ordering refining the cost function
// c. This can be done by the following procedures. They check the
// format of their input file (which should be a MATRIX file as
// described below) and return 1 if they were successful, 0 else.
//  They take as arguments
// - their input file;
// - the arguments for Buchberger's algorithm (see the comments in
//   ideal.h);
// - the output mode (verbose or not, default: not verbose; verbose
//   mode prints compiler settings and options for Buchberger's
//   algorithm).
// The last four arguments have default values and do not have to be
// specified.

// In the case of a successful computation, the procedures
// write the computed Groebner basis into a GROEBNER file which
// is named like the input file with extension replaced by
//      .GB.<alg>
// where <alg> is an abbreviation for the used algorithm:
//         ct     for Conti_Traverso(...)
//        pct     for Positive_Conti_Traverso(...)
//        ect     for Elim_Conti_Traverso(...)
//         pt     for Pottier(...)
//         hs     for Hosten_Sturmfels(...)
//         du     for DiBiase_Urbanke(...)
//        blr     for Bigatti_LaScala_Robbiano(...)

extern int Conti_Traverso(INPUT_FILE MATRIX,
                          const short& version=1,
                          const short& S_pair_criteria=11,
                          const float& interred_percentage=12.0,
                          const BOOLEAN& verbose=FALSE);
// The complete algorithm of Conti and Traverso which computes the
// toric ideal of an extended matrix and which can be used for solving
// integer programs without initial solution.

extern int Positive_Conti_Traverso(INPUT_FILE MATRIX,
```

```
                             const short& version=1,
                             const short& S_pair_criteria=11,
                             const float& interred_percentage=12.0,
                             const BOOLEAN& verbose=FALSE);
// The variant of the above algorithm for matrices and right-hand
// vectors with only nonnegative entries using one elimination
// variable less.

extern int Elim_Conti_Traverso(INPUT_FILE MATRIX,
                               const short& version=1,
                               const short& S_pair_criteria=11,
                               const float& interred_percentage=12.0,
                               const BOOLEAN& verbose=FALSE);
// A version of the Conti-Traverso-algorithm which really computes the
// toric ideal of A. Used for test purposes (correctness and
// performance of the other algorithms).

extern int Pottier(INPUT_FILE MATRIX,
                   const short& version=1,
                   const short& S_pair_criteria=11,
                   const float& interred_percentage=12.0,
                   const BOOLEAN& verbose=FALSE);
// The algorithm proposed by Pottier using one elimination variable
// and a lattice basis of the matrix kernel to compute the toric ideal
// of A.

extern int Hosten_Sturmfels(INPUT_FILE MATRIX,
                            const short& version=1,
                            const short& S_pair_criteria=11,
                            const float& interred_percentage=12.0,
                            const BOOLEAN& verbose=FALSE);
// The algorithm proposed by Hosten and Sturmfels which computes the
// toric ideal of A from a kernel lattice basis without supplementary
// variables, but with various Groebner basis calculations.

extern int DiBiase_Urbanke(INPUT_FILE MATRIX,
                           const short& version=1,
                           const short& S_pair_criteria=11,
                           const float& interred_percentage=12.0,
                           const BOOLEAN& verbose=FALSE);
// The algorithm proposed by DiBiase and Urbanke which computes the
// toric ideal of A from a kernel lattice basis using "variable
// flips".

extern int Bigatti_LaScala_Robbiano(INPUT_FILE MATRIX,
                              const short& version=1,
                              const short& S_pair_criteria=11,
```

```
                              const float& interred_percentage=12.0,
                              const BOOLEAN& verbose=FALSE);
// A modified version of the algorithm called EATI using
// "pseudo-elimination". This algorithm is quite similar to Pottier's
// algorithm, but deals with homogenous binomials.

// The second step of the IP solution is to reduce one or more given
// initial solutions / right-hand vectors with respect to the computed
// Groebner basis. In the case that the Groebner basis
// was computed via the algorithm of Conti and Traverso, it is
// sufficient to give the right-hand vectors. In all other cases we
// need explicit initial solutions. The routine

extern int solve(INPUT_FILE PROBLEM, INPUT_FILE GROEBNER);

// solves the IP-problems given by the vectors in PROBLEM with respect
// to GROEBNER which should contain a Groebner basis as computed
// above. The output is appended to a (eventually newly created) file
// named like GROEBNER with extensions replaced by
//      .sol.<alg>
// where <alg> is an abbreviation of the algorithm used to compute
// GROEBNER.

// We also provide functionality to recompute toric ideals with
// respect to another cost function:

extern int change_cost(INPUT_FILE GROEBNER, INPUT_FILE NEW_COST,
                       const short& version=1,
                       const short& S_pair_criteria=11,
                       const float& interred_percentage=12.0,
                       const BOOLEAN& verbose=FALSE);

// recomputes the Groebner basis in GROEBNER with respect to the cost
// given in NEW_COST and writes the result into the file named as
// NEW_COST with extension replaced by
//      .GB.<alg>
// where <alg> is an abbreviation of the algorithm used to compute
// GROEBNER.

//////////////////////////////////////////////////////////////////////
//////////      FORMAT OF INPUT AND OUTPUT FILES     /////////////////
//////////////////////////////////////////////////////////////////////

// The files appearing under the name MATRIX in the above declarations
// should look as follows to be accepted by the algorithms:

//      MATRIX                                (* specifie format *)
```

```
//
//      columns:
//      <number of columns>
//
//      cost vector:
//      <coefficients of the cost vector>
//
//      rows:
//      <number of rows>
//
//      matrix:
//      <matrix coefficients>
//
//      positive row space vector:          (* optional, see below *)
//      <coefficients of row space vector>

// For example:

//      MATRIX
//
//      columns:
//      3
//
//      cost vector:
//      1 1 1
//
//      rows:
//      2
//
//      matrix:
//      1 2 3
//      4 5 6
//
//      positive row space vector:
//      1 2 3

// The positive row space vector is only needed by the algorithms of
// Hosten/Sturmfels and Bigatti/LaScala/Robbiano. It is ignored by the
// other algorithms, as well as further lines in the input file. This
// allows us to use the same input format for all algorithms. The
// comments (as the word "rows:") are only written into the file to
// make it easy to read.

// The file named NEW_COST in the change_cost procedure should have
// the following format:

//      NEW_COST
```

```
//
//      variables:                          (* only weighted variables *)
//      <number of weighted variables>
//
//      cost vector:
//      <coefficients of the weight vector>

// For convenience, the MATRIX format is also accepted by the
// change_cost procedure. The lines following the cost vector are then
// ignored.

// The files appearing under the name GROEBNER in the above
// declarations are created in the following form by the algorithms
// that compute toric ideals. This form is accepted by the solve(...)
// and the change_cost(...) procedures:

//      GROEBNER                            (* specifie format *)
//
//      computed with algorithm:
//      <algorithm name abbreviation>       (* abbreviations as above *)
//      from file(s):                       (* the following four lines are
//      <name of respective MATRIX file>       optional *)
//      computation time:
//      <computation time in seconds>
//
//      term ordering:
//      elimination block
//      <number of elimination variables>
//      <LEX / DEG_LEX                      (* only if number of elimination
//      / DEG_REV_LEX>                         variables >0 *)
//      weighted block
//      <number of weighted variables>
//      <W_LEX / W_REV_LEX                  (* number of weighted variables
//      / W_DEG_LEX / W_DEG_REV_LEX>           should always be >0 *)
//      <weight_vector>
//
//      size:
//      <number of elements>
//
//      Groebner basis:
//      <basis elements>                    (* one basis element in each row,
//                                             given by the coefficients of
//                                             its vector representation *)
//      settings for the                    (* all following lines are
//      Buchberger algorithm:                  optional, verbose mode *)
//      version:                            (* as explained in ideal.h,
//      <version number>                       between 0 and 3 *)
```

```
//      S-pair criteria:
//      <{relatively prime leading terms,(* a subset of these criteria *)
//        criterion M, criterion F,
//        criterion B, second criterion}>
//      interreduction percentage:
//      <interreduction percentage>
//
//      compiler settings:              (* see the procedure
//      ...                                print_flags(...) in
//      ...                                IP_algorithms.cc *)

// If the GROEBNER file is created by change_cost(...), the algorithm
// name and MATRIX file are copied from the original GROEBNER file;
// the new_cost file is specified as a second MATRIX file.

// The files appearing under the name PROBLEM in the above
// declarations should look as follows to be accepted by the procedure
// solve(...):

//      PROBLEM                                 (* specifie format *)
//
//      vector size:
//      <vector dimension>
//
//      number of instances:
//      <number of vectors>
//
//      right-hand or initial solution vectors:
//      <vector coefficients>                   (* one vector per row *)


// solve(...) verifies if the vector size given in the PROBLEM file
// and the number of variables given in the GROEBNER file match the
// respective algorithm. In the matching case, it creates a SOLUTION
// file of the following format (not used as an input file for any
// procedure):

//      SOLUTION                              (* specifie format *)
//
//      computed with algorithm:              (* from the respective
//      <algorithm name abbreviation>            GROEBNER file *)
//      from file:                            (* the GROEBNER file *)
//      <file name>
//
//      <right-hand/initial solution> vector:
//      <vector as in the problem file>
//      solvable:                             (* only if right-hand
```

```
//      <YES/NO>                                    vector is given *)
//      optimal solution:                        (* only in the case of
//      <optimal solution vector>                   existence *)
//      value:                                   (* only in the case of
//      <optimal cost>                              existence *)
//      computation time:                        (* only reduction time *)
//      <reduction time in seconds>
//
//      ...                                      (* further vectors *)

#endif
```

A.1.7 list.h

```
// list.h

// The classes list and list_iterator are designed for the special
// needs of Buchberger's algorithm; it is not recommendable to use
// them for more general purposes.
// class list implements a linked list of binomials (or, more exactly,
// pointers on binomials).
// A list_iterator is not much more than a pointer to a list element,
// but using iterators instead of pointers makes the code of
// Buchberger's algorithm much easier to read.

// The delegation of tasks to this classes and the realization of some
// functions are quite special.
// The insertion of list elements is the task of class list, while the
// deletion is delegated to the iterator class. The reason is simple:
// Elements are almost always inserted at the beginning of a list
// (except from the case of an ordered list), but can be deleted from
// an arbitrary place (this is necessary e.g. if a binomial is reduced
// to zero during Buchberger's algorithm).

// For efficiency reasons, the iterator operations are not
// secure. They do not check if the iterator really references a list
// element or if it has reached the end of the list. To assert this,
// use the is_at_end()-test.

// I have designed an own iterator class instead of taking pointers
// for iteration as list members for flexibility reasons:
// Buchberger's algorithm requires to form pairs of binomials (or, to
// test S-pair-criteria, even triples). So we need up to three
// pointers to iterate over the generator lists. The advantage of the
// iterator class is simply that the programmer does not have to keep
// in mind the lists over which he is actually iterating.
// This was very helpful for me during the implementation of
```

```
// Buchberger's algorithm (especially for the implementation of
// SUPPORT_DRIVEN_METHODS_EXTENDED where many lists have to be
// considered).

// There are two different implementations of the class list:
// - If SL_LIST is set, we use a simply linked list.
//   The insert operations are very efficient because we have to set
//   very few pointers. But the deletion of an element is a dangerous
//   operation because it moves the following element (not the
//   binomial, but the struct) to another storage place.
// - If DL_LIST is set, we use a doubly linked list.
//   This shows to be considerably slower, but is easier to use.

#ifndef LIST_H
#define LIST_H

#define SL_LIST
// possibilities:
// SL_LIST
// DL_LIST

#include "binomial__term_ordering.h"

///////////////////////////// struct element //////////////////////

typedef struct Element
{
  binomial *entry;
  struct Element *next;

#ifdef DL_LIST

  struct Element *previous;

#endif  // DL_LIST

  // flags needed by the ideal implementation to speed up Buchberger's
  // algorithm
  BOOLEAN done;
  BOOLEAN head_reduced;

} element;

////////////////////////// class list ////////////////////////////

class list
{
```

```
private:

  element *start;
  // pointer to the beginning

public:

// constructors and destructor

  list();
  // Creates an "empty" list (with a dummy element for a user-friendly
  // iterator class, see the comments in list.cc).

  list(const list&);
  // copy-constructor

  ~list();
  // destructor

// inserting

  list& insert(binomial&);
  list& copy_insert(const binomial&);
  // These operations insert a binomial at the beginning of the list:
  // insert does this by placing pointers on it, copy_insert by
  // copying the binomial.
  // The first operation is dangerous; but the consequent use of the
  // combination insert - extract_element (cf. class list_iterator)
  // instead of copy_insert - delete_element is very important for the
  // performance of Buchberger's algorithm.

  list& _insert(binomial&);
  list& _copy_insert(const binomial&);
  // A more efficient insert function for lists that do not use
  // the 2 flags done and head_reduced - they are not set.

  list& ordered_insert(binomial&, const term_ordering&);
  list& ordered_copy_insert(const binomial&, const term_ordering&);
  // These operations insert a binomial into a list
  // according to the given term ordering.
  // The list should be ordered by the same term ordering.
  // To insert elements, we use a simple linear search.

  list& _ordered_insert(binomial&, const term_ordering&);
  list& _ordered_copy_insert(const binomial&, const term_ordering&);
  // A little more efficient ordered_insert function for lists that do
  // not use the 2 flags done and head_reduce - they are not set.
```

```
// output

  void print() const;
  void ordered_print(const term_ordering&) const;
  // Writes the list to the standard output medium.
  // The first routine writes the list elements as they are ordered in
  // the list.
  // The second one writes them in increasing order with respect to
  // the argument term ordering; the list has not to be ordered before
  // and will not be ordered after the function call. This routine is
  // designed for an easy comparison of Groebner bases.

  void print(FILE* output) const;
  void ordered_print(FILE* output, const term_ordering&) const;
  // Writes the list into the referenced file which has to be opened
  // for writing before.

  void print(ofstream&) const;
  void ordered_print(ofstream&, const term_ordering&) const;
  // Writes the list into the given ofstream.

  void format_print(ofstream&) const;
  void ordered_format_print(ofstream&, const term_ordering&) const;
  // Writes the list into the given ofstream in the format needed by
  // the IP-algorithms.

  friend class list_iterator;
};

//////////////////// class list_iterator /////////////////////////

class list_iterator
{
private:

  element* actual;

public:

// constructors and destructor

  list_iterator();
  // Sets actual to NULL.

  list_iterator(list& l);
  // Sets a list_iterator to the beginning of l.
```

```
  list_iterator(list_iterator& iter);
  // Sets a list iterator to the same position as iter.

  ~list_iterator();
  // destructor

// object information

  BOOLEAN element_is_marked_done() const;
  // Returns the "done"-component of the referenced element.

  BOOLEAN element_is_marked_head_reduced() const;
  // Returns the "head_reduced"-component of the referenced element.

  BOOLEAN is_at_end() const;
  // Returns TRUE iff the iterator is at the end of its list
  // (especially if the iterator references no list).

// assignment

  list_iterator& set_to_list(const list& l);
  // Sets the iterator to the beginning of l.

  list_iterator& operator=(const list_iterator& iter);
  // Sets the iterator to the same element as iter.

  list_iterator& next();
  // Sets the iterator to the next entry.

// comparison

  int operator==(const list_iterator& iter) const;
  int operator!=(const list_iterator& iter) const;
  // To verify if actual references the same element as iter.actual.

  int next_is(const list_iterator& iter) const;
  // Checks if actual->next references the same element as
  // iter.actual.
  // This check is needed in very special situations when using simply
  // linked lists (two iterators reference subsequent elements of a
  // list, and the first of them is deleted; then the iterator
  // referencing the second before deletion references freed memory
  // after deletion, cf. the implementation of delete and extract).

// manipulation of list elements
```

```
  // All this operations could be declared as const because they do
  // not change the "actual" pointer. For suggestive reasons, this is
  // not done because they change the referenced list element.

  binomial& get_element();
  // Returns the referenced binomial.

  list_iterator& delete_element();
  // Deletes the referenced binomial.

  list_iterator& extract_element();
  // Only resets pointers so that the referenced binomial is no longer
  // found when iterating over the list.

  list_iterator& mark_element_done();
  // Sets the "done"-component of the referenced element to TRUE.

  list_iterator& mark_element_undone();
  // Sets the "done"-component of the referenced element to FALSE.

  list_iterator& mark_element_head_reduced();
  // Sets the "head_reduced"-component of the referenced element to
  // TRUE.

  list_iterator& mark_element_head_unreduced();
  // Sets the "head_reduced"-component of the referenced element to
  // FALSE.
};
#endif  // list.h
```

A.1.8 LLL.h

```
// LLL.h

// This file contains the declaration of two routines implementing
// special variants of the LLL-algorithm.
// For further explanations see the book of Henri Cohen, A Course in
// Computational Algebraic Number Theory, Springer 1993.

// When performing the LLL-algorithm, some coefficients grow very fast.
// Therefore one should use a  data type for arbitrary long integers
// (called "BigInt" in the code).
// If no such data type is specified, BigInt is defined to be a long
// (cf. globals.h).

#ifndef LLL_H
#define LLL_H
```

```
#include "globals.h"

extern short relations(BigInt** b, const short& number_of_vectors,
                       const short& vector_dimension, BigInt**& H);
// Computes the relations of the input vectors stored in b and returns the
// dimension r of the lattice spanned by these relations.
// The return value -1 indicates that an error has occurred.
// A LLL-reduced basis of the relations is written into the two-dimensional
// array H. Memory allocation for this array is done in the routine;
// when leaving the routine, the dimension will be vector_columns x r.
// This routine corresponds to algorithm 2.7.2 in Cohen's book.

extern short integral_LLL(BigInt **b, const short& number_of_vectors,
                          const short& vector_dimension);
// Reduces the input vectors stored in b in the sense of an LLL-reduction.
// The input vectors have to be linearly independent.
// ATTENTION: The input vectors are modified during this algorithm. For
// efficiency reasons in our application, we do NOT store a transformation
// matrix!
// The return value is -1 if an error has occurred, 0 else.
// This routine corresponds to algorithm 2.6.7 in Cohen's book.

#endif  // LLL_H
```

A.1.9 matrix.h

```
// matrix.h

// This class is designed to handle the input of the integer
// programming problem. It offers facilities to convert a
// two-dimensional integer array into a matrix and to read a matrix
// from an input file.
// Furthermore, some special routines needed by the IP-algorithms are
// implemented:
// - the computation of a lattice basis for the integer kernel via the
//   LLL algorithm;
// - the computation of a kernel vector which has no zero components
//   (if such a vector exists) and which is a basis vector for the
//   kernel lattice;
// - the procedure of Hosten and Shapiro computing a small set of
//   saturation variables for the toric ideal given by the matrix
//   kernel.

#ifndef MATRIX_H
#define MATRIX_H
```

```
#include "LLL.h"

class matrix
{
private:

  short rows;

  short columns;
  // Also used as error flag (values <0):
  // -1 indicates a "semantic" error (which occurs e.g. if some
  //    constructor argument is out of range).
  // -2 indicates an error occured when reading from a file.

  Integer **coefficients;
  // the matrix entries

  BigInt **H;
  // This array is used to store the LLL-reduced lattice basis of the
  // kernel. Memory allocation is done in the LLL-routines, so the
  // array is only allocated if such a basis is really computed.

  short _kernel_dimension;
  // the number of vectors stored in H (the size of these vectors is
  // columns)
  // If _kernel_dimension==-2, no kernel basis has been computed yet.
  // If _kernel_dimension==-1, an error has occured during the kernel
  // basis computation.

public:

// constructors and destructor

  matrix(const short& row_number, const short& column_number);
  // Creates a zero matrix of the specified size.

  matrix(const short& row_number, const short& column_number,
         Integer** entries);
  // Builds a matrix from its coefficient array.

  matrix(ifstream& input);
  // Reads a matrix from the given ifstream.
  // The input stream must have the following format:
  //
  //    m=number of rows
  //    n=number of columns
  //    coefficients 0..n-1 of row 1
```

```
  //    coefficients 0..n-1 of row 2
  //             ...
  //    coefficients 0..n-1 of row m

  matrix(const short& m, const short& n, ifstream& input);
  // Reads a (m x n)-matrix from the given ifstream.
  // The input stream must have the following format:
  //
  //    coefficients 0..n-1 of row 1;
  //    coefficients 0..n-1 of row 2;
  //             ...
  //    coefficients 0..n-1 of row m.

  matrix(const matrix&);
  // copy-constructor (also copies H)

  ~matrix();
  // destructor

// object properties

  BOOLEAN is_nonnegative() const;
  // Returns TRUE if all entries of the matrix are >=0, else FALSE.

  short error_status() const;
  // Returns columns iff columns<0 (this is the "error flag"), else 0.

  short row_number() const;
  // Retuns the row number.

  short column_number() const;
  // Returns the column number.

  short kernel_dimension() const;
  // Returns the kernel dimension.

// special routines for the IP-algorithms

  short LLL_kernel_basis();
  // Computes a LLL-reduced lattice basis of the matrix kernel and
  // returns the kernel dimension (-1 if an error has occurred).
  // This dimension is also stored in the member kernel_dimension.

  short compute_nonzero_kernel_vector();
  // Transforms the kernel lattice basis stored in H into a basis that
  // contains a vector whose components are all !=0;
  // returns 0 if no such vector exists, else 1.
```

```
  // If no lattice basis has been computed before, this is done now.

  short compute_flip_variables(short*&);
  // Computes a set of flip variables for the algorithm of Di Biase
  // and Urbanke from a kernel vector with no zero components.
  // Returns the size of that set if such a kernel vector exists,
  // else -1.
  // If necessary, such a vector is computed.
  // The computed set is copied to the argument pointer (memory
  // allocation is done in the routine) to be accessible for the
  // calling function.

  short hosten_shapiro(short*& sat_var);
  // Computes a set of saturation variables for the ideal defined by
  // the kernel lattice basis and returns the size of that set.
  // If no lattice basis has been computed before, this is done now.
  // The computed set is stored in the argument pointer (memory
  // allocation is done in the routine) to be accessible for the
  // calling function.
  // This routine implements the most simple strategy for computing
  // this set: The kernel vectors are examined in their actual
  // order. A "greedy" strategy choosing "clever" kernel vectors to
  // begin with could give better results in many cases, but this is
  // not guaranteed... (And what is a clever choice?)

// output

  void print() const;
  // Writes the matrix to the standard output medium.

  void print(FILE*) const;
  // Writes the matrix to the referenced file which has to be opened
  // for writing before.

  void print(ofstream&) const;
  // Writes the matrix to the given ofstream.

  friend class ideal;
  // Our toric ideals are all constructed from matrices, and the
  // matrix class is designed only for the needs of the ideal
  // constructors. So it would be unnecessary overhead to hide the
  // matrix members from these constructors.
};
#endif
```

A.1.10 term_ordering.h

```
// term_ordering.h

// Our IP-algorithms need two kinds of term orderings: weighted term
// orderings and block orderings with an elimination block and a
// weighted block. They are constructed from the input instance and
// for the specific requirements of the algorithms. To allow an easy
// and flexible handling, the term ordering is realized as a class.

// Weight vectors for weighted term orderings have to be nonnegative
// in most contexts, but input vectors with negative components are
// not refused by the constructors. This has the following two
// reasons:
// - There is a more general notion of a term ordering which does not
//   claim the monomial 1 to be a (unique) minimal monomial. This
//   notion allows negative weights.
// - This class is written for the final purpose of integer
//   programming. The linear objective function induces the term
//   ordering and might also have negative components. Although the
//   program actually cannot treat this case, there are ways to tackle
//   it. In view of a possible later expansion of the program, the
//   nonnegativity check is delegated to a separate function instead
//   of doing it in the term ordering constructors.

// For the same reasons as mentionned in the binomial class,
// frequently used operations on Integer vectors do not perform range
// checks, nor do they check the "error flag", see below.
// These are concretely the functions weight(...),
// compare_to_zero(...) and compare(...).
// All other routines perform argument checks (range, file
// format...). This is only really important when reading from a file,
// as most other range checks are already made by the routines calling
// these functions. But as these operations are executed only a few
// times in the course of our algorithms, the checks do not slow down
// computation.

#ifndef TERM_ORDERING_H
#define TERM_ORDERING_H

#include "globals.h"

class binomial;

class term_ordering
{
protected:
```

```
  float *weight_vector;

  short weighted_block_size;
  // the number of weighted variables
  // Also the "error flag", i.e. values <0 indicate an error:
  // -1 indicates a "semantic" error (arguments out of range, for
  //    example a negative number of variables);
  // -2 indicates an error occurred when reading from a file.

  short weighted_ordering;
  // the term ordering refining the weight on the weighted variables
  // possibilities: see in globals.h

  short elimination_block_size;
  // the number of elimination variables

  short elimination_ordering;
  // the term ordering on the elimination variables

  BOOLEAN homogeneous;
  // This flag is set to TRUE if the term ordering is thought of as a
  // homogeneous one (i.e. the weighted part), else to FALSE. The
  // standard setting is FALSE.
  // The routine compare_to_zero is faster if the term ordering is
  // homogenous. It would be safer and more intuitive to take this
  // flag as a member of the ideal class; this would, however, require
  // many supplementary functions (homogeneous versions of existing
  // functions) in the ideal and even in the binomial class.

public:

// constructors and destructor

  term_ordering(const BOOLEAN& homogeneous=FALSE);
  // Sets weighted_block_size and elimination_block_size to zero.
  // With this default contructor, the term ordering can be taken as a
  // member of another class (the ideal class in our case).

  term_ordering(const short& number_of_weighted_variables,
                const float* weights,
                const short& _weighted_ordering,
                const BOOLEAN& _homogeneous=FALSE);
  // Initializes weight vector with weights,
  // weighted_block_size with number_of_weighted_variables,
  // the refining ordering as given in the argument
  // and elimination_block_size with 0.
```

```
  term_ordering(const short& number_of_weighted_variables,
                const float* weights,
                const short& _weighted_ordering,
                const short& number_of_elimination_variables,
                const short& _elimination_ordering,
                const BOOLEAN& _homogeneous=FALSE);
  // Initializes weight vector with weights,
  // weighted_block_size with number_of_weighted_variables,
  // the orderings as given in the arguments
  // and elimination_block_size with
  // _number_of_elimination_variables.

  term_ordering(ifstream& input, const short& _weighted_ordering,
                const BOOLEAN& _homogeneous=FALSE);
  // Reads a weight vector from the input stream and converts it into
  // a weighted term ordering (with the refining ordering given by the
  // second argument).
  // The input stream must have the following format:
  //
  //    size n of the weight vector (=weighted_block_size)
  //    components 0..n-1 of the weight vector

  term_ordering(const short& n, ifstream& input,
                const short& _weighted_ordering,
                const BOOLEAN& _homogeneous=FALSE);
  // Reads a weight vector of size n from the input stream and
  // converts it into a weighted term ordering (with the refining
  // ordering given by the second argument).
  // The input stream must have the following format:
  //
  //    components 0..n-1 of the weight vector

  term_ordering(const term_ordering&);
  // copy-constructor

  ~term_ordering();
  // destructor

// object properties

  short number_of_weighted_variables() const;
  // Returns weighted_block_size.

  short weight_refinement() const;
  // Returns weighted_ordering.
```

```
  short number_of_elimination_variables() const;
  // Returns elimination_block_size.

  short elimination_refinement() const;
  // Returns elimination_ordering.

  BOOLEAN is_homogeneous() const;
  // Returns homogeneous.

  short error_status() const;
  // Returns weighted_block_size if weighted_block_size<0, else 0
  // (this is the "error flag").

  BOOLEAN is_nonnegative() const;
  // Returns TRUE iff all components of the weight vector are >=0.
  // If this is not the case, Buchberger's algorithm will perhaps not
  // terminate using the actual ordering, i.e. the ordering is no
  // well ordering.

  BOOLEAN is_positive() const;
  // Returns TRUE ifs all components of the weight vector are >0.
  // If this is not the case, this may lead to problems even if all
  // weights are nonnegative: Zero weights do not give a well ordering
  // in combination with a reverse lexicographical refinement.

// frequently used comparison functions

  double weight(const Integer* v) const;
  // Returns the weight of an Integer vector v with respect to
  // weight_vector, i.e. the scalar product of the two vectors.
  // UNCHECKED PRECONDITION: v has weighted_block_size components.
  // The return type is double to avoid overflows.

  short compare_to_zero(const Integer* v) const;
  // This function is used to determine a binomials's head with
  // respect to the actual term ordering. The comparison of its two
  // monomials is done according to the binomial data structure:
  // Instead of comparing explicitely two monomials, the Integer
  // vector corresponding to their difference is compared to the zero
  // vector.
  // The function returns
  //          1, if v>0,
  //      -1, if v<0,
  //       0, if v=0
  // UNCHECKED PRECONDITION:
  // v has weighted_block_size + elimination_block_size components
```

```
  short compare(const binomial& bin1, const binomial& bin2) const;
  // Compares the given binomials by first comparing the heads, then
  // in the case of equality the tails.
  // UNCHECKED PRECONDITION: The exponent vectors of the binomials
  // have already the correct sign, i.e. the positive part of a vector
  // is really its head and not its tail.
  // The function returns
  //        -1 if bin1<bin2
  //         0 if bin1=bin2
  //         1 if bin1>bin2

// assignment and access operators

  term_ordering& operator=(const term_ordering&);
  // assignment operator with memory control

  float operator[](const short& i) const;
  // Returns weight_vector[i] (FLT_MAX if i is out of range).

// routines needed by the algorithms to manipulate the term ordering

  term_ordering& convert_to_weighted_ordering();
  // Sets elimination_block_size to 0.

  term_ordering& convert_to_elimination_ordering
  (const short& number_of_elimination_variables,
   const short& _elimination_ordering);
  // Sets elimination_block_size to number_of_elimination_variables
  // and the ordering on the elimination variables as given in the
  // argument.

  term_ordering& append_weighted_variable(const float& weight);
  // Appends the given weight to the weight vector (with memory
  // control).

  term_ordering& delete_last_weighted_variable();
  // Deletes the last weighted variable (with memory control).

  term_ordering& swap_weights(const short& i, const short& j);
  // This function is used if two weighted variables must change their
  // position (order) with respect to the refining ordering.

// output

  void print_weight_vector() const;
  void print() const;
  // Writes the term ordering to the standard output medium
```

```
  // (only the weight vector or with supplementary information).

  void print_weight_vector(FILE*) const;
  void print(FILE*) const;
  // Writes the term ordering to the referenced file
  // which has to be opened for writing before.

  void print_weight_vector(ofstream&) const;
  void print(ofstream&) const;
  // Writes the term ordering to the given ofstream.

  void format_print_weight_vector(ofstream&) const;
  // Writes the term ordering to the given ofstream in the format
  // required by the IP-algorithms.
};
#endif  // TERM_ORDERING_H
```

A.2 Implementierung

Es folgen die `.cc`-Dateien, welche die Klassen und Funktionen aus den Header-Dateien des vorigen Abschnitts implementieren. Wir zitieren nur interessante Teile des Quellcodes; für den kompletten Code (der zu umfangreich wäre) bitte e-mail an `ctheis@math.uni-sb.de`. Fehlende Stellen sind mit

`...` // + Kommentar

gekennzeichnet.

A.2.1 binomial.cc

```
// binomial.cc

// implementation of class binomial

#ifndef BINOMIAL_CC
#define BINOMIAL_CC

#include "binomial__term_ordering.h"

//////////////////// constructors and destructor /////////////////

// For a better overview, the constructor code is separated for
// NO_SUPPORT_DRIVEN_METHODS and SUPPORT_DRIVEN_METHODS.

#ifdef NO_SUPPORT_DRIVEN_METHODS

... // as with SUPPORT_DRIVEN_METHODS without setting head_support and
    // tail_support
```

```
#endif  // NO_SUPPORT_DRIVEN_METHODS

#ifdef SUPPORT_DRIVEN_METHODS

binomial::binomial(const short& number_of_variables,
                   const Integer* exponents,
                   const term_ordering& w)
    :_number_of_variables(number_of_variables),
     head_support(0), tail_support(0)
{
  // range check for rarely used constructors
  if(_number_of_variables<=0)
  {
    cerr<<"\nWARNING: binomial::binomial(const short&, "
      "const Integer*):\n argument out of range"<<endl;
    exponent_vector=NULL;
    // to avoid problems when deleting
    return;
  }

  exponent_vector=new Integer[_number_of_variables];
  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);
  // number of bits of a long int
  // determine head and tail
  short sign;
  if(w.compare_to_zero(exponents)>=0)
    sign=1;
  else
    sign=-1;

  for(short i=0;i<_number_of_variables;i++)
  {

#ifdef SUPPORT_VARIABLES_FIRST

    Integer actual_entry=sign*exponents[i];
    exponent_vector[i]=actual_entry;

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST

    short j=_number_of_variables-1-i;
    Integer actual_entry=sign*exponents[j];
    exponent_vector[j]=actual_entry;
```

```
#endif  // SUPPORT_VARIABLES_LAST

    if(i<size_of_support_vectors)
      // variable i is considered in the support vectors
      if(actual_entry>0)
        head_support|=(1<<i);
        // bit i of head_support is set to 1 (counting from 0)
      else
        if(actual_entry<0)
          tail_support|=(1<<i);
    // bit i of tail_support is set to 1
  }
}

... // other easier constructor

#endif  // SUPPORT_DRIVEN_METHODS

... // binomial::~binomial()

/////////////////// object information //////////////////////////

... // short binomial::number_of_variables() const
... // short binomial::error_status() const

/////////////// assignment and access operators //////////////////

binomial& binomial::operator=(const binomial& b)
{
  if(&b==this)
    return *this;

#ifdef SUPPORT_DRIVEN_METHODS

  head_support=b.head_support;
  tail_support=b.tail_support;

#endif  // SUPPORT_DRIVEN_METHODS

  if(_number_of_variables!=b._number_of_variables)
  {
    delete[] exponent_vector;
    _number_of_variables=b._number_of_variables;

    if(_number_of_variables<=0)
    {
      cerr<<"\nWARNING: binomial& binomial::operator=(const "
```

```
      "binomial&):\n assignment from corrupt binomial"<<endl;
      exponent_vector=NULL;
      return (*this);
    }
    exponent_vector=new Integer[_number_of_variables];
  }
  for(short i=0;i<_number_of_variables;i++)
    exponent_vector[i]=b.exponent_vector[i];
  return(*this);
}

Integer binomial::operator[](const short& i) const
{
  return exponent_vector[i];
}

///////////////// comparison operators ///////////////////////////////

BOOLEAN binomial::operator==(const binomial& b) const
{
  if(this == &b)
    return(TRUE);

#ifdef SUPPORT_DRIVEN_METHODS

  if(head_support!=b.head_support)
    return(FALSE);
  if(tail_support!=b.tail_support)
    return(FALSE);

#endif  // SUPPORT_DRIVEN_METHODS

  for(short i=0;i<_number_of_variables;i++)
    if(exponent_vector[i]!=b.exponent_vector[i])
      return(FALSE);
  return(TRUE);
}

... // BOOLEAN binomial::operator!=(const binomial& b) const

// operators for efficient comparisons with the zero binomial
// (comp_value=0)

BOOLEAN binomial::operator==(const Integer comp_value) const
{

#ifdef SUPPORT_DRIVEN_METHODS
```

```
  if(comp_value==0)
  {
    if(head_support!=0)
      return(FALSE);
    if(tail_support!=0)
      return(FALSE);
  }

#endif  // SUPPORT_DRIVEN_METHODS

  for(short i=0;i<_number_of_variables;i++)
    if(exponent_vector[i]!=comp_value)
      return(FALSE);
  return(TRUE);
}

... // BOOLEAN binomial::operator!=(const Integer comp_value) const

BOOLEAN binomial::operator<=(const Integer comp_value) const
{

#ifdef SUPPORT_DRIVEN_METHODS

  if(comp_value==0)
    if(head_support!=0)
      return(FALSE);

#endif  // SUPPORT_DRIVEN_METHODS

  for(short i=0;i<_number_of_variables;i++)
    if(exponent_vector[i]>comp_value)
      return(FALSE);
  return(TRUE);
}

... // BOOLEAN binomial::operator>=(const Integer comp_value) const

/////////// basic routines for Buchbergers's algorithm //////////////

Integer binomial::head_reductions_by(const binomial& b) const
// Returns the number of possible reductions of the actual binomial's
// head by the binomial b. This is the minimum of the quotients
// exponent_vector[i]/b.exponent_vector[i]
// where exponent_vector[i]>0 and b.exponent_vector[i]>0
// (0 if there are no such quotients).
// A negative return value means b=0 or head(b)=1.
```

```
{

#ifdef NO_SUPPORT_DRIVEN_METHODS

  Integer result=-1;
  Integer new_result=-1;
  // -1 stands for infinitely many reductions

  for(short i=0;i<_number_of_variables;i++)
    // explicit sign tests for all components
  {
    Integer actual_b_component=b.exponent_vector[i];

    if(actual_b_component>0)
      // else variable i is not involved in the head of b
    {
      Integer actual_component=exponent_vector[i];
      if(actual_component<actual_b_component)
        return 0;
      new_result=(Integer) (actual_component/actual_b_component);
      // new_result>=1
      if((new_result<result) || (result==-1))
        // new (or first) minimum
        result=new_result;
    }
 }

#endif  // NO_SUPPORT_DRIVEN_METHODS

#ifdef SUPPORT_DRIVEN_METHODS

  if((head_support&b.head_support)!=b.head_support)
    // head support of b not contained in head support, no reduction
    // possible
    return 0;

  Integer result=-1;
  Integer new_result=-1;
  // -1 stands for infinitely many reductions
  short size_of_support_vectors=CHAR_BIT*sizeof(long);
  // number of bits of a long int
  if(size_of_support_vectors>_number_of_variables)
    size_of_support_vectors=_number_of_variables;
    // number of components of the support vectors

#ifdef SUPPORT_VARIABLES_FIRST
```

```
  for(short i=0;i<size_of_support_vectors;i++)
    // test support variables
    if(b.head_support&(1<<i))
      // bit i of b.head_support is 1
    {
      new_result=(Integer) (exponent_vector[i]/b.exponent_vector[i]);
      // remember that exponent_vector[i]>0 !
      // (head support contains that of b)
      if(new_result==0)
        // exponent_vector[i]<b.exponent_vector[i]
        return 0;
      // new_result>=1
      if((new_result<result) || (result==-1))
        // new (or first) minimum
        result=new_result;
    }

  for(short i=size_of_support_vectors;i<_number_of_variables;i++)
    // test non-support variables
    // from now on we need explicit sign tests
  {
    Integer actual_b_component=b.exponent_vector[i];

    if(actual_b_component>0)
      // else variable i is not involved in the head of b
    {
      Integer actual_component=exponent_vector[i];
      if(actual_component<actual_b_component)
        return 0;
      new_result=(Integer) (actual_component/actual_b_component);
      // new_result>=1
      if((new_result<result) || (result==-1))
        // new (or first) minimum
        result=new_result;
    }
  }

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST

  for(short i=0;i<size_of_support_vectors;i++)
    // test support variables
    if(b.head_support&(1<<i))
      // bit i of b.head_support is 1
    {
      short j=_number_of_variables-1-i;
```

```
      new_result=(Integer) (exponent_vector[j]/ b.exponent_vector[j]);
      // remember that exponent_vector[_number_of_variables-1-i]>0 !
      // (head support contains that of b)
      if(new_result==0)
        // exponent_vector[_number_of_variables-1-i]
        // <b.exponent_vector[_number_of_variables-1-i]
        return 0;
      // new_result>=1
      if((new_result<result) || (result==-1))
        // new (or first) minimum
        result=new_result;
    }

  for(short i=size_of_support_vectors;i<_number_of_variables;i++)
    // test non-support variables
    // from now on we need explicit sign tests
  {
    short j=_number_of_variables-1-i;
    Integer actual_b_component=b.exponent_vector[j];

    if(actual_b_component>0)
      // else variable number_of_variables-1-i is not involved in the
      // head of b
    {
      Integer actual_component=exponent_vector[j];
      if(actual_component<actual_b_component)
        return 0;
      new_result=(Integer) (actual_component/actual_b_component);
      // new_result>=1
      if((new_result<result) || (result==-1))
        // new (or first) minimum
        result=new_result;
    }
  }

#endif  // SUPPORT_VARIABLES_LAST

#endif  // SUPPORT_DRIVEN_METHODS

  return(result);
}

... // Integer binomial::tail_reductions_by(const binomial& b) const

int binomial::reduce_head_by(const binomial& b,
                             const term_ordering& w)
{
```

```
  Integer reduction_number=head_reductions_by(b);
  if(reduction_number<=0)
    return 0;

  for(short i=0;i<_number_of_variables;i++)
    exponent_vector[i]-=(reduction_number * b.exponent_vector[i]);
  // multiple reduction
  // reduction corresponds to subtraction of vectors
  short sign=w.compare_to_zero(exponent_vector);

#ifdef NO_SUPPORT_DRIVEN_METHODS

  if(sign==0)
    // binomial reduced to zero
    return 2;
  for(short i=0;i<_number_of_variables;i++)
    exponent_vector[i]*=sign;

#endif  // NO_SUPPORT_DRIVEN_METHODS

#ifdef SUPPORT_DRIVEN_METHODS

  head_support=0;
  tail_support=0;
  if(sign==0)
    // binomial reduced to zero
    return 2;
  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);

  // recompute the support vectors

#ifdef SUPPORT_VARIABLES_FIRST

  for(short i=0;i<_number_of_variables;i++)
  {
    Integer& actual_entry=exponent_vector[i];
    // to avoid unnecessary pointer arithmetic
    actual_entry*=sign;
    if(i<size_of_support_vectors)
      if(actual_entry>0)
        head_support|=(1<<i);
      else
        if(actual_entry<0)
          tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_FIRST
```

```
#ifdef SUPPORT_VARIABLES_LAST

  for(short i=0;i<_number_of_variables;i++)
  {
    Integer& actual_entry=exponent_vector[_number_of_variables-1-i];
    // to avoid unneccessary pointer arithmetic
    actual_entry*=sign;
    if(i<size_of_support_vectors)
      if(actual_entry>0)
        head_support|=(1<<i);
      else
        if(actual_entry<0)
          tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_LAST

#endif  // SUPPORT_DRIVEN_METHODS

  return 1;
}

int binomial::reduce_tail_by(const binomial& b,
                             const term_ordering& w)
{
  Integer reduction_number=tail_reductions_by(b);
  if(reduction_number<=0)
    return 0;
  for(short i=0;i<_number_of_variables;i++)
    exponent_vector[i]+=(reduction_number * b.exponent_vector[i]);
  // multiple reduction
  // reduction corresponds to addition of vectors
  // a tail reduction does not require a sign check

#ifdef SUPPORT_DRIVEN_METHODS

  head_support=0;
  tail_support=0;
  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);

  // recompute the support vectors

#ifdef SUPPORT_VARIABLES_FIRST

  for(short i=0;i<_number_of_variables;i++)
  {
```

```
    Integer& actual_entry=exponent_vector[i];
    // to avoid unnecessary pointer arithmetic
    if(i<size_of_support_vectors)
      if(actual_entry>0)
        head_support|=(1<<i);
      else
        if(actual_entry<0)
          tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST

  for(short i=0;i<_number_of_variables;i++)
  {
    Integer& actual_entry=exponent_vector[_number_of_variables-1-i];
    // to avoid unneccessary pointer arithmetic
    if(i<size_of_support_vectors)
      if(actual_entry>0)
        head_support|=(1<<i);
      else
        if(actual_entry<0)
          tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_LAST

#endif  // SUPPORT_DRIVEN_METHODS

  return 1;
}

binomial& S_binomial(const binomial& a, const binomial& b,
                     const term_ordering& w)
{
  binomial* S_bin=new binomial(a._number_of_variables);
  binomial& result=*S_bin;
  // Note that we allocate memory for the result binomial. We often
  // use pointers or references as argument and return types because
  // the generating binomials of an ideal are kept in lists. For the
  // performance of Buchberger's algorithm it it very important to
  // avoid local copies of binomials, so a lot of attention is paid on
  // the choice of argument and return types. As this choice is done
  // in order to improve performance, it might be a bad choice with
  // respect to code reuse (there are some dangerous constructions).
```

```
  for(short i=0;i<result._number_of_variables;i++)
    result.exponent_vector[i]=a.exponent_vector[i]-b.exponent_vector[i];
  // The S-binomial corresponds to the vector difference.

  // compute head and tail
  short sign=w.compare_to_zero(result.exponent_vector);

#ifdef NO_SUPPORT_DRIVEN_METHODS

  if(sign==0)
    // binomial reduced to zero
    return result;
  for(short i=0;i<result._number_of_variables;i++)
    result.exponent_vector[i]*=sign;

#endif  // NO_SUPPORT_DRIVEN_METHODS

#ifdef SUPPORT_DRIVEN_METHODS

  result.head_support=0;
  result.tail_support=0;
  if(sign==0)
    // binomial reduced to zero
    return result;
  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);

  // recompute the support vectors

#ifdef SUPPORT_VARIABLES_FIRST

  for(short i=0;i<result._number_of_variables;i++)
  {
    Integer& actual_entry=result.exponent_vector[i];
    // to avoid unnecessary pointer arithmetic
    actual_entry*=sign;
    if(i<size_of_support_vectors)
      if(actual_entry>0)
        result.head_support|=(1<<i);
      else
        if(actual_entry<0)
          result.tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST
```

```
  for(short i=0;i<result._number_of_variables;i++)
  {
    Integer& actual_entry=result.exponent_vector
      [result._number_of_variables-1-i];
    // to avoid unneccessary pointer arithmetic
    actual_entry*=sign;
    if(i<size_of_support_vectors)
      if(actual_entry>0)
        result.head_support|=(1<<i);
      else
        if(actual_entry<0)
          result.tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_LAST

#endif  // SUPPORT_DRIVEN_METHODS

  return result;
}

///////////// criteria for unnecessary S-pairs //////////////////////

// The criteria are programmed in a way that tries to minimize pointer
// arithmetic. Therefore the code may appear a little bit inflated.

BOOLEAN relatively_prime(const binomial& a, const binomial& b)
{

#ifdef NO_SUPPORT_DRIVEN_METHODS

  // look at all variables
  for(short i=0;i<a._number_of_variables;i++)
    if((a.exponent_vector[i]>0) && (b.exponent_vector[i]>0))
      return FALSE;
  return TRUE;

#endif  // NO_SUPPORT_DRIVEN_METHODS

#ifdef SUPPORT_DRIVEN_METHODS

  if((a.head_support & b.head_support)!=0)
  // common support variable in the heads
    return FALSE;
  // no common support variable in the heads, look at remaining
  // variables
  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);
```

```
#ifdef SUPPORT_VARIABLES_FIRST

  for(short i=size_of_support_vectors;i<a._number_of_variables;i++)
    if((a.exponent_vector[i]>0) && (b.exponent_vector[i]>0))
      return FALSE;
  return TRUE;

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST

  for(short i=a._number_of_variables-1-size_of_support_vectors;
      i>=0;i--)
    if((a.exponent_vector[i]>0) && (b.exponent_vector[i]>0))
      return FALSE;
  return TRUE;

#endif  // SUPPORT_VARIABLES_LAST

#endif  // SUPPORT_DRIVEN_METHODS
}

BOOLEAN M(const binomial& a, const binomial& b, const binomial& c)
// Returns TRUE iff lcm(head(a),head(c)) properly divides
// lcm(head(b),head(c)). This is checked by comparing the positive
// components of the exponent vectors.
{

#ifdef SUPPORT_DRIVEN_METHODS

  long b_or_c=b.head_support|c.head_support;

  if((a.head_support|b_or_c) != b_or_c)
    return FALSE;
  // The support of lcm(head(a),head(c)) equals the union of the head
  // supports of a and c. The above condition verifies if the support
  // of lcm(head(a),head(c)) is contained in the support of
  // lcm(head(b),head(c)) by checking if head a involves a variable
  // that is not involved in head(b) or head(c).

#endif  // SUPPORT_DRIVEN_METHODS

  BOOLEAN properly=FALSE;

  for(short i=0;i<a._number_of_variables;i++)
  {
```

```
    Integer a_exponent=a.exponent_vector[i];
    Integer b_exponent=b.exponent_vector[i];
    Integer c_exponent=c.exponent_vector[i];
    Integer m1=MAXIMUM(a_exponent,c_exponent);
    Integer m2=MAXIMUM(b_exponent,c_exponent);

    if(m1>0)
    {
      if(m1>m2)
        return FALSE;
      if(m1<m2)
        properly=TRUE;
    }
  }
  return properly;
}

BOOLEAN F(const binomial& a, const binomial& b, const binomial& c)
// verifies if lcm(head(a),head(c))=lcm(head(b),head(c))
{

#ifdef SUPPORT_DRIVEN_METHODS

  if((a.head_support|c.head_support)
   !=(b.head_support|c.head_support))
    return FALSE;
  // The above condition verifies if the support of
  // lcm(head(a),head(c)) equals the support of lcm(head(b),head(c)).

#endif  // SUPPORT_DRIVEN_METHODS

  for(short i=0;i<a._number_of_variables;i++)
  {
    Integer a_exponent=a.exponent_vector[i];
    Integer b_exponent=b.exponent_vector[i];
    Integer c_exponent=c.exponent_vector[i];
    Integer m1=MAXIMUM(a_exponent,c_exponent);
    Integer m2=MAXIMUM(b_exponent,c_exponent);

    if((m1!=m2) && (m1>0 || m2>0))
      return FALSE;
  }
  return TRUE;
}

BOOLEAN B(const binomial& a, const binomial& b, const binomial& c)
// verifies if head(a) divides lcm(head(b),head(c)) and
```

```
// lcm(head(a),head(b))!=lcm(head(b),head(c))!=lcm(head(a),head(c))
{

#ifdef SUPPORT_DRIVEN_METHODS

  long a_or_b=a.head_support|b.head_support;
  long a_or_c=a.head_support|c.head_support;
  long b_or_c=b.head_support|c.head_support;

  if((a.head_support & b_or_c)!=a.head_support)
    return FALSE;
  // The above condition verifies if the support of head(a) is
  // contained in the support of lcm(head(b),head(c)).

  if( (a_or_c != b_or_c) && (a_or_b != b_or_c))
    // Then the inequality conditions are guaranteed...
  {
    for(short i=0;i<a._number_of_variables;i++)
    {
      Integer b_exponent=b.exponent_vector[i];
      Integer c_exponent=c.exponent_vector[i];
      if(a.exponent_vector[i]>MAXIMUM(b_exponent,c_exponent))
        return FALSE;
    }
    return (TRUE);
  }

  if(a_or_b != b_or_c)
    // Then the first inequality conditions is guaranteed...
    // Verifie only the second.
  {
    BOOLEAN not_equal=FALSE;
    for(short i=0;i<a._number_of_variables;i++)
    {
      Integer a_exponent=a.exponent_vector[i];
      Integer b_exponent=b.exponent_vector[i];
      Integer c_exponent=c.exponent_vector[i];
      Integer m=MAXIMUM(b_exponent, c_exponent);

      if(a_exponent>m)
        return FALSE;
      if(MAXIMUM(a_exponent,c_exponent) != m)
         not_equal=TRUE;
    }
    return(not_equal);
  }
```

```
  if( a_or_c != b_or_c )
    // Then the second inequality conditions is guaranteed...
    // Verifie only the first.
  {
    BOOLEAN not_equal=FALSE;
    for(short i=0;i<a._number_of_variables;i++)
    {
      Integer a_exponent=a.exponent_vector[i];
      Integer b_exponent=b.exponent_vector[i];
      Integer c_exponent=c.exponent_vector[i];
      Integer m=MAXIMUM(b_exponent, c_exponent);

      if(a_exponent > m)
        return FALSE;
      if(MAXIMUM(a_exponent,b_exponent) != m)
        not_equal=TRUE;
    }
    return(not_equal);
  }

#endif  // SUPPORT_DRIVEN_METHODS

  BOOLEAN not_equal_1=FALSE;
  BOOLEAN not_equal_2=FALSE;

  for(short i=0;i<a._number_of_variables;i++)
  {
    Integer a_exponent=a.exponent_vector[i];
    Integer b_exponent=b.exponent_vector[i];
    Integer c_exponent=c.exponent_vector[i];
    Integer m=MAXIMUM(b_exponent, c_exponent);

    if(a_exponent > m)
      return FALSE;
    if(MAXIMUM(a_exponent,b_exponent) != m)
      not_equal_1=TRUE;
    if(MAXIMUM(a_exponent,c_exponent) != m)
      not_equal_2=TRUE;
  }
  return (not_equal_1 && not_equal_2);
}

BOOLEAN second_crit(const binomial& a, const binomial& b,
                    const binomial& c)
// verifies if head(a) divides lcm(head(b),head(c))
{
```

```
#ifdef SUPPORT_DRIVEN_METHODS

  if((a.head_support &
     (b.head_support|c.head_support))!=a.head_support)
    return FALSE;
  // The above condition verifies if the support of head(a) is
  // contained in the support of lcm(head(b),head(c))

#endif  // SUPPORT_DRIVEN_METHODS.

  for(short i=0;i<a._number_of_variables;i++)
  {
    Integer b_exponent=b.exponent_vector[i];
    Integer c_exponent=c.exponent_vector[i];
    if(a.exponent_vector[i]>MAXIMUM(b_exponent,c_exponent))
      return FALSE;
  }
  return (TRUE);
}

//////// special routines needed by the IP-algorithms //////////////

BOOLEAN binomial::involves_elimination_variables(const
        term_ordering& w)
{
// The use of support information would require the distinction of
// various cases here (relation between the number of variables to
// eliminate and the number of support variables) and be quite
// difficult. It is doubtful if this would improve performance.
// As this function is not used in Buchberger's algorithm (and
// therefore rather rarely), I renounce to implement this.

  for(short i=0;i<w.number_of_elimination_variables();i++)
    // elimination variables are always the last ones
    if(exponent_vector[_number_of_variables-1-i]!=0)
      return TRUE;
  return FALSE;
}

binomial& binomial::drop_elimination_variables(const term_ordering& w)
{
  _number_of_variables-=w.number_of_elimination_variables();
  // dangerous (no compatibility check)!!

  // copy components of interest to save memory
  // the leading term has to be recomputed!!
  Integer *aux=exponent_vector;
```

```
  exponent_vector=new Integer[_number_of_variables];
  if(w.weight(aux)>=0)
    for(short i=0;i<_number_of_variables;i++)
      exponent_vector[i]=aux[i];
  else
    for(short i=0;i<_number_of_variables;i++)
      exponent_vector[i]=-aux[i];
  delete[] aux;

#ifdef SUPPORT_DRIVEN_METHODS

  // Recompute head and tail.
  // Normally, this routine is only called for binomials that do not
  // involve the variables to eliminate. But if SUPPORT_VARIABLES_LAST
  // is enabled, the support changes in spite of this. Therefore, the
  // support is recomputed... For the same reasons as mentionned in
  // the preceeding routine, the existing support information is not
  // used.

  head_support=0;
  tail_support=0;
  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);
  if(size_of_support_vectors>_number_of_variables)
    size_of_support_vectors=_number_of_variables;

#ifdef SUPPORT_VARIABLES_FIRST

  for(short i=0;i<size_of_support_vectors;i++)
  {
    Integer actual_entry=exponent_vector[i];
    if(actual_entry>0)
      head_support|=(1<<i);
    else
      if(actual_entry<0)
        tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST

  for(short i=0;i<size_of_support_vectors;i++)
  {
    Integer actual_entry=exponent_vector[_number_of_variables-1-i];
    if(actual_entry>0)
      head_support|=(1<<i);
    else
```

```
      if(actual_entry<0)
        tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_LAST

#endif  // SUPPORT_DRIVEN_METHODS

  return *this;
}

binomial& binomial::drop_last_weighted_variable(const
          term_ordering& w)
{
  _number_of_variables--;
  // dangerous!!

  // copy components of interest to save memory
  // the leading term has to be recomputed!!
  Integer *aux=exponent_vector;
  exponent_vector=new Integer[_number_of_variables];
  short last_weighted_variable=w.number_of_weighted_variables()-1;
  aux[last_weighted_variable]=0;
  // set last component to zero, so it cannot influence the weight

  if(w.weight(aux)>=0)
  {
    for(short i=0;i<last_weighted_variable;i++)
      exponent_vector[i]=aux[i];
    for(short i=last_weighted_variable;i<_number_of_variables;i++)
      exponent_vector[i]=aux[i+1];
  }
  else
  {
    for(short i=0;i<last_weighted_variable;i++)
      exponent_vector[i]=-aux[i];
     for(short i=last_weighted_variable;i<_number_of_variables;i++)
      exponent_vector[i]=-aux[i+1];
  }
  delete[] aux;

#ifdef SUPPORT_DRIVEN_METHODS

  // Recompute head and tail.
  // Normally, this routine is only called for binomials that do not involve
  // the variable to be dropped. But if SUPPORT_VARIABLES_LAST is enabled,
  // the support changes in spite of this. Therefore, the support is
```

```
  // recomputed... For the same reasons as mentionned in the preceeding
  // routines, the existing support information is not used.

  head_support=0;
  tail_support=0;
  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);
  if(size_of_support_vectors>_number_of_variables)
    size_of_support_vectors=_number_of_variables;

#ifdef SUPPORT_VARIABLES_FIRST

  for(short i=0;i<size_of_support_vectors;i++)
  {
    Integer actual_entry=exponent_vector[i];
    if(actual_entry>0)
      head_support|=(1<<i);
    else
      if(actual_entry<0)
        tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST

  for(short i=0;i<size_of_support_vectors;i++)
  {
    Integer actual_entry=exponent_vector[_number_of_variables-1-i];
    if(actual_entry>0)
      head_support|=(1<<i);
    else
      if(actual_entry<0)
        tail_support|=(1<<i);
  }

#endif  // SUPPORT_VARIABLES_LAST

#endif  // SUPPORT_DRIVEN_METHODS

  return *this;
}

int binomial::adapt_to_term_ordering(const term_ordering& w)
{
  if(w.compare_to_zero(exponent_vector)<0)
  {
    // then exchange head and tail
```

```
    for(short i=0;i<_number_of_variables;i++)
      exponent_vector[i]*=(-1);

#ifdef SUPPORT_DRIVEN_METHODS

    unsigned long swap=head_support;
    head_support=tail_support;
    tail_support=swap;

#endif

    return -1;
    // binomial changed
  }
  else
    return 1;
    // binomial unchanged
}

binomial& binomial::swap_variables(const short& i, const short& j)
{

#ifdef SUPPORT_DRIVEN_METHODS

  // First adjust head_support and tail_support.
  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);
  if(size_of_support_vectors>_number_of_variables)
    size_of_support_vectors=_number_of_variables;

#ifdef SUPPORT_VARIABLES_FIRST

  if(i<size_of_support_vectors)
    // else i is no support variable
  {
    if(exponent_vector[j]>0)
    // bit i will be 1 in the new head_support, 0 in the new
    // tail_support
    {
      head_support|=(1<<i);
      // bit i is set to 1
      tail_support&=~(1<<i);
      // bit i is set to 0
      // (in the complement ~(1<<i) all bits are 1 except from bit i)
    }

    if(exponent_vector[j]==0)
    // bit i will be 0 in the new head_support, 0 in the new
```

```
    // tail_support
    {
      head_support&=~(1<<i);
      // bit i is set to 0
      tail_support&=~(1<<i);
      // bit i is set to 0
    }

    if(exponent_vector[j]<0)
    // bit i will be 0 in the new head_support, 1 in the new
    // tail_support
    {
      head_support&=~(1<<i);
      // bit i is set to 0
      tail_support|=(1<<i);
      // bit i is set to 1
    }
  }

  if(j<size_of_support_vectors)
    // else j is no support variable
  {
    if(exponent_vector[i]>0)
    // bit j will be 1 in the new head_support, 0 in the new
    // tail_support
    {
      head_support|=(1<<j);
      // bit j is set to 1
      tail_support&=~(1<<j);
      // bit j is set to 0
      // (in the complement ~(1<<j) all bits are 1 except from bit j)
    }

    if(exponent_vector[i]==0)
    // bit j will be 0 in the new head_support, 0 in the new
    // tail_support
    {
      head_support&=~(1<<j);
      // bit j is set to 0
      tail_support&=~(1<<j);
      // bit j is set to 0
    }

    if(exponent_vector[i]<0)
    // bit j will be 0 in the new head_support, 1 in the new
    // tail_support
    {
```

```
      head_support&=~(1<<j);
      // bit j is set to 0
      tail_support|=(1<<j);
      // bit j is set to 1
    }
  }

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST

  // Using SUPPORT_VARIABLES_LAST, bit k of the support vectors
  // corresponds to exponent_vector[_number_of_variables-1-k],
  // hence bit _number_of_variables-1-i to exponent_vector[i].

  if(i>=_number_of_variables-size_of_support_vectors)
    // else i is no support variable
  {
    if(exponent_vector[j]>0)
    // bit _number_of_variables-1-i will be 1 in the new head_support,
    // 0 in the new tail_support
    {
      short k=_number_of_variables-1-i;
      head_support|=(1<<k);
      // bit _number_of_variables-1-i is set to 1
      tail_support&=~(1<<k);
      // bit _number_of_variables-1-i is set to 0
      // (in the complement ~(1<<(_number_of_variables-1-i)) all bits
      // are 1 except from bit _number_of_variables-1-i)
    }

    if(exponent_vector[j]==0)
    // bit _number_of_variables-1-i will be 0 in the new head_support,
    // 0 in the new tail_support
    {
      short k=_number_of_variables-1-i;
      head_support&=~(1<<k);
      // bit _number_of_variables-1-i is set to 0
      tail_support&=~(1<<k);
      // bit _number_of_variables-1-i is set to 0
    }

    if(exponent_vector[j]<0)
    // bit _number_of_variables-1-i will be 0 in the new head_support,
     // 1 in the new tail_support
    {
      short k=_number_of_variables-1-i;
```

```
      head_support&=~(1<<k);
      // bit _number_of_variables-1-i is set to 0
      tail_support|=(1<<k);
      // bit _number_of_variables-1-i is set to 1
    }
  }

  if(j>=_number_of_variables-size_of_support_vectors)
      // else j is no support variable
  {
   if(exponent_vector[i]>0)
    // bit _number_of_variables-1-j will be 1 in the new head_support,
    // 0 in the new tail_support
    {
      short k=_number_of_variables-1-j;
      head_support|=(1<<k);
      // bit _number_of_variables-1-j is set to 1
      tail_support&=~(1<<k);
      // bit _number_of_variables-1-j is set to 0
      // (in the complement ~(1<<(_number_of_variables-1-j)) all bits
      // are 1 except from bit _number_of_variables-1-j)
    }

    if(exponent_vector[i]==0)
    // bit _number_of_variables-1-j will be 0 in the new head_support,
    // 0 in the new tail_support
    {
      short k=_number_of_variables-1-j;
      head_support&=~(1<<k);
      // bit _number_of_variables-1-j is set to 0
      tail_support&=~(1<<k);
      // bit _number_of_variables-1-j is set to 0
    }

    if(exponent_vector[i]<0)
    // bit _number_of_variables-1-j will be 0 in the new head_support,
     // 1 in the new tail_support
    {
      short k=_number_of_variables-1-j;
      head_support&=~(1<<k);
      // bit _number_of_variables-1-j is set to 0
      tail_support|=(1<<k);
      // bit _number_of_variables-1-j is set to 1
    }
  }

#endif  // SUPPORT_VARIABLES_LAST
```

```
#endif  // SUPPORT_DRIVEN_METHODS

// Now swap the components.

  Integer swap=exponent_vector[j];
  exponent_vector[j]=exponent_vector[i];
  exponent_vector[i]=swap;

  return *this;
}

binomial& binomial::flip_variable(const short& i)
{
  if(exponent_vector[i]==0)
    // binomial does not involve variable to flip
    return *this;

#ifdef SUPPORT_DRIVEN_METHODS

// First adjust head_support and tail_support.

  short size_of_support_vectors=CHAR_BIT*sizeof(unsigned long);
  if(size_of_support_vectors>_number_of_variables)
    size_of_support_vectors=_number_of_variables;

#ifdef SUPPORT_VARIABLES_FIRST

  if(i<size_of_support_vectors)
    // else i is no support variable
  {
    if(exponent_vector[i]>0)
      // variable i will be moved from head to tail
    {
      head_support&=~(1<<i);
      // bit i is set to 0
      tail_support|=(1<<i);
      // bit i is set to 1
    }

    else
      // variable i will be moved from tail to head
      // remember that exponent_vector[i]!=0
    {
      tail_support&=~(1<<i);
      // bit i is set to 0
      head_support|=(1<<i);
```

```
      // bit i is set to 1
    }
  }

#endif  // SUPPORT_VARIABLES_FIRST

#ifdef SUPPORT_VARIABLES_LAST

  // Using SUPPORT_VARIABLES_LAST, bit k of the support vectors
  // corresponds to exponent_vector[_number_of_variables-1-k],
  // hence bit _number_of_variables-1-i to exponent_vector[i].

  if(i>=_number_of_variables-size_of_support_vectors)
    // else i is no support variable
  {
    if(exponent_vector[i]>0)
    // variable i will be moved from head to tail
    {
      short k=_number_of_variables-1-i;
      head_support&=~(1<<k);
      // bit _number_of_variables-1-i is set to 0
      tail_support|=(1<<k);
      // bit _number_of_variables-1-i is set to 1

     }

    else
    // variable i will be moved from tail to head
    {
      short k=_number_of_variables-1-i;
      tail_support&=~(1<<k);
      // bit _number_of_variables-1-i is set to 0
      head_support|=(1<<k);
      // bit _number_of_variables-1-i is set to 1
    }
  }

#endif  // SUPPORT_VARIABLES_LAST

#endif  // SUPPORT_DRIVEN_METHODS

  // Now flip the variable.
  exponent_vector[i]*=-1;
}

////////////////////// output /////////////////////////////////////
```

```
... // output routines

#endif  // BINOMIAL_CC
```

A.2.2 Buchberger.cc

```
// Buchberger.cc

// implementation of Buchberger's Algorithm.

#ifndef BUCHBERGER_CC
#define BUCHBERGER_CC

#include "ideal.h"

////////////////////////////////////////////////////////////////////
////////// S-pair computation //////////////////////////////////////
////////////////////////////////////////////////////////////////////

BOOLEAN ideal::unnecessary_S_pair(list_iterator& first_iter,
                                  list_iterator& second_iter) const
{
// This function checks several criteria to discard th S-pair of the
// binomials referenced by the iterators. The criteria depend on the
// settings of the ideal's S-pair flags.
// The arguments are iterators instead of the referenced binomials
// because we have to do some equality tests. These are more efficient
// on iterators than on binomials.

////////// criterion of relatively prime leading terms //////////////

  // An S-pair can discarded if the leading terms of the two binomials
  // are relatively prime.
  if(rel_primeness)
    if(relatively_prime(first_iter.get_element(),
                        second_iter.get_element())==TRUE)
      return TRUE;

////////// criterion M ///////////////////////////////////////////////

  if(M_criterion)
  {
    list_iterator iter;
    binomial& bin1=first_iter.get_element();
    binomial& bin2=second_iter.get_element();

    // The M-criterion of Gebauer/Moeller checks binomial triples as
```

```
    // explained in binomial.h; these are built of the elements
    // referenced by the argument iterators and a third element
    // appearing before the element referenced by second_iter in the
    // generator lists.

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

    iter.set_to_list(generators);

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

    // The support of the lcm of two monomials is the union of their
    // supports. To test criterion M, we then only have to consider
    // lists whose support is a subset of the union of
    // (first_iter.get_element()).head_support and
    // (second_iter.get_element()).head_support. As only elements
    // before second_iter.get_element() are tested, we can stop
    // iteraton as soon as we reach this element.

    short supp2=bin2.head_support%Number_of_Lists;
    short supp_union=(bin1.head_support%Number_of_Lists)|supp2;
    // supp_union (read as binary vector) is the union of the supports
    // of first_iter.get_element() and second_iter.get_element()
    // (restricted to List_Support_Variables variables).

    for(short i=0;i<S.number_of_subsets[supp_union];i++)
      // Go through the lists that contain elements whose support is a
      // subset of supp_union.
    {
      short actual_list=S.subsets_of_support[supp_union][i];
      iter.set_to_list(generators[actual_list]);
      // This is the i-th list among the generator list with elements
      // whose support is a subset of supp_union.

      if(actual_list==supp2)
        break;
      // The iteration has reached the list referenced by second_iter,
      // this is handled alone to avoid unnecessary checks.
      // Before breakin the loop, iter has to be set to this list.

      while(iter.is_at_end()==FALSE)
        // Iterate over the list with three iterators according to the
        // description of criterion M.
      {
        if(M(iter.get_element(),bin1,bin2)==TRUE)
```

```
          return TRUE;
        iter.next();
      }
    }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

    // Now, iter references second_iter's list,
    // if SUPPORT_DRIVEN_METHODS_EXTENDED are enabled or not.
    while(iter!=second_iter)
    {
      if(M(iter.get_element(),bin1,bin2)==TRUE)
        return TRUE;
      iter.next();
    }
  }

/////////////////// criterion F ///////////////////////////////////

  if(F_criterion)
  {
    list_iterator iter;
    binomial& bin1=first_iter.get_element();
    binomial& bin2=second_iter.get_element();

    // The F-criterion of Gebauer/Moeller checks binomial triples as
    // explained in binomial.h; these are built of the elements
    // referenced by the argument iterators and a third element
    // appearing before the element referenced by first_iter in the
    // generator lists.

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

    iter.set_to_list(generators);

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

    // Again, we only have to consider lists whose support is a subset
    // of the union of (first_iter.get_element()).head_support and
    // (second_iter.get_element()).head_support.
    // Additionally,we can override lists whose support is to small.

    short supp1=bin1.head_support%Number_of_Lists;
    short supp2=bin2.head_support%Number_of_Lists;
    short supp_union=supp1|supp2;
```

```
    // supp_union (read as binary vector) is the union of the supports
    // of first_iter.get_element() and second_iter.get_element()
    // (restricted to List_Support_Variables variables).

    for(short i=0;i<S.number_of_subsets[supp_union];i++)
      // Go through the lists that contain elements whose support is a
      // subset of supp_union.
    {
      short actual_list=S.subsets_of_support[supp_union][i];

      if((actual_list|supp2) != supp_union)
        continue;
      // The support of the actual list is too small, so its elements
      // cannot satisfie criterion F.

      iter.set_to_list(generators[actual_list]);
      // This is the i-th list among the generator list with elements
      // whose support is a subset of supp_union.

      if(actual_list==supp1)
        break;
      // The iteration has reached the list referenced by first_iter;
      // this is handled alone to avoid unnecessary checks.
      // iter has to be set to that list before breaking the loop.

      while(iter.is_at_end()==FALSE)
        // Iterate over the list with three iterators according to the
        // description of criterion F.
      {
        if(F(iter.get_element(),bin1,bin2)==TRUE)
          return TRUE;
        iter.next();
      }
    }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

    // Now, iter references first_iter's list,
    // if SUPPORT_DRIVEN_METHODS_EXTENDED are enabled or not.
    while(iter!=first_iter)
    {
      if(F(iter.get_element(),bin1,bin2)==TRUE)
        return TRUE;
      iter.next();
    }
  }
```

```
/////////////////// criterion B //////////////////////////////////

  if(B_criterion)
  {
    list_iterator iter;
    binomial& bin1=first_iter.get_element();
    binomial& bin2=second_iter.get_element();

    // The B-criterion of Gebauer/Moeller checks binomial triples as
    // explained in binomial.h; these are built of the elements
    // referenced by the argument iterators and a third element
    // appearing after the element referenced by second_iter in the
    // generator lists.

    iter=second_iter;
    iter.next();

    // First test second_iter's list.
    // This is the only list if NO_SUPPORT_DRIVEN_METHODS are enabled.
    while(iter.is_at_end()==FALSE)
    {
      if(B(iter.get_element(),bin1,bin2)==TRUE)
        return(TRUE);
      iter.next();
    }

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

    // Now consider the other lists.
    // Again, we only have to consider lists whose support is a subset
    // of the union of (first_iter.get_element()).head_support and
    // (second_iter.get_element()).head_support.

    short supp2=bin2.head_support%Number_of_Lists;
    short supp_union=(bin1.head_support%Number_of_Lists)|supp2;
    // supp_union (read as binary vector) is the union of the supports
    // of first_iter.get_element() and second_iter.get_element()
    // (restricted to List_Support_Variables variables).

    for(int i=0;i<S.number_of_subsets[supp_union];i++)
      // Go through the lists that contain elements whose support is a
      // subset of supp_union.
    {
      short actual_list=S.subsets_of_support[supp_union][i];

      if(actual_list<=supp2)
        continue;
```

```
      // Only lists after second_iter's list have to be considered.

      iter.set_to_list(generators[actual_list]);
      // This is the i-th list among the generator list with elements
      // whose support is a subset of supp_union.

      while(iter.is_at_end()==FALSE)
      {
        // Iterate over the list with three iterators according to the
        // description of criterion B.
        if(B(iter.get_element(),bin1,bin2)==TRUE)
          return TRUE;
        iter.next();
      }
    }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  }

////////////////// Buchberger's second criterion //////////////////

  if(second_criterion)
  {
    list_iterator iter;
    binomial& bin1=first_iter.get_element();
    binomial& bin2=second_iter.get_element();

    // The Buchberger's second criterion checks binomial triples as
    // explained in binomial.h; these are built of the elements
    // referenced by the argument iterators and a third element
    // appearing anywhere in the generator lists.

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

    iter.set_to_list(generators);
    while(iter.is_at_end()==FALSE)
      // Iterate over the list with three iterators according to the
      // description of the second criterion.
    {
      if((iter!=first_iter) && (iter!=second_iter))
        // Else the second criterion must not be applied
        // (lcm(a,b) is, of course, divisible by a and by b).
        if(second_crit(iter.get_element(),bin1,bin2)==TRUE)
          return TRUE;
      iter.next();
    }
```

```
#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

    // Again, we only have to consider lists whose support is a subset
    // of the union of (first_iter.get_element()).head_support
    // and (second_iter.get_element()).head_support.

    short supp1=bin1.head_support%Number_of_Lists;
    short supp2=bin2.head_support%Number_of_Lists;
    short supp_union=supp1|supp2;
    // supp_union (read as binary vector) is the union of the supports
    // of first_iter.get_element() and second_iter.get_element()
    // (restricted to List_Support_Variables variables)

    for(short i=0;i<S.number_of_subsets[supp_union];i++)
      // Go through the lists that contain elements whose support is a
      // subset of supp_union.
    {
      short actual_list=S.subsets_of_support[supp_union][i];
      if((actual_list==supp1) || (actual_list==supp2))
        continue;
      // The lists containing the elements referenced by the argument
      // iterators are tested separately to avoid unnecessary checks
      // for equality.

      iter.set_to_list(generators[actual_list]);
      // This is the i-th list among the generator list with elements
      // whose support is a subset of supp_union.

      while(iter.is_at_end()==FALSE)
        // Iterate over the list with three iterators according to the
        // description of the second criterion.
      {
        if(second_crit(iter.get_element(),bin1,bin2)==TRUE)
          return TRUE;
        iter.next();
      }
    }

    if(supp1==supp2)
      // The elements referenced by first_iter and second_iter appear
      // in the same list.
    {
      iter.set_to_list(generators[supp1]);
      while(iter.is_at_end()==FALSE)
```

```
      {
        if((iter!=first_iter) && (iter!=second_iter))
          // Else the second criterion must not be applied
          // (lcm(a,b) is, of course, divisible by a and by b).
          if(second_crit(iter.get_element(),bin1,bin2)==TRUE)
            return TRUE;
        iter.next();
      }
    }
    else
      // The elements referenced by first_iter and second_iter appear
      // in different lists.
    {
      // Test first_iter's list.
      iter.set_to_list(generators[supp1]);
      while(iter.is_at_end()==FALSE)
      {
        if(iter!=first_iter)
          // Else the second criterion must not be applied
          // (lcm(a,b) is, of course, divisible by a and by b).
          if(second_crit(iter.get_element(),bin1,bin2)==TRUE)
            return TRUE;
        iter.next();
      }

      // Test second_iter's list.
      iter.set_to_list(generators[supp2]);
      while(iter.is_at_end()==FALSE)
      {
        if(iter!=second_iter)
          // Else the second criterion must not be applied
          // (lcm(a,b) is, of course, divisible by a and by b).
          if(second_crit(iter.get_element(),bin1,bin2)==TRUE)
            return TRUE;
        iter.next();
      }
    }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  }

  // no criterion found to discard the S-Pair to compute
  return FALSE;
}
```

```
ideal& ideal::compute_actual_S_pairs_1()
{
// This routine implements the simplest method for the S-pair
// computation (and one of the most efficient methods). We simply
// iterate over the  generator list(s) with two pointers to look at
// each binomial pair. The "done"-mark of each list element tells us
// if this element was already considered in a previous S-pair
// computation; pairs of such "old" binomials do not have to be
// computed anymore (but pairs of an old and a new one have to be
// computed, of course). As the generator list are ordered with
// respect to the "done"-flag (all undone elements preceed all done
// elements), we can avoid unnecessary iteration steps by breaking
// the iteration at the right point.
// The computed S-pairs are stored in the aux_list for further
// computations.
// For a better overview, the code for
// NO_SUPPORT_DRIVEN_METHODS_EXTENDED and
// SUPPORT_DRIVEN_METHODS_EXTENDED is completetly separated in this
// function.
// Note that the "next()"-operations in the following routine do not
// reach a NULL pointer because of the implementation of the
// "is_at_end()"-function and because the "done"-component of the
// dummy element is set to zero.

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  list_iterator first_iter(generators);
  while(first_iter.element_is_marked_done()==FALSE)
    // new generator, compute S-pairs with all following generators
    // Notice that the new generators are always at the beginning,
    // the old generators at the end of the generator list.
  {
    binomial& bin=first_iter.get_element();
    first_iter.mark_element_done();
    list_iterator second_iter(first_iter);
    second_iter.next();
    // This may be the dummy element.

    while(second_iter.is_at_end()==FALSE)
    {
      if(unnecessary_S_pair(first_iter,second_iter)==FALSE)
         aux_list._insert(S_binomial(bin,second_iter.get_element(),w));
      second_iter.next();
    }
    first_iter.next();
  }
```

```
  // Now, first_iter references an old generator or the end of the
  // generator list. As all following generators are old ones, we are
  // done.

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  list_iterator first_iter;

  for(short i=0;i<Number_of_Lists;i++)
  {
    first_iter.set_to_list(generators[i]);

    while(first_iter.element_is_marked_done()==FALSE)
      // new generator, compute S-pairs with all following elements
      // Notice that the new generators are always at the beginning,
      // the old generators at the end of the generator lists.
    {
      binomial& bin=first_iter.get_element();
      first_iter.mark_element_done();

      // First search over the actual list with the second iterator.
      list_iterator second_iter(first_iter);
      second_iter.next();

      while(second_iter.is_at_end()==FALSE)
      {
        if(unnecessary_S_pair(first_iter,second_iter)==FALSE)
          aux_list._insert(S_binomial(bin,second_iter.get_element(),w));
      second_iter.next();
      }

      // Then search over the remaining lists.
      for(short j=i+1;j<Number_of_Lists;j++)
      {
        second_iter.set_to_list(generators[j]);
        while(second_iter.is_at_end()==FALSE)
        {
          if(unnecessary_S_pair(first_iter,second_iter)==FALSE)
            aux_list._insert(S_binomial(bin,second_iter.get_element(),w));
          second_iter.next();
        }
      }
      first_iter.next();
    }
```

```
    while(first_iter.is_at_end()==FALSE)
      // old generator, compute only S-pairs with the following new
      // generators (S-pairs with the following old generators were
      // already computed before,
      // first_iter.element_is_marked_done()==TRUE)
      // As all generators in the actual list are old ones, we can
      // start iteration with the next list.
    {
      binomial& bin=first_iter.get_element();
      list_iterator second_iter;

      for(short j=i+1;j<Number_of_Lists;j++)
        // search over remaining lists
      {
        second_iter.set_to_list(generators[j]);
        while(second_iter.element_is_marked_done()==FALSE)
          // consider only new generators
        {
          if(unnecessary_S_pair(first_iter,second_iter)==FALSE)
            aux_list._insert(S_binomial(bin,second_iter.get_element(),w));
          second_iter.next();
        }
      }
      first_iter.next();
    }
  }
#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  return(*this);
}

ideal& ideal::compute_actual_S_pairs_1a()
{
// The only difference to the previous routine is that the aux_list is
// kept ordered with respect to the ideal's term ordering, i.e. the
// inserts are replaced by ordered inserts.

... // code as above, insert(..) replaced by ordered_insert(...)

}

ideal& ideal::compute_actual_S_pairs_2()
{
// This routine implements are more dynamic S-pair-computation. As in
// the previous routines, we iterate over the generator list(s) with
// two iterators, forming pairs under the consideration of the
// "done"-flags.
```

```
// But before inserting into the aux_list, these S-pairs are reduced
// by the ideal generators. This seems to be clever, but shows to be a
// disadvantage:
// In the previous S-pair routines, the computed S-bionomials are not
// reduced at all. This is done in the appropriate Groebner basis
// routine (reduced_Groebner_basis_1 or ..._1a) when moving them from
// the aux_list to the generator lists. This meens that S-binomials
// cannot only be reduced by the generators known at the time of their
// computation, but also by the S-pairs that where already
// treated.
// The advantage of the current routine is that the immediately
// reduced S-binomial can be used to reduce the ideal itself. This
// strategy keeps the ideal almost reduced, so the minimalization will
// be faster. Furthermore, the computation of S-pairs with unreduced
// generators is avoided.
// To provide a possibility to compensate the mentionned disadvantage,
// I have written the routine minimalize_S_pairs() that interreduces
// the binomials stored in aux_list.

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  list_iterator first_iter(generators);
  Integer first_reduced=0;
  Integer second_reduced=0;
  // When reducing the ideal immediately by newly found generators, it
  // can happen that the binomials referenced by the iterators are
  // reduced to zero and then deleted. We need to make sure that the
  // iterators do not reference freed memory in such a case. These two
  // flags help us with this task.

  while(first_iter.element_is_marked_done()==FALSE)
    // new generator, compute S-pairs with all following generators
  {
    binomial& bin1=first_iter.get_element();
    first_iter.mark_element_done();
    list_iterator second_iter(first_iter);
    second_iter.next();
    // This may be the dummy element.

    while((second_iter.is_at_end()==FALSE) && (first_reduced<=0))
    {
      if(unnecessary_S_pair(first_iter,second_iter)==FALSE)
      {
        binomial& bin2=second_iter.get_element();

        // compute S-binomial
        binomial& S_bin=S_binomial(bin1,bin2,w);
```

```
        // reduce S-binomial by the actual ideal generators
        reduce(S_bin,FALSE);

        if(S_bin!=0)
        {
          // reduce the ideal generators by the S-binomial
          first_reduced=bin1.head_reductions_by(S_bin);
          second_reduced=bin2.head_reductions_by(S_bin);
          reduce_by(S_bin,first_iter,second_iter);
          aux_list._insert(S_bin);
        }
        else
          delete &S_bin;
      }

      // Move second_iter to the next element if its referenced
      // binomial has not changed (if it has changed, the binomial was
      // moved to the aux_list during the reduce_by(...)-procedure,
      // and second_iter already references a new binomial).
      if(second_reduced<=0)
        second_iter.next();
      else
        second_reduced=0;
    }

    // same procedure for first_iter
    if(first_reduced<=0)
      first_iter.next();
    else
      first_reduced=0;
  }

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  list_iterator first_iter;
  Integer first_reduced=0;
  Integer second_reduced=0;
  // When reducing the ideal immediately by newly found generators, it
  // can happen that the binomials referenced by the iterators are
  // changed (including their support!) or even reduced to zero and
  // then deleted. We need to make sure that the iterators do not
  // reference freed memory in such a case. These two flags help us
  // with this task.
```

```
for(short i=0;i<Number_of_Lists;i++)
{
  first_iter.set_to_list(generators[i]);

  while(first_iter.element_is_marked_done()==FALSE)
    // new generator, compute S-pairs with all following elements
  {
    binomial& bin1=first_iter.get_element();
    first_iter.mark_element_done();
    list_iterator second_iter(first_iter);
    second_iter.next();

    // First search over the actual list.
    while((second_iter.is_at_end()==FALSE) && (first_reduced<=0))
    {
      if(unnecessary_S_pair(first_iter,second_iter)==FALSE)
      {
        binomial& bin2=second_iter.get_element();

        // compute S-binomial
        binomial& S_bin=S_binomial(bin1,bin2,w);

        // reduce S-binomial by the actual ideal generators
        reduce(S_bin,FALSE);

        if((S_bin)!=0)
        {
          // reduce the ideal generators by the S-binomial
          first_reduced=bin1.head_reductions_by(S_bin);
          second_reduced=bin2.head_reductions_by(S_bin);
          reduce_by(S_bin,first_iter,second_iter);
          aux_list._insert(S_bin);
        }
        else
          delete &S_bin;
      }

      // Move second_iter to the next element if its referenced
      // binomial has not changed (if it has changed, the binomial
      // was moved to the aux_list during the
      // reduce_by(...)-procedure, and second_iter already
      // references a new binomial).
      if(second_reduced<=0)
        second_iter.next();
      else
        second_reduced=0;
    }
```

```
    // Then search over the remaining lists.
    for(short j=i+1;(j<Number_of_Lists) && (first_reduced<=0);j++)
    {
      second_iter.set_to_list(generators[j]);

      while((second_iter.is_at_end()==FALSE) && (first_reduced<=0))
      {
        if(unnecessary_S_pair(first_iter,second_iter)==FALSE)
        {
          binomial& bin2=second_iter.get_element();

          // compute S-binomial
          binomial& S_bin=S_binomial(bin1,bin2,w);

          // reduce S-binomial by the actual ideal generators
          reduce(S_bin,FALSE);

          if((S_bin)!=0)
          {
            // reduce the ideal generators by the S-binomial
            first_reduced=bin1.head_reductions_by(S_bin);
            second_reduced=bin2.head_reductions_by(S_bin);
            reduce_by(S_bin,first_iter,second_iter);
            aux_list._insert(S_bin);
          }
          else
            delete& S_bin;
        }

        // Move second_iter to the next element if its referenced
        // binomial has not changed.
        if(second_reduced<=0)
          second_iter.next();
        else
          second_reduced=0;
      }
    }

    // same procedure for first_iter
    if(first_reduced<=0)
      first_iter.next();
    else
      first_reduced=0;
  }

  while(first_iter.is_at_end()==FALSE)
```

```
  // old generator, compute only S-pairs with the following new
  // generators
  // As all generators in the actual list are old ones, we can
  // start iteration with the next list.
{
  binomial& bin1=first_iter.get_element();

  list_iterator second_iter(first_iter);
  second_iter.next();

 for(short j=i+1;(j<Number_of_Lists) && (first_reduced<=0);j++)
  {
    second_iter.set_to_list(generators[j]);

    while((second_iter.element_is_marked_done()==FALSE) &&
          (first_reduced<=0))
      // consider only new generators
    {
      if(unnecessary_S_pair(first_iter,second_iter)==FALSE)
      {
        binomial& bin2=second_iter.get_element();

        // compute S-binomial
        binomial& S_bin=S_binomial(bin1,bin2,w);

        // reduce S-binomial by the actual ideal generators
        reduce(S_bin,FALSE);

        if((S_bin)!=0)
        {
          // reduce the ideal generators by the S-binomial
          first_reduced=bin1.head_reductions_by(S_bin);
          second_reduced=bin2.head_reductions_by(S_bin);
          reduce_by(S_bin,first_iter,second_iter);
          aux_list._insert(S_bin);
        }
        else
          delete& S_bin;
      }

      // Move second_iter to the next element if its referenced
      // binomial has not changed.
      if(second_reduced<=0)
        second_iter.next();
      else
        second_reduced=0;
    }
```

```
      }

      // same procedure for first_iter
      if(first_reduced<=0)
        first_iter.next();
      else
        first_reduced=0;
    }
  }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  return(*this);
}

ideal& ideal::compute_actual_S_pairs_3()
{
// This routine is quite similar to the preceeding.
// The main difference is that the computed S-binomials are not stored
// in the aux_list, but in new_generators. This makes a difference
// when minimalizing the S-binomials in the appropriate Groebner basis
// routine (reduced_Groebner_basis_3) with the help of the procedure
// minimalize_new_generators(...).
// If NO_SUPPORT_DRIVEN_METHODS_EXTENDED are enabled, only the
// organization of the minimalization (in the routine
// reduced_Groebner_basis(...)) is different.
// If SUPPORT_DRIVEN_METHODS_EXTENDED are enabled, the minimalization
// can use the support information.

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

... // as in ideal& ideal::compute_actual_S_pairs_2(),
    // aux_list._insert(S_bin); replaced by add_new_generator(S_bin);

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

... // as in ideal& ideal::compute_actual_S_pairs_2(),
    // aux_list._insert(S_bin); replaced by add_new_generator(S_bin);

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  // During the reduce_by(...)-routines, some reduced generators were
  // perhaps moved to the aux_list. This list is emptied now: Like the
  // computed S-pairs, its elements are moved to the list(s)
  // new_generators. To avoid this construction, we would have to
```

```
  // write a second version of the reduce_by(...)-procedure. The
  // efficiency gains would however not be considerable.

  list_iterator iter(aux_list);
  while(iter.is_at_end()==FALSE)
  {
    add_new_generator(iter.get_element());
    iter.extract_element();
  }
  return(*this);
}

////////////////////////////////////////////////////////////////////
///////// minimalization / autoreduction ///////////////////////////
////////////////////////////////////////////////////////////////////

ideal& ideal::reduce_by(const binomial& bin,
                        list_iterator& first_iter,
                        list_iterator& second_iter)
{
// This routine reduces the ideal by the argument binomial and takes
// care that the argument list iterators are not corrupted.
// Only head reductions are performed.

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  list_iterator iter(generators);
  Integer reduced;

  while(iter.is_at_end()==FALSE)
  {
    binomial& actual=iter.get_element();

    // reduce actual binomial by bin
    reduced=actual.head_reductions_by(bin);
    if(reduced<=0)
      iter.next();
    else
      // the actual binomial has changed and will be removed or
      // moved to the aux_list
    {

#ifdef SL_LIST

      // If we use a simply linked list, we have to take care of the
      // following binomial.
      if(iter.next_is(first_iter)==TRUE)
```

```
        first_iter=iter;
      if(iter.next_is(second_iter)==TRUE)
        second_iter=iter;

#endif  // SL_LIST

#ifdef DL_LIST

      // If we use a doubly linked list, we have to take care of the
      // binomial itself.
      if(iter==first_iter)
        first_iter.next();
      if(iter==second_iter)
        second_iter.next();

#endif  // DL_LIST

      // move changed generator to the aux_list or delete it
      if(actual==0)
        iter.delete_element();
      else
      {
        aux_list._insert(actual);
        iter.extract_element();
      }
      size--;
    }
  }

#endif  // NO_SUPPORT_DRIVEN_METHOD_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  short supp1=(first_iter.get_element()).head_support%Number_of_Lists;
  short supp2=(second_iter.get_element()).head_support%Number_of_Lists;

  // Determine the lists over which we have to iterate.
  // These are the lists with elements whose support contains the
  // support of bin.
  // List i has to be checked iff the set of bits in i that are 1
  // contains the set of bits in supp that are 1.
  // Equivalent: List i has to be checked iff the set of bits in i
  // that are 0 is contained in the set of bits of supp that are 0.
  // With this formulation, we can use the subset tree as follows:

  short supp=bin.head_support%Number_of_Lists;
  short inv_supp=Number_of_Lists-supp-1;
```

```
// This bit vector is the bitwise inverse of bin.head_support
// (restricted to the variables considered in the list indices.

list_iterator iter;
Integer reduced;

for(short i=0;i<S.number_of_subsets[inv_supp];i++)
{
  short actual_list=
        Number_of_Lists-S.subsets_of_support[inv_supp][i]-1;
  // the actual list for iteration
  // The support of S.subsets_of_support[inv_supp][i] as a bit
  // vector is contained in that of inv_supp.
  // I.e. the support of
  // Number_of_Lists-S.subsets_of_support[inv_supp][i]-1
  // which is the bitwise inverse of
  // S.subsets_of_support[inv_supp][i]
  // contains the support of the bitwise inverse of inv_supp, hence
  // the support of supp.

  if((actual_list==supp1) || (actual_list==supp2))
    // The lists referenced by first_iter and second_iter are tested
    // separately to avoid unnecessary checks.
    continue;
  iter.set_to_list(generators[actual_list]);

  while(iter.is_at_end()==FALSE)
  {
    binomial& actual=iter.get_element();

    // reduce actual binomial by bin
    reduced=actual.head_reductions_by(bin);
    if(reduced<=0)
      iter.next();
    else
      // the actual binomial has changed and will be removed or
    {
      if(actual==0)
        iter.delete_element();
      else
      {
        aux_list._insert(actual);
        iter.extract_element();
      }
      size--;
    }
  }
```

```
  }

  // Now test the lists referenced by first_iter and second_iter.
  if(supp1==supp2)
    // first_iter and second_iter reference the same list
  {
    if((supp1|supp)==supp1)
      // support of bin contained in that of the element referenced by
      // first_iter, else this list has not to be tested
    {
      iter.set_to_list(generators[supp1]);

      while(iter.is_at_end()==FALSE)
      {
        binomial& actual=iter.get_element();
        reduced=actual.head_reductions_by(bin);
        if(reduced<=0)
          iter.next();
        else
        {

#ifdef SL_LIST

          if(iter.next_is(first_iter)==TRUE)
            first_iter=iter;
          if(iter.next_is(second_iter)==TRUE)
            second_iter=iter;

#endif  // SL_LIST

#ifdef DL_LIST

          if(iter==first_iter)
            first_iter.next();
          if(iter==second_iter)
            second_iter.next();

#endif  // DL_LIST

          if(actual==0)
            iter.delete_element();
          else
          {
            aux_list._insert(actual);
            iter.extract_element();
          }
          size--;
```

```
        }
      }
    }
  }

  else
    // first_iter and second_iter reference different lists
  {
    if((supp1|supp)==supp1)
      // support of bin contained in that of the element referenced by
      // first_iter, else this list has not to be tested
    {
      iter.set_to_list(generators[supp1]);

      while(iter.is_at_end()==FALSE)
      {
        binomial& actual=iter.get_element();
        reduced=actual.head_reductions_by(bin);
        if(reduced<=0)
          iter.next();
        else
        {

#ifdef SL_LIST

          if(iter.next_is(first_iter)==TRUE)
            first_iter=iter;

#endif  // SL_LIST

#ifdef DL_LIST

          if(iter==first_iter)
            first_iter.next();

#endif  // DL_LIST

          if(actual==0)
            iter.delete_element();
          else
          {
            aux_list._insert(actual);
            iter.extract_element();
          }
          size--;
        }
      }
```

```
    }

    if((supp2|supp)==supp2)
      // support of bin contained in that of the element referenced by
      // second_iter, else this list has not to be tested
    {
      iter.set_to_list(generators[supp2]);

      while(iter.is_at_end()==FALSE)
      {
        binomial& actual=iter.get_element();
        reduced=actual.head_reductions_by(bin);
        if(reduced<=0)
          iter.next();
        else
        {

#ifdef SL_LIST

          if(iter.next_is(second_iter)==TRUE)
            second_iter=iter;

#endif  // SL_LIST

#ifdef DL_LIST

          if(iter==second_iter)
            second_iter.next();

#endif  // DL_LIST

          if(actual==0)
            iter.delete_element();
          else
          {
            aux_list._insert(actual);
            iter.extract_element();
          }
          size--;
        }
      }
    }
  }

#endif

  return *this;
```

```
}

ideal& ideal::minimalize_S_pairs()
{
// This routine implements a very simple minimalization method. We
// iterate over the S-pair lists with two iterators, interreducing the
// two referenced binomials. Remember that the S-pair list does not
// use the "head_reduced"-flags. The iteration is repeated as long as
// some interreduction is done.

  list_iterator first_iter;
  short found;
  // to control if a reduction has occurred during the actual
  // iteration

  do
  {
    first_iter.set_to_list(aux_list);
    found=0;
    // no reduction occured yet

    while(first_iter.is_at_end()==FALSE)
    {
      binomial& bin1=first_iter.get_element();
      short first_changed=0;
      // to control if the first element has been reduced

      // look at all following binomials
      list_iterator second_iter(first_iter);
      second_iter.next();
      // this may be the dummy element

      while(second_iter.is_at_end()==FALSE)
      {
        binomial& bin2=second_iter.get_element();
        short second_changed=0;
        if(bin1.reduce_head_by(bin2,w)!=0)
          // head of first binomial can be reduced by second
        {
          if(bin1!=0)
            found=1;
            // found has not to be set if a binomial is reduced to
            // zero (then there are no new binomials)
          else
            // the binomial referenced by bin1 is zero
          {
```

```
#ifdef SL_LIST

            if(first_iter.next_is(second_iter))
              second_iter.next();

#endif  // SL_LIST

            first_iter.delete_element();
            first_changed=1;
          }

          break;
          // Breaks the while-loop.
          // As the element referenced by first_iter has changed,
          // the iteration with the second iterator can be restarted.
          // (We try to reduce as many elements as possible in one
          // iteration.)
        }

        if(bin2.reduce_head_by(bin1,w)!=0)
           // head of second binomial can be reduced by first
        {
          if(bin2!=0)
            found=1;
            // found has not to be set if a binomial is reduced to
            // zero (then there are no new binomials)
          else
            // binomial referenced by bin2 is zero
          {
            second_iter.delete_element();
            second_changed=1;
          }
          // As the second iterator always references an element
          // coming after first_iter's element in the generator list,
          // we do not pay attention to the deletion...
        }
        if(second_changed==0)
          second_iter.next();
      }

      // Now second_iter has reached the end of the generator list or
      // the element referenced by first_iter has been reduced...The
      // iteration is continued with a new (or changed) first
      // binomial.
      if(first_changed==0)
        first_iter.next();
    }
```

```
 }
  while(found==1);
  // When leaving the loop, no generators have been interreduced
  // during the last iteration.

  return *this;
}

ideal& ideal::minimalize_new_generators()
{
// This routine is very similar to the following one, minimalize().
// The only difference is that we interreduce the elements stored in
// new_generators instead of those stored in generators.
// The size of the ideal has not to be manipulated hereby.

... // analogopus to ideal& ideal:: minimalize()
    // 1. generators replaced by new_generators
    // 2. generators.insert(bin); respectively
    //    generators[bin.head_support%Number_of_Lists].insert(bin);
    //    replaced by add_new_generator(bin);
    // 3. no change of size
}

ideal& ideal::minimalize()
{
// For a better overview, the code for
// NO_SUPPORT_DRIVEN_METHODS_EXTENDED and
// SUPPORT_DRIVEN_METHODS_EXTENDED is completetly separated in this
// function. Note that th iteration methods are quite different for
// those two possibilities.

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

// For technical simplicity, the interreduction is done as follows:
// We iterate over the generators with two iterators.
// For each binomial pair, we examine if one binomial's head can be
// reduced by the other. If this is the case, the reducible binomial
// is reduced and moved to the aux_list if it is not 0, else deleted.
// After one iteration, the elements of the aux_list are reinserted;
// then the interreduction is restarted until no new elements are
// found.
// The above method of deleting and inserting is chosen for the
// following reasons:
// - The order undone elements - done elements in the generator lists
//   is not destroyed. Newly found elements are automatically marked
//   undone when they are reinserted.
//   The S-pair computation can as usually make use of this fact.
```

```
// - In analogy, the order head_unreduced elements - head_reduced
//   elements is not destroyed. Remark that an undone element can
//   never be reduced, as reduction is only done after an S-pair
//   computation. Newly found elements are automatically marked
//   unreduced. With the "head_reduced"-flag we make sure that any
//   binomial pair is tested only once for interreduction during the
//   whole algorithm.

  list_iterator first_iter;
  short found;
  // to control if a reduction has occurred during the actual
  // iteration

  do
  {
    first_iter.set_to_list(generators);
    found=0;
    // no reduction occured yet

    while(first_iter.element_is_marked_head_reduced()==FALSE)
      // only the first element is tested for this flag
      // the second may be an old one
    {
      binomial& bin1=first_iter.get_element();
      short first_changed=0;
      // to control if the first element has been reduced

      // look at all following binomials
      list_iterator second_iter(first_iter);
      second_iter.next();
      // this may be the dummy element

      while(second_iter.is_at_end()==FALSE)
      {
        binomial& bin2=second_iter.get_element();

        if(bin1.reduce_head_by(bin2,w)!=0)
          // head of first binomial can be reduced by second
        {

#ifdef SL_LIST

          // The binomial referenced by first_iter will be deleted or
          // extracted. When using a simply linked list, this also
          // affects the following element. We need to assure that
          // this element does not reference freed memory.
          if(first_iter.next_is(second_iter))
```

```
              second_iter.next();

#endif  // SL_LIST

          if(bin1!=0)
          {
            found=1;
            // found has not to be set if a binomial is reduced to
            // zero (then there are no new binomials to insert)
            aux_list._insert(bin1);
            first_iter.extract_element();
            // moved changed binomial to aux_list
          }
          else
            first_iter.delete_element();

          first_changed=1;
          size--;
          break;
          // As the binomial referenced by first_iter has changed,
          // the iteration with the second iterator can be restarted.
          // (We try to reduce as many elements as possible in one
          // iteration.)
        }

        if(bin2.reduce_head_by(bin1,w)!=0)
           // head of second binomial can be reduced by first
        {
          // Here we do not have to pay attention to the deletion or
          // the extraction of the element referenced by second_iter
          // because the element referenced by second_iter always
          // comes after the element referenced by first_iter in the
          // generator list.

          if(bin2!=0)
          {
            found=1;
            // found has not to be set if a binomial is reduced to
            // zero (then there are no new binomials to insert)
            aux_list._insert(bin2);
            second_iter.extract_element();
            // move the element referenced by second_iter to aux_list
          }
          else
            second_iter.delete_element();

          size--;
```

```
        // Here it makes not sense to restart iteration because the
        // deletion sets the pointers as desired.
      }
      else
        // no reduction possible
        second_iter.next();
    }

    // Now second_iter has reached the end of the generator list or
    // the binomial referenced first_iter has been reduced...The
    // iteration is continued with a new (or changed) first binomial.

    if(first_changed==0)
      first_iter.next();
  }

  // Now we have found all currently possible reductions.
  // The elements remaining in the generator list cannot be
  // interreduced and are marked head_reduced.

  first_iter.set_to_list(generators);
  while(first_iter.element_is_marked_head_reduced()==FALSE)
    {
      first_iter.mark_element_head_reduced();
      first_iter.next();
    }

  // Now reinsert reduced elements.
  first_iter.set_to_list(aux_list);

  while(first_iter.is_at_end()==FALSE)
  {
    binomial& bin=first_iter.get_element();
    reduce(bin,FALSE);
    // The binomial was only reduced by one other generator before
    // it was moved to aux_list. To reduce it by all other
    // generators now can diminish the number of iterations
    // (do-while-loop).

    if(bin==0)
      first_iter.delete_element();
    else
    {
      generators.insert(bin);
      // We do not call the add_generator(...)-routine because we do
      // not want number_of_new_binomials to be incremented.
      size++;
```

```
        first_iter.extract_element();
      }
    }
  }
  while(found==1);
  // When leaving the loop, no generators have been interreduced during
  // the last iteration; we are done.

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

// In this case, the reduction has to be organized in a different way
// to use the support information to its full extend. Without the
// support methods, we test interreduction symmetrically: if we have a
// pair of generators, we test if generator 1 can be reduced by
// generator 2 AND if generator 2 can be reduced by generator 1. So
// each unordered pair is considered only once.
// If we would implement the same for the support methods, the second
// head reduction could not use the support information.
// For this reason, we test ordered pairs: For a given generator, we
// try to reduce his head by all other generators, considering its
// head support vector.
// The method of moving changed binomials to aux_list and later
// reinserting them is kept. When using
// SUPPORT_DRIVEN_METHODS_EXTENDED, it is even necessary to choose
// such a method because the head support of some binomials may
// change, too, a fact that requires the list structure to be
// rebuilt.

  list_iterator first_iter;
  list_iterator second_iter;
  short found;
  // to control if a reduction has occurred during the actual
  // iteration

  do
  {
    found=0;
    // no reduction occurred yet

    for(short i=0;i<Number_of_Lists;i++)
    {
      first_iter.set_to_list(generators[i]);

// First try to reduce the binomials that are marked unreduced.
```

```
      while(first_iter.element_is_marked_head_reduced()==FALSE)
        // all other elements have to be tested for interreduction
      {
        binomial& bin=first_iter.get_element();
        Integer changed=0;
        // binomial referenced by bin not yet reduced

// Look for head reductions:
// Iterate over the lists that could contain reducers for the head of
// bin. The list containing bin is tested separately to avoid an
// interreduction of a binomial by itself respectively unnecessary
// checks for this when the two iterators reference different lists.

        for(short j=0;(j<S.number_of_subsets[i]-1)&&(changed==0);j++)
        {
          second_iter.set_to_list(generators
                                  [S.subsets_of_support[i][j]]);
          // This is the j-th list among the generator lists with
          // elements whose support is a subset of that of i.
          // The support of i is just the head support of bin
          // (restricted to the corresponding variables).

          while((second_iter.is_at_end()==FALSE)&&(changed==0))
          {
            changed=bin.reduce_head_by(second_iter.get_element(),w);

            if(changed!=0)
              // bin has been reduced
            {
              if(bin!=0)
              {
                found=1;
                // found has only to be set if the binomial has not
                // been reduced to zero (else there are no new
                // binomials)
                aux_list._insert(bin);
                first_iter.extract_element();
              }
              else
                first_iter.delete_element();

              size--;
            }
            second_iter.next();
          }
        }
```

```
        // The list generators[i]=generators
        //  [S.subsets_of_support[i][S.number_of_subsets[i]-1]]
        // has to be tested separately to avoid a reduction of the
        // actual binomial by itself.

        if(changed==0)
          // binomial referenced by first_iter has not yet been
          // reduced
        {
          second_iter.set_to_list(generators[i]);

          while((second_iter.is_at_end()==FALSE) && (changed==0))
          {
            if(second_iter!=first_iter)
              // the two iterators do not reference the same element
              changed=bin.reduce_head_by(second_iter.get_element(),w);

            if(changed!=0)
              // bin has been reduced
            {

#ifdef SL_LIST

              // bin will be deleted ar extracted - maybe dangerous
              // for second_iter
              if(first_iter.next_is(second_iter))
                second_iter.next();

#endif  // SL_LIST

              if(bin!=0)
              {
                found=1;
                aux_list._insert(bin);
                first_iter.extract_element();
              }
              else
                first_iter.delete_element();
              size--;
            }
            else
              // bin has not been reduced
              second_iter.next();
          }
        }

      if(changed==0)
```

```
        first_iter.next();
      // Else first_iter has already been set to the next element by
      // deletion or extraction.
      }

//  Now try to reduce the binomials that are marked reduced.

      while(first_iter.is_at_end()==FALSE)
        // only unreduced elements have to be tested for
        // interreduction
      {
        binomial& bin=first_iter.get_element();
        Integer changed=0;
        // binomial referenced by bin not yet reduced

// Look for head reductions:
// Iterate over the lists that could contain reducers for the head of
// bin. The list containing bin is tested separately to avoid an
// interreduction of a binomial by itself respectively unnecessary
// checks for this when the two iterators reference different lists.

        for(short j=0;(j<S.number_of_subsets[i]-1)&&(changed==0);j++)
        {
          second_iter.set_to_list(generators
                                  [S.subsets_of_support[i][j]]);
          // This is the j-th list among the generator lists with
          // elements whose support is a subset of that of i.
          // The support of i is just the head support of bin
          // (restricted to the corresponding variables).

          while((second_iter.element_is_marked_head_reduced()==FALSE)
                && (changed==0))
          {
            changed=bin.reduce_head_by(second_iter.get_element(),w);

            if(changed!=0)
              // bin has been reduced
            {
              if(bin!=0)
              {
                found=1;
                // found has only to be set if the binomial has not
                // been reduced to zero (else there are no new
                // binomials)
                aux_list._insert(bin);
                first_iter.extract_element();
              }
```

```
              else
                first_iter.delete_element();

              size--;
            }
            second_iter.next();
          }
        }

        // The list generators[i]=generators
        //  [S.subsets_of_support[i][S.number_of_subsets[i]-1]]
        // has to be tested separately to avoid a reduction of the
        // actual binomial by itself.

        if(changed==0)
          // binomial referenced by first_iter has not yet been
          // reduced
        {
          second_iter.set_to_list(generators[i]);

          while((second_iter.element_is_marked_head_reduced()==FALSE)
                && (changed==0))
          {
            if(second_iter!=first_iter)
              // the two iterators do not reference the same element
              changed=bin.reduce_head_by(second_iter.get_element(),w);

            if(changed!=0)
              // bin has been reduced
            {

#ifdef SL_LIST

              // bin will be deleted ar extracted - maybe dangerous
              // for second_iter
              if(first_iter.next_is(second_iter))
                second_iter.next();

#endif  // SL_LIST

              if(bin!=0)
              {
                found=1;
                aux_list._insert(bin);
                first_iter.extract_element();
              }
              else
```

```
            first_iter.delete_element();
          size--;
        }
        else
          // bin has not been reduced
          second_iter.next();
      }
    }

  if(changed==0)
    first_iter.next();
  // Else first_iter has already been set to the next element by
  // deletion or extraction.
  }
}

// Now we have found all currently possible reductions.
// The elements remaining in the generator lists cannot be
// interreduced and are marked reduced.

for(int i=0;i<Number_of_Lists;i++)
{
  first_iter.set_to_list(generators[i]);
  if(first_iter.is_at_end()==FALSE)
    while(first_iter.element_is_marked_head_reduced()==FALSE)
    {
      first_iter.mark_element_head_reduced();
      first_iter.next();
    }
}

// Now reinsert reduced elements
// It seems to be quite unimportant for the performance if an
// element is completely reduced before reinsertion or not.

first_iter.set_to_list(aux_list);
while(first_iter.is_at_end()==FALSE)
{
  binomial& bin=first_iter.get_element();
  reduce(bin,FALSE);

  if(bin==0)
    first_iter.delete_element();
  else
  {
    generators[bin.head_support%Number_of_Lists].insert(bin);
    size++;
```

```
        // We do not call the add_generator(...)-routine because we do
        // not want number_of_new_binomials to be incremented.
        first_iter.extract_element();
      }
    }
  }
  while(found==1);
  // When leaving the loop, no generators have been interreduced during
  // the last iteration; we are done.

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  return(*this);
}

ideal& ideal::final_reduce()
{
// During Buchberger's algorithm, we perform only head reductions
// (minimalizations). This strategy showed to be a little more
// efficient than the strategy to do reduce the ideal always
// completely. This method leads to a minimal, but in general not to
// the reduced Groebner basis. The actual procedure reduces such a
// minimal basis at the end of Buchberger's algorithm. It will
// probably cause problems when called in the course of the
// algorithm. For an explaination of this fact, see the following
// comment.

  minimalize();

// Now there remain only tail reductions. They are quite simple: Each
// binomial's tail is reduced as long as possible. As no binomial can
// be reduced to zero by that and a binomial cannot reduce its own
// tail, we do not have to pay special attention to that (under the
// assumption that a real term ordering (i.e. a well-ordering) is used
// and that the head and the tail of the binomial are coorect with
// respect to this ordering).
// Notice that the head can change because of a tail reduction due to
// the trivial factors elimination (the new head will always divide
// the old one). This change is especially dangerous if
// SUPPORT_DRIVEN_METHODS_EXTENDED are enabled: It may happen that the
// binomial is in the wrong list after a tail reduction.
// Furthermore, if a head change occurs, it may happen that the
// generating set is no more minimalized after this. So the reduction
// has to be restarted after such a head change (and the respective
// binomial has to be marked head_unreduced before).
// This does not seem to be very efficient.
// For this reason, the reduction routine is only written for a final
```

```
// reduction (having already computed a Groebner basis of the ideal).
// This Groebner basis is first minimalized. After that, a head change
// during tail reduction is impossible because the head is already a
// minimal generator of the initial ideal (and the new head would
// divide the old).
// However, this argumentation is only valid if the input ideal is
// saturated. In some algorithms (Hosten_Sturmfels...) this is not the
// case in intermediate steps. The final reduction may cause
// inconsistencies here. But as the list structure is rebuild after
// each intermediate Groebner basis calculation (change of the term
// ordering) and as the last Groebner basis calculation deals with a
// saturated ideal, the final result will be correct.
// (For non-saturated input ideals, the computed Groebner basis is in
// general not a Groebner basis of the input ideal, but one for an
// ideal "between" the input ideal and its saturation.)

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  list_iterator first_iter;
  first_iter.set_to_list(generators);

  while(first_iter.is_at_end()==FALSE)
  {
    binomial& bin=first_iter.get_element();
    short changed;
    // to control if bin has been reduced

    do
    {
      changed=0;
      list_iterator second_iter;
      second_iter.set_to_list(generators);

      while(second_iter.is_at_end()==FALSE)
      {
        changed+=bin.reduce_tail_by(second_iter.get_element(),w);
        // As soon as a reduction occurs, changed is set to a value !=0.
        second_iter.next();
      }
    }
    while(changed>0);

    first_iter.next();
  }

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED
```

```
#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

// In this case, the reduction has to be organized in a slightly
// different way to use the support information to its full extend.
// As soon as an reduction has taken place, the iteration over the
// reducer lists is restarted using the new tail support information.

  list_iterator first_iter;

  for(short i=0;i<Number_of_Lists;i++)
  {
    first_iter.set_to_list(generators[i]);

    while(first_iter.is_at_end()==FALSE)
    {
      binomial& bin=first_iter.get_element();
      short changed;
      // to control if bin has been reduced

      do
      {
        changed=0;
        list_iterator second_iter;
        short supp=bin.tail_support%Number_of_Lists;
        // determine the lists over which we have to iterate

        for(short j=0;(j<S.number_of_subsets[supp])
                      && (changed==0);j++)
        {
          second_iter.set_to_list(generators
                                  [S.subsets_of_support[supp][j]]);
          // This is the j-th list among the generator lists with
          // elements whose support is a subset of supp.

          while((second_iter.is_at_end()==FALSE) && (changed==0))
          {
            changed=bin.reduce_tail_by(second_iter.get_element(),w);
            // Here we can do a simple assignment; as the iteration is
            // stopped as soon as some reduction is done, reduced
            // cannot be reset to zero in this assignment.
            second_iter.next();
          }
        }
      }
      while(changed>0);

      first_iter.next();
```

```
    }
  }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  return(*this);
}

////////////////////////////////////////////////////////////////////
///////////////// auxiliary stuff //////////////////////////////////
////////////////////////////////////////////////////////////////////

short ideal::add_new_generators()
{
// Reduces the binomials in the "new_generators" list(s) by the
// generators and moves them to the "generators" list(s).

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  short result=0;
  // element inserted?
  list_iterator iter(new_generators);

  while(iter.is_at_end()==FALSE)
  {
    binomial& bin=iter.get_element();
    reduce(bin,FALSE);

    if(bin==0)
      iter.delete_element();
    else
    {
      add_generator(bin);
      iter.extract_element();
      result=1;
    }
  }

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  short result=0;
  // element inserted?
  list_iterator iter;

  for(short i=0;i<Number_of_Lists;i++)
```

```
  {
    iter.set_to_list(new_generators[i]);
    while(iter.is_at_end()==FALSE)
    {
      binomial& bin=iter.get_element();
      reduce(bin,FALSE);

      if(bin==0)
        iter.delete_element();
      else
      {
        add_generator(bin);
        iter.extract_element();
        result=1;
      }
    }
  }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  return result;
}

//////////////////////////////////////////////////////////////////////
///////////////// binomial reduction ///////////////////////////////
//////////////////////////////////////////////////////////////////////

binomial& ideal::reduce(binomial& bin, BOOLEAN complete=TRUE) const
{
// As bin is reduced by a fixed set of binomials, it is sufficient to
// do head reductions first, then tail reductions (cf. Pottier).

  list_iterator iter;
  Integer reduced;
  // to control if the binomial has been reduced during the actual
  // iteration

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  do
  {
    iter.set_to_list(generators);
    reduced=0;
    // not yet reduced

    while(iter.is_at_end()==FALSE)
    {
```

```
      reduced+=bin.reduce_head_by(iter.get_element(),w);
      // reduced is incremented (and so set to a value >0) as soon as
      // bin is really reduced
      iter.next();
    }
  }
  while((reduced>0) && (bin!=0));

  if(complete==TRUE)
    do
    {
      iter.set_to_list(generators);
      reduced=0;
      // not yet reduced

      while(iter.is_at_end()==FALSE)
      {
        reduced+=bin.reduce_tail_by(iter.get_element(),w);
        // reduced is incremented (and so set to a value >0) as soon
        // as bin is really reduced
        iter.next();
      }
    }
    while((reduced>0) && (bin!=0));

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  do
  {
    reduced=0;
    // not yet reduced

    short supp=bin.head_support%Number_of_Lists;
    // determine the lists over which we have to iterate

    // As soon as some reduction is done, the iteration is started
    // with the new support information; so we do not finish
    // iterations over lists that cannot contain reducers any more.

    for(short i=0;(i<S.number_of_subsets[supp]) && (reduced==0);i++)
    {
      iter.set_to_list(generators[S.subsets_of_support[supp][i]]);
      // This is the i-th list among the generator lists with elements
      // whose support is a subset of that of supp.
```

```
      while((iter.is_at_end()==FALSE)&&(reduced==0))
      {
        reduced=bin.reduce_head_by(iter.get_element(),w);
        // Here we can do a simple assignment; as the iteration is
        // stopped as soon as some reduction is done, reduced cannot
        // be reset to zero in this assignment.
        iter.next();
      }
    }
  }
  while((reduced>0) && (bin!=0));

  if(complete==TRUE)
    do
    {
      reduced=0;
      // not yet reduced
      short supp=bin.tail_support%Number_of_Lists;
      // determine the lists over which we have to iterate

      // As soon as some reduction is done, the iteration is started
      // with the new support information; so we do not finish
      // iterations over lists that cannot contain reducers any more.

      for(short i=0;(i<S.number_of_subsets[supp]) && (reduced==0);i++)
      {
        iter.set_to_list(generators[S.subsets_of_support[supp][i]]);
        // This is the i-th list among the generator lists with
        // elements whose support is a subset of that of supp.

        while((iter.is_at_end()==FALSE)&&(reduced==0))
        {
          reduced=bin.reduce_tail_by(iter.get_element(),w);
          // Here we can do a simple assignment; as the iteration is
          // stopped as soon as some reduction is done, reduced cannot
          // be reset to zero in this assignment
        iter.next();
      }
    }
  }
  while(reduced>0);
  // bin cannot be reduced to zero by a tail reduction

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  return(bin);
}
```

```
//////////////////////////////////////////////////////////////////////
/////////// variants of Buchberger's algorithm /////////////////////
//////////////////////////////////////////////////////////////////////

ideal& ideal::reduced_Groebner_basis_1
              (const short& S_pair_criteria=11,
               const float& interred_percentage=12.0)
{
  // set flags for the use of the S-pair criteria
  // for an explaination see in globals.h
  rel_primeness=(S_pair_criteria & 1);
  M_criterion=(S_pair_criteria & 2);
  F_criterion=(S_pair_criteria & 4);
  B_criterion=(S_pair_criteria & 8);
  second_criterion=(S_pair_criteria & 16);

  interreduction_percentage=interred_percentage;

  short done;
  // control variable for recognizing when Buchberger's algorithm has
  // reached his end

  // first minimalize the initial generating system
  minimalize();

  do
  {
    done=1;
    // no new generators found yet

    compute_actual_S_pairs_1();
    // compute the S-pairs of the actual generators
    // These are stored in a unreduced form in aux_list.

    list_iterator S_pair_iter(aux_list);
    // now reduce and insert the computed S-pairs

    while(S_pair_iter.is_at_end()==FALSE)
    {
      binomial& S_bin=S_pair_iter.get_element();
      reduce(S_bin,FALSE);

      if(S_bin!=0)
        // new generator found
      {
        add_generator(S_bin);
```

```
        S_pair_iter.extract_element();
        done=0;
        // algorithm does not terminate when a new generator is found
      }
      else
        S_pair_iter.delete_element();
    }
    // now all computed S-pairs are inserted as generators
    // enough for a minimalization?

    if(interreduction_percentage>=0)
      // intermediate interreductions allowed
    {
      if(number_of_new_binomials>=size*interreduction_percentage/100)
        // enough new generators since last minimalization
      {
        minimalize();
        number_of_new_binomials=0;
      }
    }
    // now we restart the algorithm with the new generating system
    // if the generating system has changed during the last iteration
  }
  while(done==0);
  // if done==1, all computed S-pairs reduced to zero

  // compute reduced from minimal Groebner basis
  final_reduce();

  return(*this);
}

ideal& ideal::reduced_Groebner_basis_1a
              (const short& S_pair_criteria=11,
               const float& interred_percentage=12.0)
{

... // as ideal& ideal::reduced_Groebner_basis_1(...);
    // compute_actual_S_pairs_1(); replaced by compute_actual_S_pairs_1a();

}

ideal& ideal::reduced_Groebner_basis_2
      (const short& S_pair_criteria=11,
               const float& interred_percentage=12.0)
{
  // set flags for the use of the S-pair criteria
```

```
// for an explaination see in globals.h
rel_primeness=(S_pair_criteria & 1);
M_criterion=(S_pair_criteria & 2);
F_criterion=(S_pair_criteria & 4);
B_criterion=(S_pair_criteria & 8);
second_criterion=(S_pair_criteria & 16);

interreduction_percentage=interred_percentage;

short done;
// control variable for recognizing when Buchberger's algorithm has
// reached his end

// first minimalize the initial generating system
minimalize();

do
{
  done=1;
  // no new generators found yet

  compute_actual_S_pairs_2();
  // compute the S-pairs of the actual generators
  // These are stored in a reduced version in aux_list.

  minimalize_S_pairs();
  // minimalize the binomials in aux_list
  // These are not only S-pairs, but also ideal generators that were
  // reduced by an S-pair during the S-pair computation.

  list_iterator S_pair_iter(aux_list);
  if(S_pair_iter.is_at_end()==FALSE)
    // Zero binomials are not inserted in aux_list during the S-pair
    // computation; i.e. if aux_list is not empty, a further
    // iteration step has to be done.
    done=0;

  // now insert the computed S-pairs
  while(S_pair_iter.is_at_end()==FALSE)
    // S_pairs remaining
  {
    add_generator(S_pair_iter.get_element());
    S_pair_iter.extract_element();
  }
  // now all computed S-pairs are inserted as generators
  // enough for a minimalization?
```

```
    if(interreduction_percentage>=0)
      // intermediate interreductions allowed
    {
      if(number_of_new_binomials>=size*interreduction_percentage/100)
        // enough new generators since last minimalization
      {
        minimalize();
        number_of_new_binomials=0;
      }
    }
    // now we restart the algorithm with the new generating system
    // if the generating system has changed during the last iteration
  }
  while(done==0);
  // if done==1, all computed S-pairs reduced to zero

  // compute reduced from minimal Groebner basis
  final_reduce();

  return(*this);
}

ideal& ideal::reduced_Groebner_basis_3
              (const short& S_pair_criteria=11,
               const float& interred_percentage=12.0)
{
  // set flags for the use of the S-pair criteria
  // for an explaination see in globals.h
  rel_primeness=(S_pair_criteria & 1);
  M_criterion=(S_pair_criteria & 2);
  F_criterion=(S_pair_criteria & 4);
  B_criterion=(S_pair_criteria & 8);
  second_criterion=(S_pair_criteria & 16);

  interreduction_percentage=interred_percentage;

  short not_done;
  // control variable for recognizing when Buchberger's algorithm has
  // reached his end

  // first minimalize the initial generating system
  minimalize();

  do
  {
    compute_actual_S_pairs_3();
    // compute the S-pairs of the actual generators
```

```
    // These are stored in areduced version in the list(s)
    // new_generators.

    minimalize_new_generators();
    // minimalize the binomials in new_generators
    // These are not only S-pairs, but also ideal generators that were
    // reduced by an S-pair during the S-pair computation.

    not_done=add_new_generators();
    // move binomials from new_generators to generators

    // now all computed S-pairs are inserted as generators
    // enough for a minimalization?

    if(interreduction_percentage>=0)
      // intermediate interreductions allowed
    {
      if(number_of_new_binomials>=size*interreduction_percentage/100)
        // enough new generators since last reduction
      {
        minimalize();
        number_of_new_binomials=0;
      }
    }
    // now we restart the algorithm with the new generating system
    // if the generating system has changed during the last iteration
  }
  while(not_done==1);
  // if not_done==0, all computed S-pairs reduced to zero

  // compute reduced from minimal Groebner basis
  final_reduce();

  return(*this);
}

ideal& ideal::reduced_Groebner_basis(const short& version=1,
              const short& S_pair_criteria=11,
              const float& interred_percentage=12.0)
{
  switch(version)
  {
      case 0:
        return reduced_Groebner_basis_1a(S_pair_criteria,
         interred_percentage);
      case 1:
        return reduced_Groebner_basis_1(S_pair_criteria,
```

```
interred_percentage);
      case 2:
        return reduced_Groebner_basis_2(S_pair_criteria,
interred_percentage);
      case 3:
        return reduced_Groebner_basis_3(S_pair_criteria,
interred_percentage);
      default:
        cerr<<"WARNING: ideal& ideal::reduced_Groebner_basis"
          "(const short&, const short&, const float&):\n"
          "version argument out of range, nothing done"<<endl;
        return*this;
  }
}

#endif  // BUCHBERGER_CC
```

A.2.3 change_cost.cc

```
// change_cost.cc

// This file provides an interface for changing the cost vector in
// IP-problems and for recomputing Groebner basis of toric ideals.

#ifndef CHANGE_COST_CC
#define CHANGE_COST_CC

#include "IP_algorithms.h"

int main(int argc, char *argv[])
{
  // initialize arguments for the IP-algorithms (Buchberger)
  // with default settings
  BOOLEAN verbose=FALSE;
  short version=1;
  short S_pair_criteria=11;
  double interreduction_percentage=12.0;

//////////////////// parse options //////////////////////////////

  if(argc>=3)
    // argc has to be at least 3 (except when help is required, see below):
    // program name, GROEBNER file, NEW_COST file
  {

... // analogous to solve_IP.cc, but without option -alg
```

```
///////////////// call Buchberger's algorithm ////////////////////

    // open first input stream (GROEBNER file)
    ifstream first_input(argv[argc-2]);
    if(!first_input)
    {
      cerr<<"ERROR: cannot open first input file, possibly not "
        "found"<<endl;
      return 1;
    }

    // open second input stream (NEW_COST file)
    ifstream second_input(argv[argc-1]);
    if(!second_input)
    {
      cerr<<"ERROR: cannot open second input file, possibly not "
        "found"<<endl;
      return 1;
    }

    // change term ordering
    int success=change_cost(argv[argc-2], argv[argc-1], version,
                            S_pair_criteria,
                            interreduction_percentage,
                            verbose);
    if(!success)
    {
      cerr<<"ERROR: could not change cost vector,\n"
        "no new GROEBNER file created"<<endl;
      return 1;
    }

    // Groebner basis successfully changed
    return 0;
  }

////////////////// provide help text ////////////////////////////

  // handle the case of "missing" arguments
   if((argc==2) && !strcmp(argv[1],"--help"))
    // help required
  {
    system("less change_cost.hlp");
    return 0;
  }
  else
    // invalid arguments
```

```
    cerr<<"USAGE: change_cost [options] groebner_file "
      "new_cost_file"<<endl
        <<"Type ‘change_cost --help’ for help on options."<<endl;

  return 1;
}

#endif  // CHANGE_COST_CC
```

A.2.4 ideal.cc

```
// ideal.cc

// implementation of some general ideal functions

#ifndef IDEAL_CC
#define IDEAL_CC

#include "ideal.h"

/////////////////////////////////////////////////////////////////////
////////////// private member functions /////////////////////////////
/////////////////////////////////////////////////////////////////////

///////////// subset_tree data structure ////////////////////////////

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

ideal::create_subset_tree()
{
  for(short i=0;i<Number_of_Lists;i++)
  {

// First determine the number of binary vectors whose support is a
// subset of the support of i (i read as binary vector).
// The support of i is a set of cardinality s, where s is the number
// of bits in i that are 1. Hence the desired number is 2^s.

    short s=0;
    for(short k=0;k<List_Support_Variables;k++)
      if( (i&(1<<k)) == (1<<k) )
        // bit k of i is 1
        s++;
    S.number_of_subsets[i]=(1<<s);
    // (1<<s) == 2^s

// Now determine the concrete binary vectors whose support is a subset
```

```
// of that of i. This is done in a very simple manner by comparing
// the support of each number between 0 and i (read as binary vector)
// with that of i. (Efficiency considerations are absolutely
// unimportant in this function.)

    S.subsets_of_support[i]=new short[S.number_of_subsets[i]];
    // memory allocation for subsets_of_support[i]

    short index=0;
    for(short j=0;j<Number_of_Lists;j++)
      if((i&j)==j)
        // If the support of j as a bit vector is contained in the
        // support of i as a bit vector, j is saved in the list
        // subsets_of_support[i].
  {
    S.subsets_of_support[i][index]=j;
    index++;
  }
  }
}

ideal::destroy_subset_tree()
{
  for(short i=0;i<Number_of_Lists;i++)
    delete[] S.subsets_of_support[i];
  // The arrays number_of_subsets and subsets_of_support (the
  // (short*)-array) are not dynamically allocated and do not have to
  // be deleted.
}

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

/////////// subroutines for Buchberger's algorithm ////////////////

ideal& ideal::add_new_generator(binomial& bin)
{

... // analogous to ideal& ideal::add_generator(binomial& bin)
    // no increment of size and number_of_new_binomials
}

ideal& ideal::add_generator(binomial& bin)
{
// Beside its function as a auxiliary routine for a shorter code, this
// routine offers a good way to hide if
// SUPPORT_DRIVEN_METHODS_EXTENDED are used or not. So the
// constructors do not have to take care about this.
```

```
#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  generators[(bin.head_support)%Number_of_Lists].insert(bin);
  // insert bin according to its support,
  // considering only the first List_Support_Variables variables.

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  generators.insert(bin);

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  size++;
  number_of_new_binomials++;
  return(*this);
}

//////////////// constructor subroutines //////////////////////////

ideal& ideal::Conti_Traverso_ideal(matrix& A, const term_ordering& _w)
{
  // A may have negative entries; to model this with binomials, we
  // need an inversion variable.

  w=_w;
  // The argument term ordering should be given by the objective
  // function.
  w.convert_to_elimination_ordering(A.rows+1,LEX);
  // extend term ordering into an elimination ordering of the
  // appropriate size

  Integer generator[A.columns+A.rows+1];
  // A.columns + A.rows + 1 is the number of variables for the
  // Conti-Traverso algorithm with "inversion variable".

  // build initial ideal generators
  for(short j=0;j<A.columns;j++)
  {
    for(short k=0;k<A.columns;k++)
      // original variables
      if(j==k)
        generator[k]=-1;
      else
        generator[k]=0;
```

```
    for(short i=0;i<A.rows;i++)
      // elimination variables
      generator[A.columns+i]=A.coefficients[i][j];

    generator[A.columns+A.rows]=0;
    // inversion variable

    // Note that the relative order of the variables is important:
    // If the elimination variables do not follow the other variables,
    // the conversion of the term ordering has not the desired effect.

    binomial* bin=new binomial(A.rows+1+A.columns,generator,w);
    add_generator(*bin);
  }

  // now add the "inversion generator"
  for(short j=0;j<A.columns;j++)
    generator[j]=0;
  for(short i=0;i<A.rows+1;i++)
    generator[A.columns+i]=1;
  binomial* bin=new binomial(A.rows+1+A.columns,generator,w);
  add_generator(*bin);

  return *this;
}

ideal& ideal::Positive_Conti_Traverso_ideal(matrix& A,
                                            const term_ordering& _w)
{
  // A is assumed to have only nonnegative entries; then we need no
  // "inversion variable".

  w=_w;
  // The argument term ordering should be given by the objective
  // function.
  w.convert_to_elimination_ordering(A.rows, LEX);
  // extend term ordering into an elimination ordering of the
  // appropriate size

  Integer generator[A.columns+A.rows];
  // A.columns + A.rows is the number of variables for the
  // Conti-Traverso algorithm without "inversion variable".

  // build the initial ideal generators
  for(short j=0;j<A.columns;j++)
  {
```

```
    for(short k=0;k<A.columns;k++)
      // original variables
      if(j==k)
        generator[k]=-1;
      else
        generator[k]=0;

    for(short i=0;i<A.rows;i++)
      // elimination variables
      generator[A.columns+i]=A.coefficients[i][j];

    // Note that the relative order of the variables is important:
    // If the elimination variables do not follow the other variables,
    // the conversion of the term ordering has not the desired effect.

    binomial* bin=new binomial(A.rows+A.columns,generator,w);
    add_generator(*bin);
  }
  return *this;
}

ideal& ideal::Pottier_ideal(matrix& A, const term_ordering& _w)
{
  w=_w;
  // The argument term_ordering should be given by the objective
  // function.
  w.convert_to_elimination_ordering(1,LEX);
  // add one elimination variable used to saturate the ideal

  if(A._kernel_dimension==-2)
    // kernel of A not yet computed, do this now
    A.LLL_kernel_basis();
  if((A._kernel_dimension==-1) &&  (A.columns<0))
    // error occurred in kernel computation or matrix corrupt
  {
    cout<<"\nWARNING: ideal& ideal::Pottier_ideal(matrix&, const "
      "term_ordering&):\ncannot build ideal from a corrupt input "
      "matrix"<<endl;
    size=-1;
    return *this;
  }

  Integer generator[A.columns+1];
  // This is the number of variables needed for Pottier's algorithm.

  // compute initial generating system from the kernel of A
  for(short j=0;j<A._kernel_dimension;j++)
```

```
  {
    for(short k=0;k<A.columns;k++)
    {
      // We should first verifie if the components of the LLL-reduced
      // lattice basis fit into the basic data type (Integer as
      // defined in globals.h). This overflow control does of course
      // not detect overflows in the course of the LLL-algorithm!

#ifdef _SHORT_

      if(((A.H)[j][k]>SHRT_MAX) || ((A.H)[j][k]<SHRT_MIN))
      {
        cerr<<"\nWARNING: ideal& ideal::Pottier_ideal(matrix&, const "
          "term_ordering&):\n"
          "LLL-reduced kernel basis does not fit into the used "
          "basic data type short."<<endl;
        size=-3;
        return *this;
      }

#endif // _SHORT_

#ifdef _INT_

      if(((A.H)[j][k]>INT_MAX) || ((A.H)[j][k]<INT_MIN))
      {
        cerr<<"\nWARNING: ideal& ideal::Pottier_ideal(matrix&, const "
          "term_ordering&):\n"
          "LLL-reduced kernel basis does not fit into the used "
          "basic data type int."<<endl;
        size=-3;
        return *this;
      }

#endif  // _INT_

#ifdef _LONG_

      if(((A.H)[j][k]>LONG_MAX) || ((A.H)[j][k]<LONG_MIN))
      {
        cerr<<"\nWARNING: ideal& ideal::Pottier_ideal(matrix&, const "
          "term_ordering&):\n"
          "LLL-reduced kernel basis does not fit into the used "
          "basic data type long."<<endl;
        size=-3;
        return *this;
      }
```

```
#endif  // _LONG_

#ifdef LEDA

      generator[k]=(Integer) Double((A.H)[j][k]);
      // convert BigInt type into the basic data type

#else

      generator[k]=(A.H)[j][k];

#endif  // LEDA

    }
    generator[A.columns]=0;
    // elimination variable

    // Note that the relative order of the variables is important:
    // If the elimination variable does not follow the other variables,
    // the conversion of the term ordering has not the desired effect.

    binomial* bin=new binomial(A.columns+1,generator,w);
    add_generator(*bin);
  }

  // build "saturation generator"
  // The use of the hosten_shapiro procedure is useful here because
  // the head of the computed saturation generator is smaller if less
  // variables are involved.

  short* sat_var;
  short number_of_sat_var=A.hosten_shapiro(sat_var);

  for(short j=0;j<A.columns;j++)
    generator[j]=0;
  for(short k=0;k<number_of_sat_var;k++)
    generator[sat_var[k]]=1;
  generator[A.columns]=1;

  binomial* bin=new binomial(A.columns+1,generator,w);
  add_generator(*bin);
  // The "saturation generator" seems to be a monomial, but is interpreted
  // as a binomial with tail 1 by the designed data structures.
  delete[] sat_var;

  return *this;
```

```
}

ideal& ideal::Hosten_Sturmfels_ideal(matrix& A,
                                     const term_ordering& _w)
{
  // check term ordering
  if((_w.weight_refinement()!=W_REV_LEX) && (_w.is_positive()==FALSE))
    cerr<<"\nWARNING: ideal& ideal::Hosten_Sturmfels_ideal(matrix&, "
      "const term_ordering&):\nargument term ordering should be a "
      "weighted reverse lexicographical one \nwith positive weights"
        <<endl;
  w=_w;
  // The argument term_ordering should be given by a homogenous grading.

  if(A._kernel_dimension==-2)
    // kernel of A not yet computed, do this now
    A.LLL_kernel_basis();
  if((A._kernel_dimension==-1) &&  (A.columns<0))
    // error occurred in kernel computation or matrix corrupt
  {
    cout<<"\nWARNING: ideal& ideal::Hosten_Sturmfels_ideal(matrix&, "
      "const term_ordering&):\ncannot build ideal from a corrupt input "
      "matrix"<<endl;
    size=-1;
    return *this;
  }

  Integer generator[A.columns];
  // The algorithm of Hosten and Sturmfels does not need supplementary
  // variables.

  // compute initial generating system from the kernel of A
  for(short j=0;j<A._kernel_dimension;j++)
  {
    for(short k=0;k<A.columns;k++)
    {

... // overflow control as in Pottier_ideal(...)

#ifdef LEDA

      generator[k]=(Integer) Double((A.H)[j][k]);
      // convert BigInt type into the basic data type

#else

      generator[k]=(A.H)[j][k];
```

```
#endif  // LEDA
    }

    // verifie term ordering
    if(w.weight(generator)!=0)
      cerr<<"\nWARNING: ideal& ideal::Hosten_Sturmfels_ideal(matrix&, "
        "const term_ordering&):\nInvalid row space vector does not "
        "induce homogeneous grading."<<endl;

    binomial* bin=new binomial(A.columns,generator,w);
    add_generator(*bin);
  }
  return *this;
}

ideal& ideal::DiBiase_Urbanke_ideal(matrix& A,
                                    const term_ordering& _w)
{
  w=_w;

  if(A._kernel_dimension==-2)
    // kernel of A not yet computed, do this now
    A.LLL_kernel_basis();
  if((A._kernel_dimension==-1) &&  (A.columns<0))
    // error occurred in kernel computation or matrix corrupt
  {
    cout<<"\nWARNING: ideal& ideal::DiBiase_Urbanke_ideal"
      "(matrix&, const term_ordering&):\n"
      "cannot build ideal from a corrupt input matrix"<<endl;
    size=-1;
    return *this;
  }

  // now compute flip variables
  short* F;
  // set of flip variables
  // If F[i]==j, x_j will be flipped.
  short r=A.compute_flip_variables(F);
  // number of flip variables

  if(r<0)
  {
    cout<<"Kernel of the input matrix contains no vector with nonzero "
      "components.\nPlease use another algorithm."<<endl;
    size=-1;
    return *this;
```

```
  }

  // check term ordering (as far as possible)
  BOOLEAN ordering_okay=TRUE;

  if(r>0)
  {
    if(_w.weight_refinement()!=W_LEX)
      ordering_okay=FALSE;
    for(short i=0;i<_w.number_of_weighted_variables();i++)
      if((_w[i]!=0) && (i!=F[0]))
        ordering_okay=FALSE;
  }

  if(ordering_okay==FALSE)
    cerr<<"\nWARNING: ideal& ideal::DiBiase_Urbanke_ideal(matrix&, "
      "const term_ordering&):\n"
      "argument term ordering might be inappropriate"<<endl;

  Integer generator[A.columns];
  // The algorithm of DiBiase and Urbanke does not need supplementary
  // variables.

  // compute initial generating system from the kernel of A
  for(short j=0;j<A._kernel_dimension;j++)
  {
    for(short k=0;k<A.columns;k++)
    {

... // overflow control as in Pottier_ideal(...)

#ifdef LEDA

      generator[k]=(Integer) Double((A.H)[j][k]);
      // convert BigInt type into the basic data type

#else

      generator[k]=(A.H)[j][k];

#endif  // LEDA
    }

    // flip variables
    for(short l=0;l<r;l++)
      generator[F[l]]*=-1;
```

```
    binomial* bin=new binomial(A.columns,generator,w);
    add_generator(*bin);
  }
  delete[] F;
  return *this;
}

ideal& ideal::Bigatti_LaScala_Robbiano_ideal(matrix& A,
                                             const term_ordering& _w)
{
  // check term ordering
  if((_w.weight_refinement()!=W_REV_LEX) && (_w.is_positive()==FALSE))
    cerr<<"\nWARNING: ideal& ideal::Bigatti_LaScala_Robbiano_ideal"
      "(matrix&, const term_ordering&):\n"
      "argument term ordering should be a weighted  "
      "reverse lexicographical one \nwith positive weights"<<endl;
  w=_w;
  // The argument term_ordering should be given by a homogeneous
  // grading.

  if(A._kernel_dimension==-2)
    // kernel of A not yet computed, do this now
    A.LLL_kernel_basis();
  if((A._kernel_dimension==-1) &&  (A.columns<0))
    // error occurred in kernel computation or matrix corrupt
  {
    cout<<"\nWARNING: ideal& ideal::Bigatti_LaScala_Robbiano_ideal"
      "(matrix&, const term_ordering&):\n"
      "cannot build ideal from a corrupt input matrix"<<endl;
    size=-1;
    return *this;
  }

  // now compute saturation variables
  // The techniques for computing a small set of saturation variables
  // are useful here for the following two reasons:
  // - The head of the saturation generator involves less variables,
  //   is smaller in term ordering.
  // - The weight of the pseudo-elimination variable is smaller.

  short* sat_var;
  short number_of_sat_var=A.hosten_shapiro(sat_var);

  float weight=0;
  for(short i=0;i<number_of_sat_var;i++)
    weight+=w[sat_var[i]];
  w.append_weighted_variable(weight);
```

```
  // one supplementary variable used to saturate the ideal

  Integer generator[A.columns+1];
  // The algorithm of Bigatti, La Scala and Robbiano needs one
  // supplementary weighted variable.
  // first build "saturation generator"
  for(short k=0;k<A.columns;k++)
    generator[k]=0;
  for(short i=0;i<number_of_sat_var;i++)
    generator[sat_var[i]]=1;
  generator[A.columns]=-1;

  delete[] sat_var;

  binomial* bin=new binomial(A.columns+1,generator,w);
  add_generator(*bin);

  // compute initial generating system from the kernel of A
  for(short j=0;j<A._kernel_dimension;j++)
  {
    for(short k=0;k<A.columns;k++)
    {

... // overflow control as in Pottier_ideal(...)

#ifdef LEDA

      generator[k]=(Integer) Double((A.H)[j][k]);
      // convert BigInt type into the basic data type

#else

      generator[k]=(A.H)[j][k];

#endif  // LEDA
    }

    generator[A.columns]=0;
    // saturation variable
    // Note that the relative order of the variables is important
    // (because of the reverse lexicographical refinement of the
    // weight).

    if(w.weight(generator)!=0)
      cerr<<"\nWARNING: ideal& ideal::Bigatti_LaScala_Robbiano_ideal"
        "(matrix&, const term_ordering&):\n"
        "Invalid row space vector does not induce "
```

```
        "homogeneous grading."<<endl;

    binomial* bin=new binomial(A.columns+1,generator,w);
    add_generator(*bin);
    // insert generator
  }
  return *this;
}

////////////////////////////////////////////////////////////////////
////////////// public member functions /////////////////////////////
////////////////////////////////////////////////////////////////////

///////////////// constructors and destructor //////////////////////

ideal::ideal(matrix& A, const term_ordering& _w,
             const short& algorithm)
{
  // check arguments as far as possible
  if(A.error_status()<0)
  {
    cerr<<"\nWARNING: ideal::ideal(matrix&, const term_ordering&, "
      "const short&):\n"
      "cannot create ideal from a corrupt input matrix"<<endl;
    size=-1;
    return;
  }
  if(_w.error_status()<0)
  {
    cerr<<"\nWARNING: ideal::ideal(matrix&, const term_ordering&, "
      "const short&):\n"
      "cannot create ideal with a corrupt input ordering"<<endl;
    size=-1;
    return;
  }
  if((_w.number_of_elimination_variables()!=0) &&
     (_w.number_of_weighted_variables()!=A.columns))
    cerr<<"\nWARNING: ideal& ideal::ideal(matrix&, "
      "const term_ordering&):\n"
      "argument term ordering might be inappropriate"<<endl;

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

  create_subset_tree();

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED
```

```
  size=0;

  // initialize the S-pair flags with the default values
  // (this is not really necessary, but looks nicer when outputting
  // the ideal without having computed a Groebner basis)
  rel_primeness=1;
  M_criterion=2;
  F_criterion=0;
  B_criterion=8;
  second_criterion=0;

  interreduction_percentage=12.0;

  // construct the ideal according to the algorithm
  switch(algorithm)
  {
      case CONTI_TRAVERSO:
        Conti_Traverso_ideal(A,_w);
        break;
      case POSITIVE_CONTI_TRAVERSO:
        Positive_Conti_Traverso_ideal(A,_w);
        break;
      case POTTIER:
        Pottier_ideal(A,_w);
        break;
      case HOSTEN_STURMFELS:
        Hosten_Sturmfels_ideal(A,_w);
        break;
      case DIBIASE_URBANKE:
        DiBiase_Urbanke_ideal(A,_w);
        break;
      case BIGATTI_LASCALA_ROBBIANO:
        Bigatti_LaScala_Robbiano_ideal(A,_w);
        break;
      default:
        cerr<<"\nWARNING: ideal::ideal(matrix&, const term_ordering&, "
          "const short&):\nunknown algorithm for ideal construction"
          <<endl;
        size=-1;
        return;
  }
  number_of_new_binomials=size;
}

... // ideal::ideal(const ideal& I)
    // To be sure to get a real copy of the argument ideal, the lists
    // aux_list and new_generators are also copied.
```

```
... // ideal::ideal(ifstream& input, const term_ordering& _w,
    //              const short& number_of_generators)

... // ideal::~ideal()

///////////////// object information ////////////////////////////////

... // long ideal::number_of_generators() const
... // short ideal::error_status() const

//////////////////////// output /////////////////////////////////////

... // output routines

#endif  // IDEAL_CC
```

A.2.5 ideal_stuff.cc

```
// ideal_stuff.cc

// implementation of the special ideal features needed by the
// IP-algorithms

#ifndef IDEAL_STUFF_CC
#define IDEAL_STUFF_CC

#include "ideal.h"

////////////////// elimination stuff /////////////////////////////

ideal& ideal::eliminate()
{
// eliminates the generators of the ideal involving elimination
// variables with respect to w

  if(w.number_of_elimination_variables()<=0)
    // elimination unnecessary
    return *this;

  list_iterator iter;

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

// Simply iterate over the generator list and delete the elements
// involving elimination variables.
// There is no need to change the done/undone or the reduced/unreduced
```

```
// mark of an element.

  iter.set_to_list(generators);
  while((iter.is_at_end())==FALSE)
  {
    binomial& bin=iter.get_element();
    if(bin.involves_elimination_variables(w)==TRUE)
    {
      iter.delete_element();
      size--;
    }
    else
    {
      bin.drop_elimination_variables(w);
      iter.next();
    }
  }

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

// Iterate over the generator lists and check whether the elements
// involve elimination variables.
// As the set of support variables may be changed by the elimination,
// the elements that are not deleted are first moved to the aux_list
// and then reinserted according to their new support.
// The elimination variables are dropped while reinserting.
// In general, elimination is done only once. The time needed for this
// is linear in the number of generators (in the Groebner basis). The
// elimination itself is therefore very fast in comparison to the
// Groebner basis computation needed for it... So we renounce to a
// complicated optimization of this procedure (the support information
// is not used). In fact, tests show that elimination time is really
// negligible.

  // elimination
  for(short i=0;i<Number_of_Lists;i++)
  {
    iter.set_to_list(generators[i]);
    while((iter.is_at_end())==FALSE)
    {
      binomial& bin=iter.get_element();
      if(bin.involves_elimination_variables(w)==TRUE)
      {
        iter.delete_element();
        size--;
```

```
      }
      else
      {
        aux_list._insert(bin);
        iter.extract_element();
        // As the generators are reinserted later, we do not decrement the
        // size (and so do not need to increment it during reinsertion).
      }
    }
  }

  // reinsertion
  iter.set_to_list(aux_list);
  while(iter.is_at_end()==FALSE)
  {
    binomial& bin=iter.get_element();
    bin.drop_elimination_variables(w);
    generators[bin.head_support%Number_of_Lists].insert(bin);
    // size is not incremented, see above...
    iter.extract_element();
  }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  // finally adapt term ordering
  w.convert_to_weighted_ordering();
  return *this;
}

ideal& ideal::pseudo_eliminate()
{
  if(w.number_of_weighted_variables()<=0)
  {
    cerr<<"WARNING: ideal& ideal::pseudo_eliminate()\n"
      "cannot be performed without weighted variables"<<endl;
    return *this;
  }

  list_iterator iter;
  short last_weighted_variable=w.number_of_weighted_variables()-1;

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

// Simply iterate over the generator list and delete the elements
// involving the last weighted variable.
// There is no need to change the done/undone or the reduced/unreduced
// mark of an element.
```

```
  iter.set_to_list(generators);
  while(iter.is_at_end()==FALSE)
  {
    binomial& bin=iter.get_element();
    if(bin[last_weighted_variable]!=0)
      // weighted variable to drop is involved in bin
    {
      iter.delete_element();
      size--;
    }
    else
    {
      bin.drop_last_weighted_variable(w);
      iter.next();
    }
  }

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

// Iterate over the generator lists and check whether the elements
// involve the last weighted variable.
// As the set of support variables may be changed by the
// pseudo-elimination, the elements that are not deleted are first
// moved to the aux_list and then reinserted according to their new
// support.
// The last weight variable is dropped while reinserting.
// For the time needed by this function see the remarks for
// ideal::eliminate().

  for(short i=0;i<Number_of_Lists;i++)
  {
    iter.set_to_list(generators[i]);
    while((iter.is_at_end())==FALSE)
    {
      binomial& bin=iter.get_element();
      if(bin[last_weighted_variable]!=0)
      {
        iter.delete_element();
        size--;
      }
      else
      {
        aux_list._insert(bin);
        iter.extract_element();
```

```
        // As the generators are reinserted later, we do not decrement
        // the size (and so do not need to increment it during
        // reinsertion).
      }
    }
  }

  iter.set_to_list(aux_list);
  while(iter.is_at_end()==FALSE)
  {
    binomial& bin=iter.get_element();
    bin.drop_last_weighted_variable(w);
    generators[bin.head_support%Number_of_Lists].insert(bin);
    // size is not incremented, see above...
    iter.extract_element();
  }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  // finally adapt term ordering
  w.delete_last_weighted_variable();
  return *this;
}

/////////////// management of the term ordering ////////////////////

ideal& ideal::change_term_ordering_to(const term_ordering& _w)
{
  // first check compatibility
  if((w.number_of_weighted_variables()
      +w.number_of_elimination_variables())
     !=(_w.number_of_weighted_variables()
      +_w.number_of_elimination_variables()))
  {
    cerr<<"WARNING: ideal& ideal::change_term_ordering_to"
      "(const term_ordering&):\n"
      "incompatibility detected, term ordering not changed."<<endl;
    return *this;
  }
  // change term ordering
  w=_w;

  list_iterator iter;

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

// Simply iterate over the generator list. Because the change of the
```

```
// term ordering, the "done" and "reduced" marks of the elements have
// to be deleted.

  iter.set_to_list(generators);
  while((iter.is_at_end())==FALSE)
  {
    (iter.get_element()).adapt_to_term_ordering(w);
    iter.mark_element_undone();
    iter.mark_element_head_unreduced();
    iter.next();
  }

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

// As head and tail might have to be exchanged, the elements are first
// moved to the aux_list and then reinserted according to their new
// head.

  for(short i=0;i<Number_of_Lists;i++)
  {
    iter.set_to_list(generators[i]);
    while((iter.is_at_end())==FALSE)
    {
      binomial& bin=iter.get_element();
      if(bin.adapt_to_term_ordering(w)==-1)
        // head and tail exchanged
      {
        aux_list._insert(bin);
        iter.extract_element();
        // As the generators are reinserted later, we do not decrement
        // the size (and so do not need to increment it during
        // reinsertion).
      }
      else
      {
        // Although the S-pairs of the remaining elements have already
        // been computed once, the "done" marks have to be deleted:
        // With a new term ordering, the results of the S-pair
        // reduction can change - as well as the interreduction results.
        iter.mark_element_undone();
        iter.mark_element_head_unreduced();
        iter.next();
      }
    }
  }
```

```
  // reinsertion
  iter.set_to_list(aux_list);
  while(iter.is_at_end()==FALSE)
  {
    binomial& bin=iter.get_element();
    generators[bin.head_support%Number_of_Lists].insert(bin);
    // size is not incremented, see above...
    iter.extract_element();
  }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  return *this;
}

/////////// manipulation of the variables /////////////////////////

ideal& ideal::swap_variables_unsafe(const short& i, const short& j)
{
  // first check arguments
  if((i<0) || (i>=w.number_of_weighted_variables())
     || (j<0) || (j>=w.number_of_weighted_variables()))
  {
    cout<<"WARNING: ideal::swap_variables(const short&, "
      "const short&)\n or ideal::swap_variables_unsafe"
      "(const short&, const short&):\n"
      "index out of range"<<endl;
    return *this;
  }

  if(i==j)
    return(*this);
  // special case to avoid unnecessary overhead

  list_iterator iter;

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  iter.set_to_list(generators);
  while((iter.is_at_end())==FALSE)
  {
    (iter.get_element()).swap_variables(i,j);
    iter.next();
  }

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED
```

```
#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

// As head_support and tail_support are manipulated, the elements are
// first moved to the aux_list and then reinserted according to their
// new head. But head and tail are not adapted to the new term
// ordering induced by the change of the variable order - this is only
// done in the "safe" routine swap_variables(const short&, const
// short&).

  for(short l=0;l<Number_of_Lists;l++)
  {
    iter.set_to_list(generators[l]);
     while((iter.is_at_end())==FALSE)
    {
      binomial& bin=iter.get_element();
      bin.swap_variables(i,j);
      aux_list._insert(bin);
      iter.extract_element();
      // As the generators are reinserted later, we do not decrement the
      // size (and so do not need to increment it during reinsertion).
    }
  }
  // reinsertion
  iter.set_to_list(aux_list);
  while(iter.is_at_end()==FALSE)
  {
    binomial& bin=iter.get_element();
    generators[bin.head_support%Number_of_Lists].insert(bin);
    // size is not incremented, see above...
    iter.extract_element();
  }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  // finally adapt term ordering
  w.swap_weights(i,j);
  return *this;
}

ideal& ideal::swap_variables(const short& i, const short& j)
{
  swap_variables_unsafe(i,j);
  change_term_ordering_to(w);
  // This rebuilds the list structure...
  return *this;
}
```

```
ideal& ideal::flip_variable_unsafe(const short& i)
{
  // first check argument
  if((i<0) || (i>=w.number_of_weighted_variables()))
  {
    cout<<"WARNING: ideal::flip_variables(const short&):\n"
      "argument out of range, nothing done"<<endl;
    return *this;
  }

  list_iterator iter;

#ifdef NO_SUPPORT_DRIVEN_METHODS_EXTENDED

  iter.set_to_list(generators);
  while((iter.is_at_end())==FALSE)
  {
    (iter.get_element()).flip_variable(i);
    iter.next();
  }

#endif  // NO_SUPPORT_DRIVEN_METHODS_EXTENDED

#ifdef SUPPORT_DRIVEN_METHODS_EXTENDED

// As head_support and tail_support can change, the elements are first
// moved to the aux_list and then reinserted according to their new
// head.

  for(short l=0;l<Number_of_Lists;l++)
  {
    iter.set_to_list(generators[l]);
    while((iter.is_at_end())==FALSE)
    {
      binomial& bin=iter.get_element();
      bin.flip_variable(i);
      aux_list._insert(bin);
      iter.extract_element();
      // As the generators are reinserted later, we do not decrement the
      // size (and so do not need to increment it during reinsertion).
    }
  }

  // reinsertion
  iter.set_to_list(aux_list);
  while(iter.is_at_end()==FALSE)
  {
```

```
    binomial& bin=iter.get_element();
    generators[bin.head_support%Number_of_Lists].insert(bin);
    iter.extract_element();
    // size is not incremented, see above...
  }

#endif  // SUPPORT_DRIVEN_METHODS_EXTENDED

  return *this;
}

#endif  // IDEAL_STUFF_CC
```

A.2.6 IP_algorithms.cc

```
// IP_algorithms.cc

#ifndef IP_ALGORITMS_CC
#define IP_ALGORITHMS_CC

#include "IP_algorithms.h"
#include <time.h>
#include <unistd.h>

/////////// method for printing compilation settings ///////////////

... // void print_flags(ofstream& output)

////////////////////////////////////////////////////////////////////
///////////////// IP algorithms ////////////////////////////////////
////////////////////////////////////////////////////////////////////

int Conti_Traverso(INPUT_FILE MATRIX,
                   const short& version=1,
                   const short& S_pair_criteria=11,
                   const float& interred_percentage=12.0,
                   const BOOLEAN& verbose=FALSE)
{
///////////////////////// input ////////////////////////////////////

  char format_string[128]; // to verify file format
  short constraints;       // number of equality constraints
  short variables;         // number of variables (without auxiliary
                           // variables)
  ifstream input(MATRIX);

  // verify existence of file
```

```
if(!input)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "cannot read from input file, possibly not found"<<endl;
  return 0;
}

// read format specification
input>>format_string;
if(!input)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input failure when reading format specification,\n"
    "input format not accepted"<<endl;
  return 0;
}
if(strcmp(format_string,"MATRIX"))
  cerr<<"Warning: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input file has suspicious format"<<endl;

// read number of variables
input>>format_string;
if(!input)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input failure before reading number of variables "
    "/ matrix columns,\ninput format not accepted"<<endl;
  return 0;
}
if(strcmp(format_string,"columns:"))
  cerr<<"WARNING: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input file has suspicious format"<<endl;
input>>variables;
if(!input)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input failure when reading number of variables "
    "/ matrix columns,\ninput format not accepted"<<endl;
  return 0;
}
if(variables<=0)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "number of variables / matrix columns must be positve"<<endl;
  return 0;
}
```

```
// read term ordering
input>>format_string;
if(!input)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input failure before reading cost vector,\n"
    "input format not accepted"<<endl;
  return 0;
}
if(strcmp(format_string,"cost"))
  cerr<<"WARNING: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input file has suspicious format"<<endl;
input>>format_string;
if(!input)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input failure before reading cost vector,\n"
    "input format not accepted"<<endl;
  return 0;
}
if(strcmp(format_string,"vector:"))
  cerr<<"WARNING: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input file has suspicious format"<<endl;
term_ordering w(variables,input,W_LEX);
if(w.error_status()<0)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input failure when reading cost vector, input format not "
    "accepted"<<endl;
  return 0;
}
if(w.is_nonnegative()==FALSE)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "cost vectors with negative components are not supported"<<endl;
  return 0;
}

// read number of constraints
input>>format_string;
if(!input)
{
  cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
    "input failure before reading number of constraints "
    "/ matrix rows,\ninput format not accepted"<<endl;
  return 0;
}
```

```
  if(strcmp(format_string,"rows:"))
    cerr<<"WARNING: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
      "input file has suspicious format"<<endl;
  input>>constraints;
  if(!input)
  {
    cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
      "input failure when reading number of constraints "
      "/ matrix rows,\ninput format not accepted"<<endl;
    return 0;
  }
  if(constraints<=0)
  {
    cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
      "number of constraints / matrix rows must be positve"<<endl;
    // Solving problems without constraints is possible, but not very
    // interesting (because trivial). To avoid the problems and the
    // overhead incurred by an "empty" matrix, such problems are
    // refused.
    return 0;
  }

  // read matrix
  input>>format_string;
  if(!input)
  {
    cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
      "input failure before reading matrix,\n"
      "input format not accepted"<<endl;
    return 0;
  }
  if(strcmp(format_string,"matrix:"))
    cerr<<"WARNING: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
      "input file has suspicious format"<<endl;
  matrix A(constraints,variables,input);
  if(A.error_status()<0)
  {
    cerr<<"ERROR: int Conti_Traverso(INPUT_FILE, const BOOLEAN&):\n"
      "input failure when reading matrix, input format not accepted"
      <<endl;
    return 0;
  }

////////////////////// computation ///////////////////////////////

  // prepare time measurement
  clock_t start, end;
```

```
  // create toric ideal
  ideal I(A,w,CONTI_TRAVERSO);

  // compute the standard basis
  start=clock();
  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

////////////////////// output ///////////////////////////////////

  // create output file
  char GROEBNER[128];
  short i=0;
  while(MATRIX[i]!='\0' && MATRIX[i]!='.')
  {
    GROEBNER[i]=MATRIX[i];
    i++;
  }
  GROEBNER[i]='\0';
  strcat(GROEBNER,".GB.ct");
  ofstream output(GROEBNER);

  // format output file
  output.flags(output.flags()|ios::fixed);
  // output of fixed point numbers
  output<<"GROEBNER"<<endl<<endl;
  output<<"computed with algorithm:"<<endl;
  output<<"ct"<<endl;
  output<<"from file(s):"<<endl;
  output<<MATRIX<<endl;
  output<<"computation time"<<endl;
  output<<setw(8)<<setprecision(2)<<elapsed<<" sec"<<endl<<endl;
  output<<"term ordering:"<<endl;
  output<<"elimination block"<<endl;
  // elimination variables (>0)
  output<<constraints+1<<endl;
  output<<"LEX"<<endl;
  output<<"weighted block"<<endl;
  // weighted variables (>0)
  output<<variables<<endl;
  output<<"W_LEX"<<endl;
  w.format_print_weight_vector(output);
```

```
  output<<"size:"<<endl;
  output<<I.number_of_generators()<<endl<<endl;
  output<<"Groebner basis:"<<endl;
  I.format_print(output);
  output<<endl;

  if(verbose==TRUE)
  {
    output<<"settings for the Buchberger algorithm:"<<endl;
    output<<"version:"<<endl;
    if(version==0)
      output<<"1a"<<endl;
    else
      output<<version<<endl;
    output<<"S-pair criteria:"<<endl;
    if(S_pair_criteria & REL_PRIMENESS)
      output<<"relatively prime leading terms"<<endl;
    if(S_pair_criteria & M_CRITERION)
      output<<"criterion M"<<endl;
    if(S_pair_criteria & F_CRITERION)
      output<<"criterion F"<<endl;
    if(S_pair_criteria & B_CRITERION)
      output<<"criterion B"<<endl;
    if(S_pair_criteria & SECOND_CRITERION)
      output<<"second criterion"<<endl;
    output<<"interreduction frequency:"<<endl;
    output<<interred_percentage<<" %"<<endl<<endl;
    print_flags(output);
    output<<endl;
  }
  return 1;
}

//////////////////////////////////////////////////////////////////

int Positive_Conti_Traverso(INPUT_FILE MATRIX,
                            const short& version=1,
                            const short& S_pair_criteria=11,
                            const float& interred_percentage=12.0,
                            const BOOLEAN& verbose=FALSE)
{
////////////////////////// input ////////////////////////////////

... // analogous to Conti_Traverso(...)

  if(A.is_nonnegative()==FALSE)
  {
```

```
    cerr<<"ERROR: int Positive_Conti_Traverso(INPUT_FILE, const "
      "BOOLEAN&):\ninput matrix has negative coefficients,\n"
      "please use the general Conti-Traverso algorithm"<<endl;
    return 0;
  }

//////////////////// computation ////////////////////////////////

  // prepare time measurement
  clock_t start, end;

  // create toric ideal
  ideal I(A,w,POSITIVE_CONTI_TRAVERSO);

  // compute the standard basis
  start=clock();
  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

////////////////////// output ///////////////////////////////////

... // analogous to Conti_Traverso(...)
}

/////////////////////////////////////////////////////////////////

int Elim_Conti_Traverso(INPUT_FILE MATRIX,
                        const short& version=1,
                        const short& S_pair_criteria=11,
                        const float& interred_percentage=12.0,
                        const BOOLEAN& verbose=FALSE)
{
/////////////////////// input ///////////////////////////////////

... // analogous to Conti_Traverso(...)

/////////////////// computation ////////////////////////////////

  // prepare time measurement
  clock_t start, end;

  // create toric ideal: this is done with the CONTI_TRAVERSO
  // constructor; some elimination has to be done later
```

```
  ideal I(A,w,CONTI_TRAVERSO);

  // compute the standard basis and eliminate auxiliary variables
  start=clock();
  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);
  I.eliminate();
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

////////////////////// output ////////////////////////////////////////

... // analogous to Conti_Traverso(...)

}

////////////////////////////////////////////////////////////////////

int Pottier(INPUT_FILE MATRIX,
            const short& version=1,
            const short& S_pair_criteria=11,
            const float& interred_percentage=12.0,
            const BOOLEAN& verbose=FALSE)
{
/////////////////////////// input //////////////////////////////////

... // analogous to Conti_Traverso(...)

///////////////////////// computation ///////////////////////////////

  // prepare time measurement
  clock_t start, end;

  // create toric ideal
  ideal I(A,w,POTTIER);

  // compute the standard basis and eliminate auxiliary variable
  start=clock();
  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);
  I.eliminate();
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;
```

```
///////////////////// output /////////////////////////////////////

... // analogous to Conti_Traverso(...)

}

/////////////////////////////////////////////////////////////////////

int Hosten_Sturmfels(INPUT_FILE MATRIX,
                     const short& version=1,
                     const short& S_pair_criteria=11,
                     const float& interred_percentage=12.0,
                     const BOOLEAN& verbose=FALSE)
{
////////////////////////// input //////////////////////////////////

... // analogous to Conti_Traverso(...)

  // read positive vector in the row space of the matrix
  // such a vector induces a homogeneous grading on the ideal
  input>>format_string;
  if(!input)
  {
    cerr<<"ERROR: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
      "input failure before reading positive row space vector,\n"
      "input format not accepted"<<endl;
    return 0;
  }
  if(strcmp(format_string,"positive"))
    cerr<<"WARNING: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
      "input file has suspicious format"<<endl;
  input>>format_string;
  if(!input)
  {
    cerr<<"ERROR: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
      "input failure before reading positive row space vector,\n"
      "input format not accepted"<<endl;
    return 0;
  }
  if(strcmp(format_string,"row"))
    cerr<<"WARNING: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
      "input file has suspicious format"<<endl;
  input>>format_string;
  if(!input)
  {
    cerr<<"ERROR: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
```

```
    "input failure before reading positive row space vector,\n"
    " input format not accepted"<<endl;
  return 0;
 }
 if(strcmp(format_string,"space"))
  cerr<<"WARNING: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
    "input file has suspicious format"<<endl;
 input>>format_string;
 if(!input)
 {
  cerr<<"ERROR: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
    "input failure before reading positive row space vector,\n"
    "input format not accepted"<<endl;
  return 0;
 }
 if(strcmp(format_string,"vector:"))
  cerr<<"WARNING: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
    "input file has suspicious format"<<endl;

 float *hom_grad=new float[variables];
 for(short i=0;i<variables;i++)
 {
  input>>hom_grad[i];
  if(!input)
  {
    cerr<<"ERROR: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
      "input failure when reading positive grading "
      "/ row space vector,\ninput format not accepted"<<endl;
    delete[] hom_grad;
    return 0;
  }
  if(hom_grad[i]<=0)
  {
    cerr<<"ERROR: int Hosten_Sturmfels(INPUT_FILE, const BOOLEAN&):\n"
      "row space vector / grading must be positive"<<endl;
    delete[] hom_grad;
    return 0;
  }
 }

////////////////////////// computation ///////////////////////////////

 // prepare time measurement
 clock_t start, end;

 // construct homogeneous term ordering
 term_ordering w(variables, hom_grad, W_REV_LEX, HOMOGENEOUS);
```

```
delete[] hom_grad;

// create toric ideal
ideal I(A,w,HOSTEN_STURMFELS);

// compute the standard basis of the saturation:

start=clock();

// determine saturation variables
short *sat_var;
short number_of_sat_var=A.hosten_shapiro(sat_var);
// memory for sat_var is allocated in the hosten_shapiro procedure

// saturate the ideal
for(short i=0;i<number_of_sat_var;i++)
{
  I.swap_variables(sat_var[i],variables-1);
  // This operation simply makes the i-th saturation variable the
  // cheapest. This involves swapping the weights in the ideal's
  // term ordering, the variables in the generating binomials and
  // rebuilding the list structure if
  // SUPPORT_DRIVEN_METHODS_EXTENDED are used.

  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);
  // This calculation saturates the ideal with respect to the i-th
  // variable.

  I.swap_variables_unsafe(sat_var[i],variables-1);
  // This operation does the same as ideal::swap_variables(...)
  // except from rebuilding the list structure. It may cause an
  // inconsistent structure, but is more efficient then the safe
  // method. Before performing a Groebner basis computation,
  // however, the ideal structure has to be rebuild. Procedures like
  // ideal::swap_variables(...) and
  // ideal::change_term_ordering_to(...) do this. But these are
  // exactly the operations that follow.
}
delete[] sat_var;

// Now the ideal is saturated. The last Groebner basis computation
// is done with respect to the term ordering induced by the
// objective function.
I.change_term_ordering_to(c);
I.reduced_Groebner_basis(version,S_pair_criteria,
```

```
                         interred_percentage);
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

///////////////////////// output //////////////////////////////////

... // analogous to Conti_Traverso(...)

}

///////////////////////////////////////////////////////////////////

int DiBiase_Urbanke(INPUT_FILE MATRIX,
                    const short& version=1,
                    const short& S_pair_criteria=11,
                    const float& interred_percentage=12.0,
                    const BOOLEAN& verbose=FALSE)
{
/////////////////////////// input /////////////////////////////////

... // analogous to Conti_Traverso(...)

///////////////////////// computation /////////////////////////////

  // prepare time measurement
  clock_t start, end;

  // compute flip variables (also to check the suitability of the
  // algorithm)
  short* F;
  short r=A.compute_flip_variables(F);
  if(r<0)
    // algorithm not suitable
  {
    cout<<"ERROR: DiBiase_Urbanke(INPUT_FILE, const BOOLEAN&):\n"
      "Kernel of the input matrix contains no vector with nonzero "
      "components,\ninstance cannot be solved with this algorithm.\n"
      "No output file created. Please use another algorithm."<<endl;
    return 0;
  }

  ideal *I;
  // We use a pointer here because we need to distinguish two cases
  // for the ideal construction.
  start=clock();
```

```
if(r==0)
  // no variable flip needed
  // create toric ideal from the lattice basis already with respect
  // to the objective function
{
  I=new ideal(A,c,DIBIASE_URBANKE);
  I->reduced_Groebner_basis(version,S_pair_criteria,
                            interred_percentage);
}

else
{
  // construct term ordering to start with
  // Here we have much freedom: The term ordering must only be an
  // elimination ordering for the actual flip variable.
  // To avoid supplementary functions to manipulate term orderings,
  // we simply realize this as follows: The weight of the actual
  // flip variable is set to 1, all other weights are set to
  // zero. This still allows the use of different term orderings (by
  // setting the refining ordering).

  float* weights=new float[variables];
  for(short j=0;j<variables;j++)
    weights[j]=0;
  weights[F[0]]=1;
  term_ordering w(variables, weights, W_LEX);
  // Which term ordering is the best here?
  delete[] weights;

  // create toric ideal
  I=new ideal(A,w,DIBIASE_URBANKE);
  I->reduced_Groebner_basis(version,S_pair_criteria,
                            interred_percentage);
  I->flip_variable_unsafe(F[0]);
  // "unsafe" means that head and tail can be exchanged without
  // rebuilding the list structure (cf. the routine
  // ideal::swap_variables_unsafe(...) )
  // But the following change of the term ordering will correct this.

  for(short l=1;l<r;l++)
  {
    w.swap_weights(F[l-1],F[l]);
    // This means concretely:
    // The weight of the variable x_F[l-1] is set to zero, that of
    // x_F[l] to one.
    // Now, x_F[l] is the elimination variable.
```

```
      I->change_term_ordering_to(w);
      I->reduced_Groebner_basis(version,S_pair_criteria,
                                interred_percentage);
      I->flip_variable_unsafe(F[l]);
    }
    // Now we have a generating system of the saturated ideal.
    // Compute Groebner basis with respect to the objective function.
    I->change_term_ordering_to(c);
    I->reduced_Groebner_basis(version,S_pair_criteria,
                              interred_percentage);
  }
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

///////////////////////// output ////////////////////////////////////

... // analogous to Conti_Traverso(...)

/////////////////////// memory cleanup //////////////////////////////

  if(r>0)
    delete[] F;
  delete I;

  return 1;
}

/////////////////////////////////////////////////////////////////////

int Bigatti_LaScala_Robbiano(INPUT_FILE MATRIX,
                             const short& version=1,
                             const short& S_pair_criteria=11,
                             const float& interred_percentage=12.0,
                             const BOOLEAN& verbose=FALSE)
{
////////////////////////// input ////////////////////////////////////

... // analogous to Hosten_Sturmfels(...)

///////////////////////// computation ///////////////////////////////

  // prepare time measurement
  clock_t start, end;
```

```
  // construct homogeneous term ordering
  term_ordering w(variables, hom_grad, W_REV_LEX, HOMOGENEOUS);
  delete[] hom_grad;

  // create toric ideal
  ideal I(A,w,BIGATTI_LASCALA_ROBBIANO);

  // compute the standard basis
  start=clock();
  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);

  // now we have a generating system
  // to perform the substitution of the auxiliary variable t by the
  // saturation variables:
  // make t the revlex most expensive variable by swapping with the
  // first, recompute the Groebner basis, undo the swap
  I.swap_variables(0,variables);
  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);
  I.swap_variables(0,variables);

  I.pseudo_eliminate();
  // eliminate the auxiliary variable

  // Now the ideal is saturated. Compute the Groebner basis with
  // respect to the term ordering induced by the objective function.
  I.change_term_ordering_to(c);
  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

//////////////////////// output //////////////////////////////////

... // analogous to Conti_Traverso(...)

}

////////////////////////////////////////////////////////////////////

int solve(INPUT_FILE PROBLEM, INPUT_FILE GROEBNER)
{
  char format_string[128];
  short problem_variables;
```

```
  short elimination_variables;
  short weighted_variables;
  char elimination_refinement[128];
  char weighted_refinement[128];
  char algorithm[128];
  long size;
  long instances;
  ifstream problem(PROBLEM);
  ifstream groebner(GROEBNER);

... // verify existence of files

// read GROEBNER file

... // ( ... // without comment = code for reading file and verifying
                                  format)

  groebner>>algorithm;

... //

  if(strcmp(algorithm,"ct") && strcmp(algorithm,"pct") &&
     strcmp(algorithm,"ect") && strcmp(algorithm,"pt") &&
     strcmp(algorithm,"hs") && strcmp(algorithm,"du")
     && strcmp(algorithm,"blr"))
    cerr<<"WARNING: int solve(INPUT_FILE, INPUT_FILE):\n"
      "second input file computed with unknown algorithm"<<endl;

  // override optional lines
  do
  {
    groebner>>format_string;

... //

  }
  while(strcmp(format_string,"term"));

... //

  groebner>>elimination_variables;

... //

  if(elimination_variables<0)
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
```

```
      "number of elimination variables in second input file must be "
      "nonnegative"<<endl;
    return 0;
  }
  if(elimination_variables>0)
  {
    groebner>>elimination_refinement;

... //

    if(strcmp(elimination_refinement,"LEX") &&
       strcmp(elimination_refinement,"DEG_LEX") &&
       strcmp(elimination_refinement,"DEG_REV_LEX"))
    {
      cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
        "unknown elimination ordering in second input file"<<endl;
      return 0;
    }
  }

... //

  groebner>>weighted_variables;

... //

  if(weighted_variables<=0)
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
      "number of weighted variables in second input file must be "
      "positive"<<endl;
    return 0;
  }
  groebner>>weighted_refinement;

... //

  if(strcmp(weighted_refinement,"W_LEX") &&
     strcmp(weighted_refinement,"W_REV_LEX") &&
     strcmp(weighted_refinement,"W_DEG_LEX") &&
     strcmp(weighted_refinement,"W_DEG_REV_LEX"))
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
      "unknown weighted ordering in second input file"<<endl;
    return 0;
  }
  short weighted_ref;
```

```
  if(!strcmp(weighted_refinement,"W_LEX"))
    weighted_ref=W_LEX;
  if(!strcmp(weighted_refinement,"W_REV_LEX"))
    weighted_ref=W_REV_LEX;
  if(!strcmp(weighted_refinement,"W_DEG_LEX"))
    weighted_ref=W_DEG_LEX;
  if(!strcmp(weighted_refinement,"W_DEG_REV_LEX"))
    weighted_ref=W_DEG_REV_LEX;
  term_ordering w(weighted_variables,groebner,weighted_ref);
  if(w.error_status()<0)
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
      "input failure when reading new cost vector from second "
      "input file,\ninput format not accepted"<<endl;
    return 0;
  }
  if(w.is_nonnegative()==FALSE)
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
      "cost vectors with negative components are not supported"<<endl;
    return 0;
  }
  if(elimination_variables>0)
  {
    short elimination_ref;
    if(!strcmp(elimination_refinement,"LEX"))
      elimination_ref=LEX;
    if(!strcmp(elimination_refinement,"DEG_LEX"))
      elimination_ref=DEG_LEX;
    if(!strcmp(elimination_refinement,"DEG_REV_LEX"))
      elimination_ref=DEG_REV_LEX;
    w.convert_to_elimination_ordering(elimination_variables,
                                      elimination_ref);
    if(w.error_status()<0)
    {
      cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
        "input failure when reading new cost vector from second "
        "input file,\ninput format not accepted"<<endl;
      return 0;
    }
  }

  // read ideal

... //

  groebner>>size;
```

```
... //

  if(size<0)
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
      "ideal in second input file is corrupt"<<endl;
    return 0;
  }

... //

  ideal I(groebner,w,size);

... //

  if(I.error_status()<0)
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
      "cannot compute with corrupt ideal\n"<<endl;
    return 0;
  }

// read PROBLEM file

... //

  // read vector dimension

... //

  problem>>problem_variables;

... //

  if(problem_variables<=0)
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
      "vector dimension in first input file must be positive"<<endl;
    return 0;
  }
  // consistency with respect to the number of variables is checked
  // later

  // read number of instances

... //
```

```
  problem>>instances;

... //

  if(instances<=0)
  {
    cerr<<"WARNING: int solve(INPUT_FILE, INPUT_FILE):\n"
      "number of instances is <1, no output file created"<<endl;
    return 1;
    // no error
  }

... // read problem vectors (format string)

  // the vectors are read in the section "computation" because we need
  // to distinguish between the different algorithms

//////////////////// output (first part) ////////////////////////////

  // open output file (append mode)
  char SOLUTION[128];
  short i=0;
  while(GROEBNER[i]!='\0' && GROEBNER[i]!='.')
  {
    SOLUTION[i]=GROEBNER[i];
    i++;
  }
  SOLUTION[i]='\0';
  strcat(SOLUTION,".sol.");
  strcat(SOLUTION,algorithm);
  ofstream output(SOLUTION,ios::app);

  // format output file
  output.flags(output.flags()|ios::fixed);
  // output of fixed point numbers
  output<<"SOLUTION"<<endl<<endl;
  output<<"computed with algorithm:"<<endl;
  output<<algorithm<<endl;
  output<<"from file:"<<endl;
  output<<GROEBNER<<endl;

/////////// computation and output (second part) //////////////////

  // distinguish 3 cases to verify the consistency of the vector
  // dimension and the number of variables
```

```
// Conti-Traverso: vectors are read as right-hand vectors
// vector dimension = number of elimination variables without
// inversion variable
if(!strcmp(algorithm,"ct"))
{
  if(problem_variables!=elimination_variables-1)
  {
    cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
      "vector dimension in first input file and number of "
      "variables in\nsecond input file do not match with respect "
      "to the\nConti-Traverso-algorithm"<<endl;
    return 0;
  }

  Integer right_hand[weighted_variables+elimination_variables];

  for(short k=0;k<instances;k++)
  {
    // at the beginning, the variables of interest are zero
    for(short i=0;i<weighted_variables;i++)
      right_hand[i]=0;

    // right-hand vector is read from the input stream into the
    // elimination variables
    for(short i=weighted_variables;
        i<weighted_variables+elimination_variables-1;i++)
    {
      problem>>right_hand[i];
      if(!problem)
      {
        cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
          "input failure when reading right-hand vector "
          <<k+1<<" from first input \nfile, this and all following "
          "right-hand vectors will be ignored"<<endl;
        return 0;
      }
    }

    // determine the exponent of the inversion variable, i.e.
    // - min{negative components of right_hand}.
    Integer min=0;
    for(short i=weighted_variables;
        i<weighted_variables+elimination_variables-1;i++)
      if(right_hand[i]<min)
        min=right_hand[i];

    // transform right_hand so that all components are nonnegative
```

```
if(min<0)
  for(short i=weighted_variables;
      i<weighted_variables+elimination_variables-1;i++)
    right_hand[i]-=min;

// set exponent of the inversion variable
right_hand[weighted_variables+elimination_variables-1]=-min;

// construct binomial to reduce
binomial to_reduce(weighted_variables+elimination_variables,
                   right_hand);

// prepare time measurement
clock_t start, end;

// reduce binomial
start=clock();
I.reduce(to_reduce,TRUE);
end=clock();

// time measurement
double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

// output
output<<"right-hand vector:"<<endl;
for(short i=weighted_variables;
    i<weighted_variables+elimination_variables-1;i++)
  output<<setw(8)<<right_hand[i]+min;
  // original vector
output<<endl;
output<<"solvable:"<<endl;
BOOLEAN solvable=TRUE;
for(short i=weighted_variables;
    i<=weighted_variables+elimination_variables-1;i++)
  if(to_reduce[i]!=0)
  {
    solvable=FALSE;
    break;
  }
if(solvable==TRUE)
{
  output<<"YES"<<endl;
  output<<"optimal solution:"<<endl;
  for(short i=0;i<weighted_variables;i++)
    output<<setw(8)<<to_reduce[i];
  output<<endl;
  output<<"value:"<<endl;
```

```
        // compute optimal cost
        float value=0;
        for(short i=0;i<weighted_variables;i++)
          value+=w[i]*to_reduce[i];
        output<<setw(10)<<setprecision(2)<<value<<endl;
      }
      else
        output<<"NO"<<endl;
      output<<"computation time"<<endl;
      output<<setw(8)<<setprecision(3)<<elapsed<<" sec"<<endl<<endl;
    }
  }
  else
    // Positive Conti-Traverso: vectors are read as right-hand vectors
    // vector dimension = number of elimination variables
    if(!strcmp(algorithm,"pct"))
    {
      if(problem_variables!=elimination_variables)
      {
        cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
          "vector dimension in first input file and number of "
          "variables in\nsecond input file do not match with respect "
          "to the\nPositive Conti-Traverso algorithm"<<endl;
        return 0;
      }

      Integer right_hand[weighted_variables+elimination_variables];
      BOOLEAN error=FALSE;    // to test legality of right-hand
                              // vectors

      for(short k=0;k<instances;k++)
      {
        // at the beginning, the variables of interest are zero
        for(short i=0;i<weighted_variables;i++)
          right_hand[i]=0;

        // right-hand vector is read from the input stream into the
        // elimination variables
        for(short i=weighted_variables;
            i<weighted_variables+elimination_variables;i++)
        {
          problem>>right_hand[i];
          if(!problem)
          {
            cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
              "input failure when reading right-hand vector "
              <<k+1<<" from first input \nfile, this and all following "
```

```
        "right-hand vectors will be ignored"<<endl;
      return 0;
    }
    if(right_hand[i]<0)
    {
      cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
        "right-hand vectors with negative components cannot be "
        "handled\nby the Positive Conti-Traverso algorithm,\n"
        "right-hand vector "<<k+1<<" will be ignored"<<endl;
      error=TRUE;
    }
  }
  if(error==TRUE)
  {
    error=FALSE;
    continue;
    // for-loop
  }

  // construct binomial to reduce
  binomial to_reduce(weighted_variables+elimination_variables,
                     right_hand);

  // prepare time measurement
  clock_t start, end;

  // reduce binomial
  start=clock();
  I.reduce(to_reduce,TRUE);
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

  // output
  output<<"right-hand vector:"<<endl;
  for(short i=weighted_variables;
      i<weighted_variables+elimination_variables;i++)
    output<<setw(8)<<right_hand[i];
  // original vector
  output<<endl;
  output<<"solvable:"<<endl;
  BOOLEAN solvable=TRUE;
  for(short i=weighted_variables;
      i<weighted_variables+elimination_variables;i++)
    if(to_reduce[i]!=0)
    {
```

```
          solvable=FALSE;
          break;
        }
      if(solvable==TRUE)
      {
        output<<"YES"<<endl;
        output<<"optimal solution:"<<endl;
        for(short i=0;i<weighted_variables;i++)
          output<<setw(8)<<to_reduce[i];
        output<<endl;
      output<<"value:"<<endl;
      // compute optimal cost
      float value=0;
      for(short i=0;i<weighted_variables;i++)
        value+=w[i]*to_reduce[i];
      output<<setw(10)<<setprecision(2)<<value<<endl;
      }
      else
        output<<"NO"<<endl;
      output<<"computation time"<<endl;
      output<<setw(8)<<setprecision(3)<<elapsed<<" sec"<<endl<<endl;
    }
  }
  else
    // other algorithms: vectors are read as initial solutions
    // vector dimension = number of weighted variables
  {
    if(problem_variables!=weighted_variables)
    {
      cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
        "vector dimension in first input file and number of "
        "variables in\nsecond input file do not match with respect "
        "to the\nchosen algorithm"<<endl;
      return 0;
    }

    Integer initial_solution[weighted_variables];

    for(short k=0;k<instances;k++)
    {
      // initial solution vector is read from the input stream into
      // the elimination variables
      for(short i=0;i<weighted_variables;i++)
      {
        problem>>initial_solution[i];
        if(!problem)
        {
```

```
          cerr<<"ERROR: int solve(INPUT_FILE, INPUT_FILE):\n"
            "input failure when reading right-hand vector "<<k+1
            <<" from first input \nfile, this and all following "
            "right-hand vectors will be ignored"<<endl;
          return 0;
        }
        if(initial_solution[i]<0)
          cerr<<"WARNING: int solve(INPUT_FILE, INPUT_FILE):\n"
            "initial solution vectors should be nonnegative"<<endl;
      }

      // construct binomial to reduce
      binomial to_reduce(weighted_variables,initial_solution);

      // prepare time measurement
      clock_t start, end;

      // reduce binomial
      start=clock();
      I.reduce(to_reduce,TRUE);
      end=clock();

      // time measurement
      double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

      // output
      output<<"initial solution vector:"<<endl;
      for(short i=0;i<weighted_variables;i++)
        output<<setw(8)<<initial_solution[i];
      // original vector
      output<<endl;
      output<<"optimal solution:"<<endl;
      for(short i=0;i<weighted_variables;i++)
        output<<setw(8)<<to_reduce[i];
      output<<endl;
      output<<"value:"<<endl;
      // compute optimal cost
      float value=0;
      for(short i=0;i<weighted_variables;i++)
        value+=w[i]*to_reduce[i];
      output<<setw(10)<<setprecision(2)<<value;
      output<<"computation time"<<endl;
      output<<setw(8)<<setprecision(3)<<elapsed<<" sec"<<endl<<endl;
    }
  }
  return 1;
}
```

```
////////////////////////////////////////////////////////////////////

int change_cost(INPUT_FILE GROEBNER, INPUT_FILE NEW_COST,
                const short& version=1,
                const short& S_pair_criteria=11,
                const float& interred_percentage=12.0,
                const BOOLEAN& verbose=FALSE)
{
  char format_string[128];
  short elimination_variables;
  short weighted_variables;
  char elimination_refinement[128];
  char weighted_refinement[128];
  short new_variables;
  char algorithm[128];
  long old_size;
  ifstream old(GROEBNER);
  ifstream _new(NEW_COST);

... // verify existence of files

// read first part of GROEBNER file (until term ordering reached)

... // analogous to solve(...)

// read NEW_COST file

  // read format specification
  _new>>format_string;
  if(!_new)
  {
    cerr<<"ERROR: int change_cost(INPUT_FILE, INPUT_FILE):\n"
      "input failure when reading format specification\n"
      "from second input file, input format not accepted"<<endl;
    return 0;
  }
  if(strcmp(format_string,"MATRIX") &&
     strcmp(format_string,"NEW_COST"))

  {
    cerr<<"WARNING: int change_cost(INPUT_FILE, INPUT_FILE):\n"
      "second input file has suspicious format"<<endl;
  }

  // read number of variables
```

```
... //

  _new>>new_variables;

... //

  if(new_variables<=0)
  {
    cerr<<"ERROR: int change_cost(INPUT_FILE, INPUT_FILE):\n"
      "number of variables in second input file must be positve"
      <<endl;
    return 0;
  }

  // Now we can verify consistency of both files with respect to the
  // number of weighted variables:
  if(weighted_variables!=new_variables)
  {
    cerr<<"ERROR: int change_cost(INPUT_FILE, INPUT_FILE):\n"
      "number of variables in first and second input file do not "
      "match"<<endl;
    return 0;
  }

  // read term ordering

... //

  // the term ordering to refine the weight is taken to be the same as
  // that for the old Groebner basis
  short weighted_ref;
  if(!strcmp(weighted_refinement,"W_LEX"))
    weighted_ref=W_LEX;
  if(!strcmp(weighted_refinement,"W_REV_LEX"))
    weighted_ref=W_REV_LEX;
  if(!strcmp(weighted_refinement,"W_DEG_LEX"))
    weighted_ref=W_DEG_LEX;
  if(!strcmp(weighted_refinement,"W_DEG_REV_LEX"))
    weighted_ref=W_DEG_REV_LEX;
  term_ordering w(weighted_variables,_new,weighted_ref);
  if(w.error_status()<0)
  {
    cerr<<"ERROR: int change_cost(INPUT_FILE, INPUT_FILE):\n"
      "input failure when reading new cost vector from second "
      "input file,\ninput format not accepted"<<endl;
    return 0;
  }
```

```
  if(w.is_nonnegative()==FALSE)
  {
    cerr<<"ERROR: int change_cost(INPUT_FILE, INPUT_FILE):\n"
      "cost vectors with negative components are not supported"<<endl;
    return 0;
  }
  if(elimination_variables>0)
  {
    short elimination_ref;
    if(!strcmp(elimination_refinement,"LEX"))
      elimination_ref=LEX;
    if(!strcmp(elimination_refinement,"DEG_LEX"))
      elimination_ref=DEG_LEX;
    if(!strcmp(elimination_refinement,"DEG_REV_LEX"))
      elimination_ref=DEG_REV_LEX;
    w.convert_to_elimination_ordering(elimination_variables,
                                      elimination_ref);
    if(w.error_status()<0)
    {
      cerr<<"ERROR: int change_cost(INPUT_FILE, INPUT_FILE):\n"
        "input failure when reading new cost vector from second "
        "input file,\ninput format not accepted"<<endl;
      return 0;
    }
  }

// read second part of the GROEBNER file

  // override old weight vector
  do
  {
    old>>format_string;
    if(!old)
    {
      cerr<<"ERROR: int change_cost(INPUT_FILE, INPUT_FILE):\n"
        "input failure before reading number of ideal generators \n"
        "from first input file, input format not accepted"<<endl;
      return 0;
    }
  }
  while(strcmp(format_string,"size:"));

  // read old Groebner basis

... // analogous to solve(...)

//////////////////// computation ///////////////////////////////////
```

```
  // prepare time measurement
  clock_t start, end;

  // compute new Groebner basis
  start=clock();
  I.reduced_Groebner_basis(version,S_pair_criteria,
                           interred_percentage);
  end=clock();

  // time measurement
  double elapsed=((double) (end-start))/CLOCKS_PER_SEC;

///////////////////////// output ///////////////////////////////////

  // create output file
  char NEW_GROEBNER[128];
  short i=0;
  while(NEW_COST[i]!='\0' && NEW_COST[i]!='.')
  {
    NEW_GROEBNER[i]=NEW_COST[i];
    i++;
  }
  NEW_GROEBNER[i]='\0';
  strcat(NEW_GROEBNER,".GB.");
  strcat(NEW_GROEBNER,algorithm);
  ofstream output(NEW_GROEBNER);

... // analogous to Conti_Traverso(...)

}

#endif // IP_ALGORITHMS_CC
```

A.2.7 list.cc

```
// list.cc

// implementation of class list and class list iterator

#ifndef LIST_CC
#define LIST_CC

#include "list.h"

///////////////////////////////////////////////////////////////////
/////////////////// class list ////////////////////////////////////
```

```
/////////////////////////////////////////////////////////////////

////////////// constructors and destructor ///////////////////////

list::list()
{

#ifdef DL_LIST

  element* begin_dummy=new element;

#endif  // DL_LIST

  element* end_dummy=new element;

#ifdef DL_LIST

  begin_dummy->previous=NULL;
  begin_dummy->next=end_dummy;
  begin_dummy->entry=NULL;
  begin_dummy->done=FALSE;
  begin_dummy->head_reduced=FALSE;
  end_dummy->previous=begin_dummy;

#endif  // DL_LIST

  end_dummy->next=NULL;
  end_dummy->entry=NULL;
  end_dummy->done=TRUE;
  end_dummy->head_reduced=TRUE;

#ifdef DL_LIST

  start=begin_dummy;

#endif  // DL_LIST

#ifdef SL_LIST

  start=end_dummy;

#endif  // SL_LIST

}

// The dummy elements have the following functions:
// - The end_dummy guarantees that the deletion method of the simply
```

```
//   linked list works: Deletion is done by copying the next element
//   to the actual position and then deleting the original, see below
//   for an explanation. This would cause problems when deleting the
//   last element; then the next-pointer of the preceeding element
//   would reference freed memory (it cannot be manipulated, is
//   unknown). So the end_dummy as an artificial last element avoids
//   this problem.
// - With the described deletion method for a simply linked list, the
//   start pointer of a list has never to be changed by the
//   list_iterator class (as long as the list is not empty; but with
//   the end_dummy, the list never becomes empty). So a list iterator
//   must never know on which list he operates on.
//   Doubly linked lists use another, more secure deletion method
//   (which really deletes the actual element). Therefore, the comfort
//   of a start pointer which has never to be manipulated by the
//   iterators must be reached in another way: This is the purpose of
//   the begin_dummy, an artificial first element which is referenced
//   by the start pointer and never changed.
// - With the dummy elements, we do need to distinguish the cases of
//   inserting at the beginning or deleting at the end of a list from
//   other insert/delete operations.
// - Furthermore, the end_dummy is a stopper for Buchberger's
//   algorithm: By setting all flags to TRUE, we can eliminate many
//   calls of the is_at_end()-function of the list iterator class.
// The dummy elements should never be deleted (the deletion method is
// not secure)!

list::list(const list& l)
{

#ifdef DL_LIST

  element* begin_dummy=new element;

#endif  // DL_LIST

  element* end_dummy=new element;

#ifdef DL_LIST

  begin_dummy->previous=NULL;
  begin_dummy->next=end_dummy;
  begin_dummy->entry=NULL;
  begin_dummy->done=FALSE;
  begin_dummy->head_reduced=FALSE;

  end_dummy->previous=NULL;
```

```
#endif  // DL_LIST

  end_dummy->next=NULL;
  end_dummy->entry=NULL;
  end_dummy->done=TRUE;
  end_dummy->head_reduced=TRUE;

#ifdef DL_LIST

  start=begin_dummy;
  element *iter=(l.start)->next;

#endif  // DL_LIST

#ifdef SL_LIST

  element* iter=l.start;

#endif  // SL_LIST

  if(iter==NULL)
  {
    cerr<<"\nWARNING: list::list(const list&):\ntry to construct a list from"
      " a corrupt one; empty list created"<<endl;
    return;
  }
  while((iter->next)!=NULL)
    // end_dummy not reached
  {
    copy_insert(*(iter->entry));
    iter=iter->next;
  }
}

list::~list()
{

#ifdef DL_LIST

  element *iter=start->next;

#endif  // DL_LIST

#ifdef SL_LIST

  element *iter=start;
```

```
#endif  // SL_LIST

  while(iter->next!=NULL)
    // delete non-dummy elements
  {
    element* aux=iter;
    iter=iter->next;
    delete aux->entry;
    delete aux;
  }
  // end_dummy reached, delete it
  delete iter;

#ifdef DL_LIST

  // delete begin_dummy
  delete start;

#endif  // DL_LIST
}

///////////////// inserting ////////////////////////////////

// For a better overview, the code for the simply and for the doubly
// linked list is separated (except from the "copy-insert"-routines).

#ifdef DL_LIST

list& list::insert(binomial& bin)
{
  // insert at the beginning (after the begin_dummy)

  // initialize element
  element* aux=new element;
  aux->entry=&bin;
  aux->done=FALSE;
  aux->head_reduced=FALSE;

  // set pointers
  aux->next=start->next;
  aux->next->previous=aux;
  aux->previous=start;
  start->next=aux;

  return(*this);
}
```

```
... // list& list::_insert(binomial& bin)
    // like list& list::insert(binomial& bin) without setting done and
    // head_reduced

list& list::ordered_insert(binomial& bin, const term_ordering& w)
{
  // search for the place to insert
  element* iter=start->next;
  while(iter->entry!=NULL)
    // end_dummy not reached
  {
    if(w.compare(*(iter->entry),bin)<=0)
      // bin is bigger in term ordering then the referenced binomial
      iter=iter->next;
    else
      break;
  }
  // now bin is smaller in term ordering then the referenced binomial
  // or the referenced binomial is the end_dummy

  // initialize element
  element*aux=new element;
  aux->entry=&bin;
  aux->done=FALSE;
  aux->head_reduced=FALSE;

  // set pointers
  aux->previous=iter->previous;
  aux->previous->next=aux;
  aux->next=iter;
  iter->previous=aux;

  return *this;
}

... // list& list::_ordered_insert(binomial& bin, const term_ordering& w)

#endif  // DL_LIST

#ifdef SL_LIST

list& list::insert(binomial& bin)
{
  // insert at the beginning

  // initialize element
```

```
  element* aux=new element;
  aux->entry=&bin;
  aux->done=FALSE;
  aux->head_reduced=FALSE;

  // set pointers
  aux->next=start;
  start=aux;

  return(*this);
}

... // list& list::_insert(binomial& bin)

list& list::ordered_insert(binomial& bin, const term_ordering& w)
{
  // search for the place to insert
  element* iter=start;
  while(iter->entry!=NULL)
    // end_dummy not reached
  {
    if(w.compare(*(iter->entry),bin)<=0)
      // bin is bigger in term ordering then the referenced binomial
      iter=iter->next;
    else
      break;
  }
  // now bin is smaller in term ordering then the referenced binomial
  // or the referenced binomial is the end_dummy

  // here we have to consider a special case
  if(iter==start)
    return insert(bin);

  // insert new element by first allocating a new list place behind
  // the referenced element, then moving the referenced element to
  // that new place...
  element*aux=new element;
  aux->entry=iter->entry;
  aux->done=iter->done;
  aux->head_reduced=iter->head_reduced;
  aux->next=iter->next;

  // .. and finally inserting bin at the old place
  // Remember that we cannot insert a new element between the
  // referenced element and its predecessor (we do not know the
  // preceeding element)
```

```
  iter->entry=&bin;
  iter->done=FALSE;
  iter->head_reduced=FALSE;
  iter->next=aux;

  return *this;
}

... // list& list::_ordered_insert(binomial& bin, const term_ordering& w)

#endif  // SL_LIST

// copy-insert routines

list& list::copy_insert(const binomial& bin)
{
  binomial* _bin=new binomial(bin);
  return insert(*_bin);
}

... // list& list::_copy_insert(const binomial& bin)
... // list& list::ordered_copy_insert(const binomial& bin,
    //         const term_ordering& w)
... // list& list::_ordered_copy_insert(const binomial& bin,
    //                                const term_ordering& w)

////////////////////// output //////////////////////////////////////

void list::ordered_print(ofstream& output,
                         const term_ordering& w) const
{

#ifdef DL_LIST

  element* iter=start->next;

#endif  // DL_LIST

#ifdef SL_LIST

  element *iter=start;

#endif  // SL_LIST

  if(iter==NULL)
  {
    cerr<<"\nWARNING: void list::ordered_print"
```

```
      "(const term_ordering&) const:\n"
      "cannot print corrupt list"<<endl;
    output<<"\nWARNING: void list::ordered_print"
      "(const term_ordering&) const:\n"
      "cannot print corrupt list\n"<<endl;
    return;
  }

  list aux;
  while(iter->next!=NULL)
  {
    aux._ordered_insert(*(iter->entry),w);
    iter=iter->next;
  }
  aux.print(output);

  // delete aux, but only the element structs, not the binomials
  // (these are still stored in the actual list!)

#ifdef DL_LIST

  iter=(aux.start)->next;

#endif  // DL_LIST

#ifdef SL_LIST

  iter=aux.start;

#endif

  while((iter->next)!=NULL)
  {
    element* aux1=iter->next;
    iter->next=iter->next->next;
    delete aux1;
  }
  // the dummy elements are deleted by the destructor
}

... // other output routines

//////////////////////////////////////////////////////////////////////
////////////////// class list_iterator ///////////////////////////////
//////////////////////////////////////////////////////////////////////

// implementation of class list_iterator
```

```
// Most of these function can be inlined. I have tried this and not
// improved the performance. Perhaps the compiler does this
// automatically...

///////////////////// constructors and destructor ///////////////////

list_iterator::list_iterator()
{
  actual=NULL;
}

list_iterator::list_iterator(list& l)
{

#ifdef DL_LIST

  actual=(l.start)->next;

#endif // DL_LIST

#ifdef SL_LIST

  actual=l.start;

#endif // SL_LIST
}

list_iterator::list_iterator(list_iterator& iter)
{
  actual=iter.actual;
}

list_iterator::~list_iterator()
{
}

//////////////// object information //////////////////////////////////

BOOLEAN list_iterator::element_is_marked_done() const
{
  return(actual->done);
}

... // BOOLEAN list_iterator::element_is_marked_head_reduced() const

BOOLEAN list_iterator::is_at_end() const
{
```

```
  if(actual==NULL)
  // actual references no list
    return(TRUE);

  if(actual->next==NULL)
  // actual references dummy element
    return(TRUE);

  // actual references a real element
  return(FALSE);
}

///////////////////// assigment //////////////////////////////////

list_iterator& list_iterator::set_to_list(const list& l)
{

#ifdef DL_LIST

  actual=(l.start)->next;

#endif  // DL_LIST

#ifdef SL_LIST

  actual=l.start;

#endif  // SL_LIST

  return *this;
}

list_iterator& list_iterator::operator=(const list_iterator& iter)
{
  if((&iter)!=this)
    actual=iter.actual;
  return *this;
}

list_iterator& list_iterator::next()
{
  actual=actual->next;
  return *this;
}

/////////////// comparison ////////////////////////////////////////
```

```
int list_iterator::operator==(const list_iterator& iter) const
{
  return(actual==iter.actual);
}

... // int list_iterator::operator!=(const list_iterator& iter) const

int list_iterator::next_is(const list_iterator& iter) const
{
  return ((actual->next)==iter.actual);
}

////////// manipulation of list elements //////////////////////////

// For a better overview, the code of the delete- and extract-routine
// is separated for simply and doubly linked lists.

binomial& list_iterator::get_element()
{
  return(*(actual->entry));
}

#ifdef DL_LIST

list_iterator& list_iterator::delete_element()
{
  binomial* aux1=actual->entry;
  element*  aux2=actual;
  actual->previous->next=actual->next;
  actual->next->previous=actual->previous;
  actual=actual->next;
  delete aux1;
  delete aux2;
  return *this;
}

list_iterator& list_iterator::extract_element()
{
  element* aux=actual;
  actual->previous->next=actual->next;
  actual->next->previous=actual->previous;
  actual=actual->next;
  delete aux;
  return *this;
}

#endif  // DL_LIST
```

```
#ifdef SL_LIST

// When deleting or extracting an element of a simply linked list, the
// next-pointer of the previous element cannot be manipulated (is
// unkonwn!). So deletion must be done by copying the next element to
// the actual position and then deleting the original. Notice that
// only pointers are copied, never binomials.

list_iterator& list_iterator::delete_element()
{
  binomial* aux1=actual->entry;
  element* aux2=actual->next;
  actual->done=actual->next->done;
  actual->head_reduced=actual->next->head_reduced;
  actual->entry=actual->next->entry;
  actual->next=actual->next->next;
  delete aux1;
  delete aux2;
  return *this;
}

list_iterator& list_iterator::extract_element()
{
  element* aux=actual->next;
  actual->done=actual->next->done;
  actual->head_reduced=actual->next->head_reduced;
  actual->entry=actual->next->entry;
  actual->next=actual->next->next;
  delete aux;
  return *this;
}

#endif  // SL_LIST

list_iterator& list_iterator::mark_element_done()
{
  actual->done=TRUE;
  return *this;
}

... // list_iterator& list_iterator::mark_element_undone()
... // list_iterator& list_iterator::mark_element_head_reduced()
... // list_iterator& list_iterator::mark_element_head_unreduced()

#endif  // LIST_CC
```

A.2.8 LLL.cc

```
// LLL.cc

#ifndef LLL_CC
#define LLL_CC

#include "LLL.h"

// subroutines for the LLL-algorithms

void REDI_KB(const short& k, const short& l, BigInt** b,
             const short& number_of_vectors,
             const short& vector_dimension,
             BigInt** H, BigInt* d, BigInt** lambda)
// the REDI procedure for relations(...) (to compute the Kernel Basis,
// algorithm 2.7.2 in Cohen's book)
{
#ifdef GMP
  if(abs(BigInt(2)*lambda[k][l])<=d[l+1])
#else   // GMP
#ifdef LEDA
  if(abs(2*lambda[k][l])<=d[l+1])
#else   // LEDA
  if(labs(2*lambda[k][l])<=d[l+1])
    // labs is the abs-function for long ints
#endif  // LEDA
#endif  // GMP
    return;

#ifdef GMP
  BigInt q=(BigInt(2)*lambda[k][l]+d[l+1])/(BigInt(2)*d[l+1]);
#else   // GMP
#ifdef LEDA
  rational _q(2*lambda[k][l]+d[l+1],2*d[l+1]);
  BigInt q=floor(_q);
#else   // LEDA
  long q=(long int)
         floor(((float)(2*lambda[k][l]+d[l+1]))/(2*d[l+1]));
#endif  // LEDA
#endif  // GMP

  // q is the integer quotient of the division
  // (2*lambda[k][l]+d[l+1])/(2*d[l+1]).
  // Because of the rounding mode (always towards zero) of GNU C++,
  // we cannot use the built-in integer division
  // here; it causes errors when dealing with negative
```

```
  // numbers. Therefore the complicated casts: The divident is first
  // casted to a float which causes the division result to be a
  // float. This result is explicitly rounded downwards. As the
  // floor-function returns a double (for range reasons), this has to
  // be casted to an integer again.

  for(short n=0;n<number_of_vectors;n++)
    H[k][n]-=q*H[l][n];
  // H[k]=H[k]-q*H[l]
  for(short m=0;m<vector_dimension;m++)
    b[k][m]-=q*b[l][m];
  // b[k]=b[k]-q*b[l]
  lambda[k][l]-=q*d[l+1];
  for(short i=0;i<=l-1;i++)
    lambda[k][i]-=q*lambda[l][i];
}

void REDI_IL(const short& k, const short& l, BigInt** b,
             const short& vector_dimension, BigInt* d,
             BigInt** lambda)
// the REDI procedure for the integer LLL algorithm (algorithm 2.6.7
// in Cohen's book)
{

... // like REDI_KB(...) without setting H

}

void SWAPI(const short& k, const short& k_max, BigInt** b, BigInt* d,
           BigInt** lambda)
// the SWAPI procedure of algorithm 2.6.7
{
  // exchange b[k] and b[k-1]
  // This can be done efficiently by swapping pointers (not entries).
  BigInt* swap=b[k];
  b[k]=b[k-1];
  b[k-1]=swap;

  if(k>1)
    for(short j=0;j<=k-2;j++)
    {
      // exchange lambda[k][j] and lambda[k-1][j]
      BigInt swap=lambda[k][j];
      lambda[k][j]=lambda[k-1][j];
      lambda[k-1][j]=swap;
    }
```

```
  BigInt _lambda=lambda[k][k-1];
  BigInt B=(d[k-1]*d[k+1] + _lambda*_lambda)/d[k];
  // It might be better to choose another evaluation order for this
  // formula, see below.

  for(short i=k+1;i<=k_max;i++)
  {
    BigInt t=lambda[i][k];
    lambda[i][k]=(d[k+1]*lambda[i][k-1] - _lambda*t)/d[k];
    lambda[i][k-1]=(B*t + _lambda*lambda[i][k])/d[k+1];
  }
  d[k]=B;
}

void SWAPK(const short& k, const short& k_max, BigInt** b, BigInt** H,
           char *f, BigInt* d, BigInt** lambda)
// the SWAPK procedure of algorithm 2.7.2
{
  // exchange H[k] and H[k-1]
  // This can be done efficiently by swapping pointers (not entries).
  BigInt *swap=H[k];
  H[k]=H[k-1];
  H[k-1]=swap;

  // exchange b[k] and b[k-1] by the same method
  swap=b[k];
  b[k]=b[k-1];
  b[k-1]=swap;

  if(k>1)
    for(short j=0;j<=k-2;j++)
    {
      // exchange lambda[k][j] and lambda[k-1][j]
      BigInt swap=lambda[k][j];
      lambda[k][j]=lambda[k-1][j];
      lambda[k-1][j]=swap;
    }

  BigInt _lambda=lambda[k][k-1];
  if(_lambda==0)
  {
    d[k]=d[k-1];
    f[k-1]=0;
    f[k]=1;
    lambda[k][k-1]=0;
    for(short i=k+1;i<=k_max;i++)
    {
```

```
      lambda[i][k]=lambda[i][k-1];
      lambda[i][k-1]=0;
    }
  }
  else
    // lambda!=0
  {
    for(short i=k+1;i<=k_max;i++)
      lambda[i][k-1]=(_lambda*lambda[i][k-1])/d[k];

    // Multiplie lambda[i][k-1] by _lambda/d[k].
    // One could also write
    //   lambda[i][k-1]*=(lambda/d[k]);   (*)
    // Without a BigInt class, this can prevent overflows when
    // computing _lambda*lambda[i][k-1].
    // But examples show that lambda/d[k] is in general not an
    // integer. So (*) could lead to problems due to the inexact
    // floating point arithmetic... Therefore, we chose the secure
    // evaluation order in all such cases.

    BigInt t=d[k+1];
    d[k]=(_lambda*_lambda)/d[k];
    d[k+1]=d[k];
    for(short j=k+1;j<=k_max-1;j++)
      for(short i=j+1;i<=k_max;i++)
        lambda[i][j]=(lambda[i][j]*d[k])/t;
    for(short j=k+1;j<=k_max;j++)
      d[j+1]=(d[j+1]*d[k])/t;
  }
}

short relations(BigInt **b, const short& number_of_vectors,
                const short& vector_dimension, BigInt**& H)
{
// first check arguments
  if(number_of_vectors<0)
  {
    cerr<<"\nWARNING: short relations(BigInt**, const short&, "
      "const short&, BigInt**):\n"
      "argument number_of_vectors out of range"<<endl;
    return -1;
  }
  if(vector_dimension<=0)
  {
    cerr<<"\nWARNING: short relations(BigInt**, const short&, "
      "const short&, BigInt**):\n"
      "argument vector_dimension out of range"<<endl;
```

```
    return -1;
  }

// consider special case
  if(number_of_vectors==1)
    // Only one vector which has no relations if it is not zero,
    // else relation 1.
  {
    short r=1;    // Suppose the only column of the matrix is zero.
    for(short m=0;m<vector_dimension;m++)
      if(b[0][m]!=0)
        // nonzero entry detected
        r=0;
    if(r==1)
    {
      H=new (BigInt*)[1];
      H[0]=new BigInt[1];
      H[0][0]=1;
      // This is the lattice basis of the relations...
    }
    return r;
  }

// memory allocation
// The names are chosen (as far as possible) according to Cohen's
// book. However, for technical reasons, the indices do not run from 1
// to (e.g.) number_of_vectors, but from 0 to number_of_vectors-1.
// Therefore all indices are shifted by -1 in comparison with this
// book, except from the indices of the array d which has size
// number_of_vectors+1.

  H=new (BigInt*)[number_of_vectors];
  for(short n=0;n<number_of_vectors;n++)
    H[n]=new BigInt[number_of_vectors];
  char* f=new char[number_of_vectors];
  BigInt* d=new BigInt[number_of_vectors+1];
  BigInt** lambda=new (BigInt*)[number_of_vectors];
  for(short n=1;n<number_of_vectors;n++)
    lambda[n]=new BigInt[n];
  // We only need lambda[n][k] for n>k.

// Step 1: Initialization

  short k=1;
  short k_max=0;
  // for iteration
  d[0]=1;
```

```
  BigInt t=0;
  for(short m=0;m<vector_dimension;m++)
    t+=b[0][m]*b[0][m];
  // Now, t is the scalar product of b[0] with itself.

  for(short n=0;n<number_of_vectors;n++)
    for(short l=0;l<number_of_vectors;l++)
      if(n==l)
        H[n][l]=1;
      else
        H[n][l]=0;
  // Now, H equals the matrix I_(number_of_vectors).

  if(t!=0)
  {
    d[1]=t;
    f[0]=1;
  }
  else
  {
    d[1]=1;
    f[0]=0;
  }

// The other steps are not programmed with "goto" as in Cohen's book.
// Instead, we enter a do-while-loop which terminates iff
// k>=number_of_vectors.

  do
  {

// Step 2: Incremental Gram-Schmidt

    if(k>k_max)
      // else we can immediately go to step 3.
    {
      k_max=k;

      for(short j=0;j<=k;j++)
        if((f[j]==0) && (j<k))
          lambda[k][j]=0;
        else
        {
          BigInt u=0;

          // compute scalar product of b[k] and b[j]
          for(short m=0;m<vector_dimension;m++)
```

```
          u+=b[k][m]*b[j][m];

        for(short i=0;i<=j-1;i++)
          if(f[i]!=0)
            u=(d[i+1]*u-lambda[k][i]*lambda[j][i])/d[i];

        if(j<k)
          lambda[k][j]=u;
        else
          // j==k
          if(u!=0)
          {
            d[k+1]=u;
            f[k]=1;
          }
          else
            // u==0
          {
            d[k+1]=d[k];
            f[k]=0;
          }
      }
  }

// Step 3: Test f[k]==0 and f[k-1]!=0

  do
  {
    if(f[k-1]!=0)
      REDI_KB(k,k-1,b,number_of_vectors,vector_dimension,H,d,lambda);
    if((f[k-1]!=0) && (f[k]==0))
    {
      SWAPK(k,k_max,b,H,f,d,lambda);

      if(k>1)
        k--;
      else
        k=1;
      // k=max(1,k-1)
    }
    else
      break;
  }
  while(1);

  // Now the conditions above are no longer satisfied.
  for(short l=k-2;l>=0;l--)
```

```
      if(f[l]!=0)
        REDI_KB(k,l,b,number_of_vectors,vector_dimension,H,d,lambda);
    k++;

// Step 4: Finished?
  }
  while(k<number_of_vectors);

// Now we have computed a lattice basis of the relations of the b[i].
// Prepare the LLL-reduction.

  // Compute the dimension r of the relations.
  short r=0;
  for(short n=0;n<number_of_vectors;n++)
    if(f[n]==0) // n==r!!
      r++;
    else
      break;

  // Delete the part of H that is no longer needed (especially the
  // vectors H[r],...,H[number_of_vectors-1]).
  BigInt **aux=H;
  if(r>0)
    H=new (BigInt*)[r];
  for(short i=0;i<r;i++)
    H[i]=aux[i];

  for(short i=r;i<number_of_vectors;i++)
    delete[] aux[i];
  delete[] aux;
  delete[] f;
  delete[] d;
  for(short i=1;i<number_of_vectors;i++)
    delete[] lambda[i];
  delete[] lambda;

  integral_LLL(H,r,number_of_vectors);
  return r;
}

short integral_LLL(BigInt** b, const short& number_of_vectors,
                   const short& vector_dimension)
{
// first check arguments
  if(number_of_vectors<0)
  {
    cerr<<"\nWARNING: short integral_LLL(BigInt**, const short&, "
```

```
      "const short&):\nargument number_of_vectors out of range"
        <<endl;
    return -1;
  }
  if(vector_dimension<=0)
  {
    cerr<<"\nWARNING: short integral_LLL(BigInt**, const short&, "
      "const short&):\nargument vector_dimension out of range"
        <<endl;
    return -1;
  }

// consider special case
  if(number_of_vectors<=1)
    // 0 or 1 input vector, nothing to be done
    return 0;

// memory allocation
// The names are chosen (as far as possible) according to Cohen's
// book. However, for technical reasons, the indices do not run from 1
// to (e.g.) number_of_vectors, but from 0 to number_of_vectors-1.
// Therefore all indices are shifted by -1 in comparison with this
// book, except from the indices of the array d which has size
// number_of_vectors+1.

  BigInt* d=new BigInt[number_of_vectors+1];
  BigInt** lambda=new (BigInt*)[number_of_vectors];
  for(short s=1;s<number_of_vectors;s++)
    lambda[s]=new BigInt[s];
  // We only need lambda[n][k] for n>k.

// Step 1: Initialization

  short k=1;
  short k_max=0;
  // for iteration
  d[0]=1;
  d[1]=0;
  for(short n=0;n<vector_dimension;n++)
    d[1]+=b[0][n]*b[0][n];
  // Now, d[1] is the scalar product of b[0] with itself.

// The other steps are not programmed with "goto" as in Cohen's book.
// Instead, we enter a do-while-loop which terminates iff k>r.

  do
  {
```

```
// Step 2: Incremental Gram-Schmidt

    if(k>k_max)
      // else we can immediately go to step 3.
    {
      k_max=k;

      for(short j=0;j<=k;j++)
      {
        BigInt u=0;
        // compute scalar product of b[k] and b[j]
        for(short n=0;n<vector_dimension;n++)
          u+=b[k][n]*b[j][n];
        for(short i=0;i<=j-1;i++)
        {
          u*=d[i+1];
          u-=lambda[k][i]*lambda[j][i];
          u/=d[i];
          //u=(d[i+1]*u-lambda[k][i]*lambda[j][i])/d[i];
        }
        if(j<k)
          lambda[k][j]=u;
        else
          // j==k
          d[k+1]=u;
      }

      if(d[k+1]==0)
      {
        cerr<<"\nERROR: void integral_LLL(BigInt**, const short&, "
          "const short&):\n"
          "input vectors must be linearly independent"<<endl;
        return -1;
      }
    }

// Step 3: Test LLL-condition

    do
    {
      REDI_IL(k,k-1,b,vector_dimension,d,lambda);
      if(4*d[k+1]*d[k-1] < 3*d[k]*d[k] - lambda[k][k-1]*lambda[k][k-1])
      {
        SWAPI(k,k_max,b,d,lambda);
        if(k>1)
          k--;
```

```
        // k=max(1,k-1)
      }
      else
        break;
    }
    while(1);

    // Now the condition above is no longer satisfied.
    for(short l=k-2;l>=0;l--)
      REDI_IL(k,l,b,vector_dimension,d,lambda);
    k++;

// Step 4: Finished?
  }
  while(k<number_of_vectors);

// Now, b contains the LLL-reduced lattice basis.
// Memory cleanup.

  delete[] d;
  for(short i=1;i<number_of_vectors;i++)
    delete[] lambda[i];
  delete[] lambda;

  return 0;
}

#endif  // LLL_CC
```

A.2.9 matrix.cc

```
// matrix.cc

// implementation of class matrix

#ifndef MATRIX_CC
#define MATRIX_CC

#include "matrix.h"

////////// constructors and destructor //////////////////////////////

matrix::matrix(const short& row_number, const short& column_number,
               Integer** entries)
    :rows(row_number),columns(column_number)
{
  _kernel_dimension=-2;
```

```
  // LLL-algorithm not yet performed

  // argument check
  if((rows<=0)||(columns<=0))
    // bad input, set "error flag"
  {
    cerr<<"\nWARNING: matrix::matrix(const short&, const short&, "
      "Integer**):\nargument out of range"<<endl;
    columns=-1;
    return;
  }

  // memory allocation and initialization
  coefficients=new (Integer*)[rows];
  for(short i=0;i<rows;i++)
    coefficients[i]=new Integer[columns];
  for(short i=0;i<rows;i++)
    for(short j=0;j<columns;j++)
      coefficients[i][j]=entries[i][j];
  // coefficients[i] is the i-th row vector
}

... // other constructors

... // matrix::~matrix()

//////////////// object properties ///////////////////////////////////

... // BOOLEAN matrix::is_nonnegative() const
... // short matrix::error_status() const
... // short matrix::row_number() const
... // short matrix::column_number() const

////////// special routines for the IP-algorithms //////////////////

short matrix::LLL_kernel_basis()
{
  // copy the column vectors of the actual matrix
  // (They are modified by the LLL-algorithm!)
  BigInt** b=new (BigInt*)[columns];
  for(short n=0;n<columns;n++)
    b[n]=new BigInt[rows];
  for(short n=0;n<columns;n++)
    for(short m=0;m<rows;m++)
      b[n][m]=coefficients[m][n];

  // compute a LLL-reduced basis of the relations of
```

```
  // b[0],...,b[columns-1]
  _kernel_dimension=relations(b,columns,rows,H);
  // The kernel lattice basis is now stored in the member H (vectors
  // H[0],...,H[_kernel_dimension-1]).

  // delete auxiliary vectors
  for(short n=0;n<columns;n++)
    delete[] b[n];
  delete[] b;

  return _kernel_dimension;
}

short matrix::compute_nonzero_kernel_vector()
{
  if(_kernel_dimension==-2)
    // lattice basis not yet computed
    LLL_kernel_basis();

  if(_kernel_dimension==-1)
  {
    cerr<<"\nWARNING: short matrix::compute_nonzero_kernel_vector"
      "(BigInt*&):\nerror in kernel basis, cannot compute the "
      "desired vector"<<endl;
    return 0;
  }
  if(_kernel_dimension==0)
  {
    cerr<<"\nWARNING: short matrix::compute_nonzero_kernel_vector"
      "(BigInt*&):\nkernel dimension is zero"<<endl;
    return 0;
  }
  // Now, the kernel dimension is positive.

  BigInt *M=new BigInt[_kernel_dimension];
  // M stores a number by which the algorithm decides which vector to
  // take next.

// STEP 1: Determine the vector with the least zero components (if it
// is not unique, choose the smallest).

  // determine number of zero components
  for(short i=0;i<_kernel_dimension;i++)
  {
    M[i]=0;
    for(short j=0;j<columns;j++)
      if(H[i][j]==0)
```

```
        M[i]+=1;
  }

  // determine minimal number of zero components
  BigInt min=columns;
  // columns is an upper bound (not reached because the kernel basis
  // cannot contain the zero vector)
  for(short i=0;i<_kernel_dimension;i++)
    if(M[i]<min)
      min=M[i];

  // add the square of the norm to the vectors with the least zero
  // components and discard the others (the norm computation is why we
  // have chosen the M[i] to be BigInts)
  for(short i=0;i<_kernel_dimension;i++)
    if(M[i]!=min)
      M[i]=-1;
    else
      for(short j=0;j<columns;j++)
        M[i]+=H[i][j]*H[i][j];
  // As the lattice basis does not contain the zero vector, at least
  // one M[i] is positive!

  // determine the start vector, i.e. the one with least zero
  // components, but smallest possible (euclidian) norm
  short min_index=-1;
  for(short i=0;i<_kernel_dimension;i++)
    if(M[i]>0)
      if(min_index==-1)
        min_index=i;
      else
        if(M[i]<M[min_index])
          min_index=i;

  // Now, H[min_index] is the vector to be transformed into one with
  // nonzero components.
  // For a better overview, it is swapped with the first vector
  // (only pointers).
  if(min_index!=0)
  {
    BigInt* swap=H[min_index];
    H[min_index]=H[0];
    H[0]=swap;
  }

// Now construct the desired vector.
// This is done by adding a linear combination of
```

```
// H[1],...,H[_kernel_dimension-1] to H[0]. It is important that the
// final result, written as a linear combination of
// H[0],...,H[_kernel_dimension-1], has coefficient 1 or -1 at H[0]
// (to make sure that it is together with
// H[1],...,H[_kernel_dimension] still a  l a t t i c e  basis).

  for(short current_position=1;current_position<columns;
      current_position++)
    // in fact, this loop will terminate before the condition in the
    // for-statement will terminate it...
  {

// STEP 2: Nonnegative vector already found?

    BOOLEAN found=TRUE;
    for(short j=0;j<columns;j++)
      if(H[0][j]==0)
        found=FALSE;
    if(found==TRUE)
      // H[0] has only positive entries,
      return 1;
    // else there are further zero components

// STEP 3: Can a further zero component be "eliminated"?
// If this is the case, find a basis vector that can do this.

    // determine number of components in each remaining vector that
    // are zero in the vector itself as well as in the already
    // constructed vector
    for(short i=current_position;i<_kernel_dimension;i++)
      M[i]=0;
    short remaining_zero_components=0;
    for(short j=0;j<columns;j++)
      if(H[0][j]==0)
      {
        remaining_zero_components++;
        for(short i=current_position;i<_kernel_dimension;i++)
          if(H[i][j]==0)
            M[i]++;
      }

    // determine minimal number of such components
    min=remaining_zero_components;
    // this is the number of zero components in H[0] and an upper
    // bound for the M[i]
    for(short i=current_position;i<_kernel_dimension;i++)
      if(M[i]<min)
```

```
        min=M[i];
    if(min==remaining_zero_components)
      // all zero components in H[0] are zero in each remaining vector
      // => desired vector does not exist
      return 0;

    // add the square of the norm to the vectors with the least common
    // zero components, discard the others
    for(short i=current_position;i<_kernel_dimension;i++)
      if(M[i]!=min)
        M[i]=-1;
      else
        for(short j=0;j<columns;j++)
          M[i]+=H[i][j]*H[i][j];
    //  Again, at least one M[i] is positive!

    // determine vector to proceed with
    // This is the vector with the least common zero components with
    // respect to H[0], but the smallest possible norm.
    short min_index=0;
    for(short i=current_position;i<_kernel_dimension;i++)
      if(M[i]>0)
        if(min_index==0)
          min_index=i;
        else
          if(M[i]<M[min_index])
            min_index=i;

    // Now, a multiple of H[min_index] will be added to the already
    // constructed vector H[0].
    // For a better handling, it is swapped with the vector at
    // current_position (only pointers).
    if(min_index!=current_position)
    {
      BigInt* swap=H[min_index];
      H[min_index]=H[current_position];
      H[current_position]=swap;
    }

// STEP 4: Choose a convenient multiple of H[current_position] to add
// to H[0]. The number of factors "mult" that have to be tested is
// bounded by the number of nonzero components in H[0] (for each such
// components, there is at most one such factor that will eliminate it
// in the linear combination H[0] + mult*H[current_position].

    found=FALSE;
    for(short mult=1;found==FALSE;mult++)
```

```
    {
      found=TRUE;
      // check if any component !=0 of H[0] becomes zero by adding
      // mult*H[current_position]
      for(short j=0;j<columns;j++)
        if(H[0][j]!=0)
          if(H[0][j]+mult*H[current_position][j]==0)
            found=FALSE;
      if(found==TRUE)
        for(short j=0;j<columns;j++)
          H[0][j]+=mult*H[current_position][j];
      else
        // try -mult
      {
        found=TRUE;
        // check if any component !=0 of H[0] becomes zero by
        // subtracting mult*H[current_position]
        for(short j=0;j<columns;j++)
          if(H[0][j]!=0)
            if(H[0][j]-mult*H[current_position][j]==0)
              found=FALSE;
        if(found==TRUE)
          for(short j=0;j<columns;j++)
            H[0][j]-=mult*H[current_position][j];
      }
    }
  }

// When reaching this line, an error must have occurred.
  cerr<<"FATAL ERROR in short matrix::compute_nonzero_kernel_vector()"
      <<endl;
  abort();
}

short matrix::compute_flip_variables(short*& F)
{
  // first compute nonzero vector
  int okay=compute_nonzero_kernel_vector();
  if(!okay)
  {
    cout<<"\nWARNING: short matrix::compute_flip_variables"
      "(short*&):\nkernel of the matrix contains no vector with "
      "nonzero components,\nno flip variables computed"<<endl;
    return -1;
  }

  // compute variables to flip; these might either be those
```

```
  // corresponding to the positive components of the kernel vector
  // without zero components or those corresponding to the negative
  // ones
  short r=0;
  // number of flip variables
  for(short j=0;j<columns;j++)
    if(H[0][j]<0)
      r++;
  // remember that all components of H[0] are !=0
  if(r==0)
    // no flip variables
    return 0;

  if(2*r>columns)
    // more negative than positive components in H[0]
    // all variables corresponding to positive components will be
    // flipped
  {
    r=columns-r;
    F=new short[r];
    short counter=0;
    for(short j=0;j<columns;j++)
      if(H[0][j]>0)
      {
        F[counter]=j;
        counter++;
      }
  }
  else
    // more positive than negative components in v (or equal number)
    // all variables corresponding to negative components will be
    // flipped
  {
    F=new short[r];
    short counter=0;
    for(short j=0;j<columns;j++)
      if(H[0][j]<0)
      {
        F[counter]=j;
        counter++;
      }
  }
  return r;
}

short matrix::hosten_shapiro(short*& sat_var)
{
```

```
if(_kernel_dimension==-2)
  // lattice basis not yet computed
  LLL_kernel_basis();

if(_kernel_dimension==-1)
{
  cerr<<"\nWARNING: short matrix::hosten_shapiro(short*&):\n"
    "error in kernel basis, cannot compute the saturation variables"
    <<endl;
  return 0;
}
if(_kernel_dimension==0)
  // the toric ideal corresponding to the kernel lattice is the zero
  // ideal, no saturation variables necessary
  return 0;
// Now, the kernel dimension is positive.

if(columns==1)
  // matrix consists of one zero column, kernel is generated by the
  // vector (1) corresponding to the toric ideal <x-1> which is
  // already saturated
  return 0;

short number_of_sat_var=0;
sat_var=new short[columns/2];
BOOLEAN* ideal_saturated_by_var=new BOOLEAN[columns];
// auxiliary array used to remember by which variables the ideal has
// still to be saturated
for(short j=0;j<columns;j++)
  ideal_saturated_by_var[j]=FALSE;

for(short k=0;k<_kernel_dimension;k++)
{
  // determine number of positive and negative components in H[k]
  // corresponding to variables by which the ideal has still to be
  // saturated
  short pos_sat_var=0;
  short neg_sat_var=0;

  for(short j=0;j<columns;j++)
  {
    if(ideal_saturated_by_var[j]==FALSE)
    {
      if(H[k][j]>0)
        pos_sat_var++;
      else
        if(H[k][j]<0)
```

```
          neg_sat_var++;
      }
    }

    // now add the smaller set to the saturation variables
    if(pos_sat_var<=neg_sat_var)
    {
      for(short j=0;j<columns;j++)
        if(ideal_saturated_by_var[j]==FALSE)
          if(H[k][j]>0)
            // ideal has to be saturated by the variables
            // corresponding to positive components
          {
            sat_var[number_of_sat_var]=j;
            ideal_saturated_by_var[j]=TRUE;
            number_of_sat_var++;
          }
          else
            if(H[k][j]<0)
              // then the ideal is automatically saturated by the
              // variables corresponding to negative components
              ideal_saturated_by_var[j]=TRUE;
    }
    else
    {
      for(short j=0;j<columns;j++)
        if(ideal_saturated_by_var[j]==FALSE)
          if(H[k][j]<0)
            // ideal has to be saturated by the variables
            // corresponding to negative components
          {
            sat_var[number_of_sat_var]=j;
            ideal_saturated_by_var[j]=TRUE;
            number_of_sat_var++;
          }
          else
            if(H[k][j]>0)
              // then the ideal is automatically saturated by the
              // variables corresponding to positive components
              ideal_saturated_by_var[j]=TRUE;
    }
  }

  // clean up memory
  delete[] ideal_saturated_by_var;

  return number_of_sat_var;
```

```
}

/////////////////// output ///////////////////////////////////////

... // output routines

#endif  // matrix.cc
```

A.2.10 solve_IP.cc

```
// solve_IP.cc

// This file provides the interface to call the implemented
// IP-algorithms. This includes a short help text on the choice of
// algorithms and their arguments and the required formats of the
// input file.

#ifndef SOLVE_IP_CC
#define SOLVE_IP_CC

#include "IP_algorithms.h"

int main(int argc, char *argv[])
{
  // initialize arguments for the IP-algorithms (Buchberger)
  // with default settings
  BOOLEAN verbose=FALSE;
  short version=1;
  short S_pair_criteria=11;
  double interreduction_percentage=12.0;

//////////////////// parse options /////////////////////////////////

  short alg_option=0;
  // no algorithm specified yet

  if(argc>=3)
    // argc has to be at least 3 (except when help is required, see
    // below): program name, (MATRIX or) GROEBNER file, PROBLEM file
  {
    for(short i=1;i<argc-2;i++)
      // these are the input options
    {
      if(!strcmp(argv[i],"-v") || !strcmp(argv[i],"--verbose"))
      {
        verbose=TRUE;
        continue;
```

```
    // read next argument
  }

  if(!strcmp(argv[i],"-V") || !strcmp(argv[i],"-version"))
  {
    // check if next argument is a valid version number
    i++;
    if(!strcmp(argv[i],"1"))
    {
      version=1;
      continue;
    }
    if(!strcmp(argv[i],"1a"))
    {
      version=0;
      continue;
    }
    if(!strcmp(argv[i],"2"))
    {
      version=2;
      continue;
    }
    if(!strcmp(argv[i],"3"))
    {
      version=3;
      continue;
    }
    // When reaching this line, the version argument was invalid.
    cerr<<"ERROR: invalid version argument"<<endl
        <<"Type ‘solve_IP --help’ for help on options."<<endl;
    return 1;
  }

  if(!strcmp(argv[i],"-S"))
  {
    // initialize argument to the use of no criteria
    S_pair_criteria=0;
    do
    {
      // Check what S-pair criterion the next argument indicates.
      // If none: The algorithm is called without S-pair criteria.
      i++;
      if(i>=argc-2)
        // file arguments reached
        break;
      if(!strcmp(argv[i],"RP"))
      {
```

```
        S_pair_criteria|=1;
        continue;
      }
      if(!strcmp(argv[i],"M"))
      {
        S_pair_criteria|=2;
        continue;
      }
      if(!strcmp(argv[i],"F"))
      {
        S_pair_criteria|=4;
        continue;
      }
      if(!strcmp(argv[i],"B"))
      {
        S_pair_criteria|=8;
        continue;
      }
      if(!strcmp(argv[i],"2"))
      {
        S_pair_criteria|=16;
        continue;
      }
      // option does not belong to an S-pair criterion
      break;
    }
    while(1);
    i--;        // reset counter so that the outer loop
    continue;   // continues with the just considered argument
  }

  if(!strcmp(argv[i],"-p"))
  {
    // read interreduction percentage
    i++;
    char** rest;
    interreduction_percentage=strtod(argv[i],rest);
    // The function strtod converts a string into a double. As the
    // specification is not correct, it may cause trouble...
    // For example, if the argument can be read as a float, the
    // value of rest should be NULL. This is not the
    // case. Therefore, we can only check by the following string
    // comparison if a float has been built. An argument as 15xy
    // is thereby considered as valid (with
    // interreduction_percentage==15.0).

    if(!(strcmp(argv[i],*rest)))
```

```
        // argument was no float
      {
        cerr<<"ERROR: invalid argument for interreduction percentage"
            <<endl
            <<"Type `solve_IP --help' for help on options."<<endl;
        return 1;
      }
      continue;
    }

    if(!strcmp(argv[i],"-alg") || !strcmp(argv[i],"--algorithm"))
    {
      // check if next argument is a valid algorithm specification
      i++;
      if(!strcmp(argv[i],"ct") || !strcmp(argv[i],"pct") ||
         !strcmp(argv[i],"ect") || !strcmp(argv[i],"pt") ||
         !strcmp(argv[i],"hs") || !strcmp(argv[i],"du") ||
         !strcmp(argv[i],"blr"))
      {
        alg_option=i;
        continue;
      }
      // When reaching this line, the algorithm argument was
      // invalid.
      cerr<<"ERROR: unkwon algorithm"<<endl
          <<"Type `solve_IP --help' for help on options."<<endl;
      return 1;
    }

    // When reaching this line, the argument is no legal option.
    cerr<<"ERROR: invalid option "<<argv[i]<<endl
        <<"Type `solve_IP --help' for help on options."<<endl;
    return 1;
  }

//////////////// call IP-algorithms ///////////////////////////////

  // open first input stream (MATRIX or GROEBNER file)
  ifstream first_input(argv[argc-2]);
  if(!first_input)
  {
    cerr<<"ERROR: cannot open first input file, possibly not found"
        <<endl;
    return 1;
  }
  // open second input stream (PROBLEM file)
  ifstream second_input(argv[argc-1]);
```

```
if(!second_input)
{
  cerr<<"ERROR: cannot open second input file, possibly not found"
      <<endl;
  return 1;
}

// check if the first input file is a MATRIX file; in that case,
// we have to compute a Groebner basis first
char format_string[128];
first_input>>format_string;

if(!strcmp(format_string,"MATRIX"))
{
  if(alg_option==0)
  {
    cerr<<"ERROR: first input file is a MATRIX file;\n"
      "algorithm has to be specified"<<endl;
    return 1;
  }

  // perform specified algorithm to compute a Groebner basis of
  // the toric ideal defined by the input matrix
  int success;
  if(!strcmp(argv[alg_option],"ct"))
    success=Conti_Traverso(argv[argc-2], version, S_pair_criteria,
                           interreduction_percentage, verbose);
  if(!strcmp(argv[alg_option],"pct"))
    success=Positive_Conti_Traverso(argv[argc-2], version,
                                    S_pair_criteria,
                                    interreduction_percentage,
                                    verbose);
  if(!strcmp(argv[alg_option],"ect"))
    success=Elim_Conti_Traverso(argv[argc-2], version,
                                S_pair_criteria,
                                interreduction_percentage,
                                verbose);
  if(!strcmp(argv[alg_option],"pt"))
    success=Pottier(argv[argc-2], version, S_pair_criteria,
                    interreduction_percentage, verbose);
  if(!strcmp(argv[alg_option],"hs"))
    success=Hosten_Sturmfels(argv[argc-2], version,
                             S_pair_criteria,
                             interreduction_percentage, verbose);
  if(!strcmp(argv[alg_option],"du"))
    success=DiBiase_Urbanke(argv[argc-2], version,
                            S_pair_criteria,
```

```
                                interreduction_percentage, verbose);
      if(!strcmp(argv[alg_option],"blr"))
        success=Bigatti_LaScala_Robbiano(argv[argc-2], version,
                                         S_pair_criteria,
                                         interreduction_percentage,
                                         verbose);
      if(!success)
      {
        cerr<<"ERROR: could not perform required IP-algorithm,\n"
          "no GROEBNER file created"<<endl;
        return 1;
      }

      // Groebner basis successfully computed
      // now solve problems
      char* GROEBNER=new char[128];
      short i=0;
      while(argv[argc-2][i]!='\0' && argv[argc-2][i]!='.')
      {
        GROEBNER[i]=argv[argc-2][i];
        i++;
      }
      strcat(GROEBNER,".GB.");
      strcat(GROEBNER,argv[alg_option]);

      success=solve(argv[argc-1],GROEBNER);
      delete[] GROEBNER;
      return(!success);
      // ! because 0 indicates in general a regular program
      // termination
    }

    else
      // first input file is no MATRIX file, GROEBNER file assumed
    {
      if(argc>3)
        // input options are ignored when program is called with a
        // GROEBNER file
        cerr<<"WARNING: program called with a GROEBNER file,\n"
          "specified options will not have any effect"<<endl;
          return(!solve(argv[argc-1],argv[argc-2]));
    }
  }

////////////////// provide help text //////////////////////////////

  // handle the case of "missing" arguments
```

```
  if((argc==2) && !strcmp(argv[1],"--help"))
    // help required
  {
    system("less solve_IP.hlp");
    return 0;
  }
  else
    // invalid arguments
    cerr<<"USAGE: solve_IP [options] toric_file problem_file"<<endl
        <<"Type `solve_IP --help' for help on options."<<endl;
  return 1;
}

#endif  // SOLVE_IP_CC
```

A.2.11 term_ordering.cc

```
// term_ordering.cc

// implementation of class term ordering

#ifndef TERM_ORDERING_CC
#define TERM_ORDERING_CC

#include "binomial__term_ordering.h"

/////////// constructors and destructor //////////////////////////

term_ordering::term_ordering(const short&
                             number_of_weighted_variables,
                             const float* weights,
                             const short& _weighted_ordering,
                             const short&
                             number_of_elimination_variables,
                             const short& _elimination_ordering,
                             const BOOLEAN& _homogeneous=FALSE)
    :weighted_block_size(number_of_weighted_variables),
     elimination_block_size(number_of_elimination_variables),
     homogeneous(_homogeneous)
{
  if((_weighted_ordering<4) || (_weighted_ordering>7))
    // unknown ordering refining the weight, set "error flag"
    weighted_block_size=-1;
  else
    weighted_ordering=_weighted_ordering;
  if((_elimination_ordering<1) || (_elimination_ordering>3))
    // unknown ordering on the elimination variables, set "error flag"
```

```
    weighted_block_size=-1;
  else
    elimination_ordering=_elimination_ordering;
  if((weighted_block_size<0)||(elimination_block_size<0))
    // argument out of range, set "error flag"
    weighted_block_size=-1;

  if(weighted_block_size>0)
  {
    weight_vector=new float[weighted_block_size];
    BOOLEAN negative_weight=FALSE;
    BOOLEAN zero_weight=FALSE;
    // for checking the input

    for(short i=0;i<weighted_block_size;i++)
    {
      weight_vector[i]=weights[i];
      // initialize weight vector with weights
      if(weight_vector[i]<0)
        negative_weight=TRUE;
      if(weight_vector[i]==0)
        zero_weight=TRUE;
    }
    if(negative_weight==TRUE)
      cerr<<"\nWARNING:term_ordering::term_ordering(const short&, "
        "const float*, const short&, const short&, const short&):\n"
        "weight vector with negative components does not define "
        "a well ordering"<<endl;
    if((weighted_ordering==W_REV_LEX) && (zero_weight==TRUE))
      cerr<<"\nWARNING:term_ordering::term_ordering(const short&, "
        "const float*, const short&, const short&, const short&):\n"
        "zero weights refined by a reverse lexicographical "
        "ordering do not define a well ordering"<<endl;
  }
  else
    if(weighted_block_size<0)
      cerr<<"\nWARNING:term_ordering::term_ordering(const short&, "
        "const float*, const short&, const short&, const short&):\n"
        "bad input in term ordering constructor"<<endl;
}

... // other (easier) constructors

... // term_ordering::~term_ordering()

///////////// object properties //////////////////////////////////
```

```
... // short term_ordering::number_of_weighted_variables() const
... // short term_ordering::weight_refinement() const
... // short term_ordering::number_of_elimination_variables() const
... // short term_ordering::elimination_refinement() const
... // BOOLEAN term_ordering::is_homogeneous() const
... // short term_ordering::error_status() const
... // BOOLEAN term_ordering::is_nonnegative() const
... // BOOLEAN term_ordering::is_positive() const

////////////// frequently used comparison functions /////////////////

double term_ordering::weight(const Integer* v) const
{
  double result=0;
  for(short i=0;i<weighted_block_size;i++)
    result+=(weight_vector[i]*v[i]);
  return(result);
}

short term_ordering::compare_to_zero(const Integer* v) const
{
  unsigned short last_index
          =weighted_block_size+elimination_block_size;
  double w=0;

  // First check the elimination variables.
  if(elimination_block_size>0)
    switch(elimination_ordering)
    {
        case LEX:
          for(short i=weighted_block_size;i<last_index;i++)
          {
            Integer actual_component=v[i];
            if(actual_component>0)
              return 1;
            if(actual_component<0)
              return -1;
          }
        break;

        case DEG_LEX:
          // compute the degree
          for(short i=weighted_block_size;i<last_index;i++)
            w+=v[i];
          if(w>0)
            return 1;
          if(w<0)
```

```
          return -1;
        // if the degree is zero:
        // tie breaking with the lexicographical ordering
        for(short i=weighted_block_size;i<last_index;i++)
        {
          Integer actual_component=v[i];
          if(actual_component>0)
            return 1;
          if(actual_component<0)
            return -1;
        }
        break;

      case DEG_REV_LEX:
        //compute the degree
        for(short i=weighted_block_size;i<last_index;i++)
          w+=v[i];
        if(w>0)
          return 1;
        if(w<0)
          return -1;
        // if the degree is zero:
        // tie breaking with the reverse lexicographical ordering
        for(short i=last_index-1;i>=weighted_block_size;i--)
        {
          Integer actual_component=v[i];
          if(actual_component<0)
            return 1;
          if(actual_component>0)
            return -1;
        }
  }

// When reaching this line, the vector components corresponding to
// elimination variables are all zero.
// Compute the weight.
// If the term ordering is a homogeneous one, this is superfluous.
if(!homogeneous)
{
  w=weight(v);
  if(w>0)
    return 1;
  if(w<0)
    return -1;
}

// When reaching this line, the weight of the vector components
```

```
// corresponding to weighted variables is zero.
// Tie breaking with the term ordering refining the weight.
switch(weighted_ordering)
{
    case W_LEX:
      for(short i=0;i<weighted_block_size;i++)
      {
        Integer actual_component=v[i];
        if(actual_component>0)
          return 1;
        if(actual_component<0)
          return -1;
      }
      break;

    case W_REV_LEX:
      for(short i=weighted_block_size-1;i>=0;i--)
      {
        Integer actual_component=v[i];
        if(actual_component<0)
          return 1;
        if(actual_component>0)
          return -1;
      }
      break;

    case W_DEG_LEX:
      for(short i=0;i<weighted_block_size;i++)
        w+=v[i];
      if(w>0)
        return 1;
      if(w<0)
        return -1;
      for(short i=0;i<weighted_block_size;i++)
      {
        Integer actual_component=v[i];
        if(actual_component>0)
          return 1;
        if(actual_component<0)
          return -1;
      }
      break;

    case W_DEG_REV_LEX:
      for(short i=0;i<weighted_block_size;i++)
        w+=v[i];
      if(w>0)
```

```
          return 1;
        if(w<0)
          return -1;
        for(short i=weighted_block_size-1;i>=0;i--)
        {
          Integer actual_component=v[i];
          if(actual_component<0)
            return 1;
          if(actual_component>0)
            return -1;
        }
  }
  // When reaching this line, the argument vector is the zero vector.
  return 0;
}

short term_ordering::compare(const binomial& bin1,
                             const binomial& bin2) const
{
  unsigned short last_index
                =weighted_block_size+elimination_block_size;
  double w1=0;
  double w2=0;
  Integer* v1=bin1.exponent_vector;
  Integer* v2=bin2.exponent_vector;

  // First compare the heads of the input binomials.
  // The code is analogous to the routine compare_to_zero(...), except
  // from the fact that we have to consider the sign of the vector
  // components.

  // First check the elimination variables.
  if(elimination_block_size>0)
    switch(elimination_ordering)
    {
        case LEX:
          for(short i=weighted_block_size;i<last_index;i++)
          {
            Integer comp1=v1[i];
            Integer comp2=v2[i];
            if(comp1>0 || comp2>0)
            {
              if(comp1>comp2)
                return 1;
              if(comp1<comp2)
                return -1;
            }
```

```
  }
break;

case DEG_LEX:
  // compute the degree of the heads in the elimination
  // variables
  for(short i=weighted_block_size;i<last_index;i++)
  {
    Integer comp1=v1[i];
    Integer comp2=v2[i];
    if(comp1>0)
      w1+=comp1;
    if(comp2>0)
      w2+=comp2;
  }
  if(w1>w2)
    return 1;
  if(w1<w2)
    return -1;
  // if the degree is equal:
  // tie breaking with the lexicographical ordering
  for(short i=weighted_block_size;i<last_index;i++)
  {
    Integer comp1=v1[i];
    Integer comp2=v2[i];
    if(comp1>0 || comp2>0)
    {
      if(comp1>comp2)
        return 1;
      if(comp1<comp2)
        return -1;
    }
  }
  break;

case DEG_REV_LEX:
  //compute the degree of the heads in the elimination
  //variables
  for(short i=weighted_block_size;i<last_index;i++)
  {
    Integer comp1=v1[i];
    Integer comp2=v2[i];
    if(comp1>0)
      w1+=comp1;
    if(comp2>0)
      w2+=comp2;
  }
```

```
        if(w1>w2)
          return 1;
        if(w1<w2)
          return -1;
        // if the degree is equal:
        // tie breaking with the reverse lexicographical ordering
        for(short i=last_index-1;i>=weighted_block_size;i--)
        {
          Integer comp1=v1[i];
          Integer comp2=v2[i];
          if(comp1>0 || comp2>0)
          {
            if(comp1<comp2)
              return 1;
            if(comp1>comp2)
              return -1;
          }
        }
  }

// When reaching this line, the heads are equal in the elimination
// variables.
// Compute the weight of the heads.
w1=0;
for(short i=0;i<weighted_block_size;i++)
{
  Integer actual_component=v1[i];
  if(actual_component>0)
    w1+=actual_component*weight_vector[i];
}
w2=0;
for(short i=0;i<weighted_block_size;i++)
{
  Integer actual_component=v2[i];
  if(actual_component>0)
    w2+=actual_component*weight_vector[i];
}
if(w1>w2)
  return 1;
if(w1<w2)
  return -1;

// When reaching this line, the weight of the heads in the weighted
// variables are equal.
// Tie breaking with the term ordering refining the weight.
switch(weighted_ordering)
{
```

```
case W_LEX:
  for(short i=0;i<weighted_block_size;i++)
  {
    Integer comp1=v1[i];
    Integer comp2=v2[i];
    if(comp1>0 || comp2>0)
    {
      if(comp1>comp2)
        return 1;
      if(comp1<comp2)
        return -1;
    }
  }
  break;

case W_REV_LEX:
  for(short i=weighted_block_size-1;i>=0;i--)
  {
    Integer comp1=v1[i];
    Integer comp2=v2[i];
    if(comp1>0 || comp2>0)
    {
      if(comp1<comp2)
        return 1;
      if(comp1>comp2)
        return -1;
    }
  }
  break;

case W_DEG_LEX:
  for(short i=0;i<weighted_block_size;i++)
  {
    Integer comp1=v1[i];
    Integer comp2=v2[i];
    if(comp1>0)
      w1+=comp1;
    if(comp2>0)
      w2+=comp2;
  }
  if(w1>w2)
    return 1;
  if(w1<w2)
    return -1;
  for(short i=0;i<weighted_block_size;i++)
  {
    Integer comp1=v1[i];
```

```
        Integer comp2=v2[i];
        if(comp1>0 || comp2>0)
        {
          if(comp1>comp2)
            return 1;
          if(comp1<comp2)
            return -1;
        }
      }
      break;

    case W_DEG_REV_LEX:
      for(short i=0;i<weighted_block_size;i++)
      {
        Integer comp1=v1[i];
        Integer comp2=v2[i];
        if(comp1>0)
          w1+=comp1;
        if(comp2>0)
          w2+=comp2;
      }
      if(w1>w2)
        return 1;
      if(w1<w2)
        return -1;
      for(short i=weighted_block_size-1;i>=0;i--)
      {
        Integer comp1=v1[i];
        Integer comp2=v2[i];
        if(comp1>0 || comp2>0)
        {
          if(comp1<comp2)
            return 1;
          if(comp1>comp2)
            return -1;
        }
      }
}

// When reaching this line, the heads of the binomials are equal.
// Now we decide by the tails.
// This part of the code could also be omitted in the current
// context: The compare(...)-function is only called when dealing
// with ordered lists. This is done in two cases:
// - When computing with ordered S-pair lists, it doesn't really
//   matter if such similar binomials are in the right order.
// - When outputting a reduced Groebner basis, it cannot happen that
```

```
  //   two heads are equal.

  w1=0;
  w2=0;

... // analogous to the head comparison
    // v1[i] / v2[i] replaced by -v1[i] / -v2[i]

  return 0;
}

/////////// assignment and access functions ///////////////////////

term_ordering& term_ordering::operator=(const term_ordering& w)
{
  if(&w==this)
    return *this;

  if(weighted_block_size>0)
    delete[] weight_vector;
  weighted_block_size=w.weighted_block_size;
  weighted_ordering=w.weighted_ordering;
  elimination_block_size=w.elimination_block_size;
  elimination_ordering=w.elimination_ordering;
  homogeneous=w.homogeneous;

  if(weighted_block_size>0)
  {
    weight_vector=new float[weighted_block_size];
    for(short i=0;i<weighted_block_size;i++)
      weight_vector[i]=w.weight_vector[i];
  }
  else
    if(weighted_block_size<0)
      cerr<<"\nWARNING: term_ordering& term_ordering::"
        "operator=(const term_ordering&):\n"
        "assignment from corrupt term ordering"<<endl;

  return(*this);
}

float term_ordering::operator[](const short& i) const
{
  if((i<0) || (i>=weighted_block_size))
  {
    cerr<<"\nWARNING: float term_ordering::operator[]"
      "(const short& i):\n"
```

```
      "access to invalid weight vector component"<<endl;
    return FLT_MAX;
  }
  else
    return weight_vector[i];
}

///////// routines needed by the IP-algorithms ////////////////////
///////// to manipulate the term ordering /////////////////////////

term_ordering& term_ordering::convert_to_weighted_ordering()
{
  elimination_block_size=0;
  return(*this);
}

term_ordering& term_ordering::convert_to_elimination_ordering
(const short& number_of_elimination_variables,
 const short& _elimination_ordering)
{
  if((_elimination_ordering<1) || (_elimination_ordering>3))
    // unknown ordering on the elimination variables, set "error flag"
    weighted_block_size=-1;
  else
    elimination_ordering=_elimination_ordering;
  if(number_of_elimination_variables<0)
    // argument out of range, set error flag
    weighted_block_size=-1;
  else
    elimination_block_size=number_of_elimination_variables;
  if(weighted_block_size<0)
    cerr<<"\nWARNING: term_ordering& term_ordering::"
      "convert_to_elimination_ordering(const short&, const short&):\n"
      "argument out of range"<<endl;
  return(*this);
}

term_ordering& term_ordering::append_weighted_variable
                              (const float& weight)
{
  if(weighted_block_size>=0)
  {
    float *aux=weight_vector;
    weight_vector=new float[weighted_block_size+1];
    for(short i=0;i<weighted_block_size;i++)
      weight_vector[i]=aux[i];
    weight_vector[weighted_block_size]=weight;
```

```
    if(weighted_block_size>0)
      delete[] aux;
    weighted_block_size++;
  }
  else
    cerr<<"\nWARNING: term_ordering& term_ordering::"
      "append_weighted_variable(const float&):\n"
      "called for a corrupt term ordering, term ordering not changed"
        <<endl;
  return(*this);
}

term_ordering& term_ordering::delete_last_weighted_variable()
{
  if(weighted_block_size>0)
  {
    float *aux=weight_vector;
    if(weighted_block_size>1)
      weight_vector=new float[weighted_block_size-1];
    for(short i=0;i<weighted_block_size-1;i++)
      weight_vector[i]=aux[i];
    weighted_block_size--;
    delete[] aux;
  }
  else
    cerr<<"\nWARNING: term_ordering& term_ordering::"
      "delete_last_weighted_variable():\n"
      "called for a maybe corrupt term ordering without weighted "
      "variables,\nterm ordering not changed"<<endl;
  return(*this);
}

term_ordering& term_ordering::swap_weights(const short& i,
                                           const short& j)
{
  if((i<0) || (i>=weighted_block_size) ||
     (j<0) || (j>=weighted_block_size))
  {
    cout<<"\nWARNING: term_ordering& termordering::swap_weights"
      "(const short, const short):\nindex out of range"<<endl;
    return *this;
  }
  float swap=weight_vector[j];
  weight_vector[j]=weight_vector[i];
  weight_vector[i]=swap;
  return *this;
}
```

```
/////////////////// output /////////////////////////////////////////

... // output routines

#endif  // TERM_ORDERING_CC
```

A.2.12 toric_ideal.cc

```
// toric_ideal.cc

// This file provides an interface to compute toric ideals from a
// matrix without solving IP-problems.

#ifndef TORIC_IDEAL_CC
#define TORIC_IDEAL_CC

#include "IP_algorithms.h"

int main(int argc, char *argv[])
{
  // initialize arguments for the Buchberger algorithm
  // with default settings
  BOOLEAN verbose=FALSE;
  short version=1;
  short S_pair_criteria=11;
  double interreduction_percentage=12.0;

///////////////////// parse options /////////////////////////////////

  short alg_option=0;
  // no algorithm specified yet

  if(argc>=2 && strcmp(argv[1],"--help"))
    // argc has to be at least 2
    // program name, MATRIX file
  {

... // analogous to solve_IP.cc

////////// call algorithms for computing toric ideals //////////////

    // open input stream (MATRIX file)
    ifstream input(argv[argc-1]);
    if(!input)
    {
      cerr<<"ERROR: cannot open input file, possibly not found"<<endl;
```

```
    return 1;
  }
  if(alg_option==0)
  {
    cerr<<"ERROR: no valid algorithm specified"<<endl;
    return 1;
  }

  // perform specified algorithm to compute a Groebner basis of the
  // toric ideal defined by the input matrix
  int success;
  if(!strcmp(argv[alg_option],"ct"))
    success=Conti_Traverso(argv[argc-1], version, S_pair_criteria,
                           interreduction_percentage, verbose);
  if(!strcmp(argv[alg_option],"pct"))
    success=Positive_Conti_Traverso(argv[argc-1], version,
                                    S_pair_criteria,
                                    interreduction_percentage,
                                    verbose);
  if(!strcmp(argv[alg_option],"ect"))
    success=Elim_Conti_Traverso(argv[argc-1], version,
                                S_pair_criteria,
                                interreduction_percentage, verbose);
  if(!strcmp(argv[alg_option],"pt"))
    success=Pottier(argv[argc-1], version, S_pair_criteria,
                    interreduction_percentage, verbose);
  if(!strcmp(argv[alg_option],"hs"))
    success=Hosten_Sturmfels(argv[argc-1], version, S_pair_criteria,
                             interreduction_percentage, verbose);
  if(!strcmp(argv[alg_option],"du"))
    success=DiBiase_Urbanke(argv[argc-1], version, S_pair_criteria,
                            interreduction_percentage, verbose);
  if(!strcmp(argv[alg_option],"blr"))
    success=Bigatti_LaScala_Robbiano(argv[argc-1], version,
                                     S_pair_criteria,
                                     interreduction_percentage,
                                     verbose);
  if(!success)
  {
    cerr<<"ERROR: could not perform required IP-algorithm,\n"
      "no GROEBNER file created"<<endl;
    return 1;
  }

  // Groebner basis successfully computed
  return 0;
}
```

```
//////////////////// provide help text /////////////////////////////

  // handle the case of "missing" arguments
  if((argc==2) && !strcmp(argv[1],"--help"))
    // help required
  {
    system("less toric_ideal.hlp");
    return 0;
  }
  else
    // invalid arguments
    cerr<<"USAGE: toric_ideal [options] matrix_file"<<endl
        <<"Type 'toric_ideal --help' for help on options."<<endl;

  return 1;
}

#endif  // TORIC_IDEAL_CC
```

A.3 Makefile

Folgendes Makefile kann zur Installation des Programmpakets auf vielen UNIX-ähnlichen Systemen verwendet werden:

```
# Compiler settings:
CC        := g++
CFLAGS    := -O2
LDFLAGS   := -lgmp

MAIN1 := solve_IP
OBJECTS1:= $(MAIN1).o

MAIN2 := change_cost
OBJECTS2:= $(MAIN2).o

MAIN3 := toric_ideal
OBJECTS3:= $(MAIN3).o

all: $(MAIN1) $(MAIN2) $(MAIN3)

Do_Link        = $(CC) $(filter %.o,$^) $(LDFLAGS) -o $@

%.o: %.cc
$(CC) $(CFLAGS) -c $< -o $@

clean:
```

```
rm -f *.o *~

##############################################################

$(MAIN1): term_ordering.o binomial.o ./list.o BigInt.o LLL.o \
          ./matrix.o ideal.o Buchberger.o ideal_stuff.o \
          IP_algorithms.o testdata.o \
          $(OBJECTS1)
$(Do_Link)

$(MAIN2): term_ordering.o binomial.o ./list.o BigInt.o LLL.o \
  ./matrix.o ideal.o Buchberger.o ideal_stuff.o \
  IP_algorithms.o testdata.o \
          $(OBJECTS2)
$(Do_Link)

$(MAIN3): term_ordering.o binomial.o ./list.o BigInt.o LLL.o \
          ./matrix.o ideal.o Buchberger.o ideal_stuff.o \
          IP_algorithms.o testdata.o \
          $(OBJECTS3)
$(Do_Link)
```

A.4 Hilfedateien

Die drei ausführbaren Programme des Pakets zeigen, mit der Option `--help` aufgerufen, folgende Hilfetexte an:

A.4.1 toric_ideal.hlp

```
USAGE: toric_ideal [options] matrix_file

DESCRIPTION:

toric_ideal is a program for computing the toric ideal of a matrix A.
This ideal is always given by a reduced Groebner basis which is
computed via Buchberger's algorithm.
For this purpose, we can use six different algorithms:
The algorithm of Conti/Traverso (ct) computes the toric ideal of an
extended matrix. This ideal can be used later for solving integer
programming problems for given right-hand vectors. By eliminating
auxiliary variables, we can get the toric ideal of the original matrix
from it. The same is true for the "positive" variant of the
Conti-Traverso-algorithm (pct) which can only be applied if A has
nonnegative entries.
All other algorithms compute the toric ideal of A itself. Except from
the elimination version of the Conti-Traverso algorithm (ect), they
are in general more efficient than the algorithm mentionned
```

before. These are the algorithms of Pottier (pt), Bigatti/La Scala/Robbiano (blr), Hosten/Sturmfels (hs) and Di Biase/Urbanke (du).

FILE FORMAT:

The input file has to be a MATRIX file. toric_ideal produces a GROEBNER file named like the MATRIX file with extensions replaced by

.GB.<alg>

where GB stands for GROEBNER and <alg> is the abbreviation for the used algorithm as above.

A MATRIX file should look as follows:

```
  MATRIX

  columns:
  <number of columns>

  cost vector:
  <coefficients of the cost vector>

  rows:
  <number of rows>

  matrix:
  <matrix coefficients>

  positive row space vector:
  <coefficients of row space vector>
```

The last two lines are only needed when toric_ideal is called with the algorithms of Hosten/Sturmfels or Bigatti/La Scala/Robbiano, i.e. the options

```
-alg hs
or
-alg blr
```

The other algorithms ignore these lines.

Example:

```
  MATRIX

  columns:
```

```
  3

  cost vector:
  1 1 1

  rows:
  2

  matrix:
  1 2 3
  4 5 6

  positive row space vector:
  1 2 3

A GROEBNER file looks as follows:

  GROEBNER

  computed with algorithm:
  <algorithm name abbreviation>       (* abbreviations as above *)
  from file(s):                       (* the following four lines are
  <name of respective MATRIX file>       optional *)
  computation time:
  <computation time in seconds>

  term ordering:
  elimination block
  <number of elimination variables>
  <LEX / DEG_LEX                      (* only if number of elimination
  / DEG_REV_LEX>                         variables >0 *)
  weighted block
  <number of weighted variables>
  <W_LEX / W_REV_LEX                  (* number of weighted variables
  / W_DEG_LEX / W_DEG_REV_LEX>           should always be >0 *)
  <weight_vector>

  size:
  <number of elements>

  Groebner basis:
  <basis elements>

  <settings for the Buchberger
   algorithm and compiler settings>  (* optional *)

The Groebner basis consists always of binomials of the form x^a - x^b
```

```
where x^a and x^b are relatively prime. Such a binomial can be
represented by the vector a-b. The basis elements in the GROEBNER file
are given by the coefficients of this vector representation.
The settings for Buchberger's algorithm and the compiler flags are
produced when the GROEBNER file is generated by a call of toric_ideal
with the verbose output option

-v, --verbose

Example (not belonging to the example above):

  GROEBNER

  computed with algorithm:
  du

  term ordering:
  elimination block:
  0
  weighted block:
  3
  W_LEX
  1 2 3

  size:
  1

  Groebner basis:
  2 3 -2     (*  x^2 * y^3 - z^2  *)


OPTIONS:

 -alg       <alg>,
--algorithm <alg>         algorithm to use for computing the toric
                          ideal; <alg> may be
             ct           for Conti/Traverso,
             pct          for positive Conti/Traverso,
             ect          for Conti/Traverso with elimination,
             pt           for Pottier,
             hs           for Hosten/Sturmfels,
             du           for Di Biase/Urbanke,
             blr          for Bigatti-LaScal-Robbiano.

 -p         <number>      percentage of new generators to cause an
                          autoreduction during Buchberger's algorithm;
                          <number> may be an arbitrary float, a
```

```
                             negative value allows no intermediate
                             autoreductions
  default is
                             -p 12.0

 -S [RP] [M] [B] [M] [2]  criteria to use in Buchberger's algorithm
                             to discard unneccessary S-pairs
             RP              relatively prime leading terms
             M               Gebauer-Moeller criterion M
             F               Gebauer-Moeller criterion F
             B               Gebauer-Moeller criterion B
             2               Buchberger's second criterion
  default is
                             -S RP M B

 -v,
--verbose                    verbose output mode; writes the settings for
                             Buchberger's algorithm and the compiler
                             flags into the output GROEBNER file

-V <number>,
--version <number>           version of Buchberger's algorithm to use;
                             <number> may be 1, 1a, 2 or 3
  default is
                             -V 1
```

A.4.2 solve_IP.hlp

```
USAGE: solve_IP [options] toric_file problem_file

DESCRIPTION:

solve_IP is a program for solving integer programming problems with
the Buchberger algorithm:

Let A an integral (mxn)-matrix, b a vector with m integral
coefficients and c a vector with n nonnegative real coefficients. The
solution of the IP-problem

    minimize{ cx | Ax=b, all components of x are nonnegative integers }

proceeds in two steps:

1. We compute the toric ideal of A and its Groebner basis with respect
   to a term ordering refining the cost function c.

2. We reduce the right-hand vector b or an initial solution of the
```

problem modulo this ideal.

For these purposes, we can use seven different algorithms:
The algorithm of Conti/Traverso (ct) can compute an optimal solution of the IP-problem directly from the right-hand vector b. The same is true for its "positive" variant (pct) which can only be applied if A and b have nonnegative entries.
All other algorithms need initial solutions of the IP-problem. Except from the elimination version of the Conti-Traverso algorithm (ect), they are in general more efficient than the algorithm mentionned before. These are the algorithms of Pottier (pt), Bigatti/La Scala/Robbiano (blr), Hosten/Sturmfels (hs) and Di Biase/Urbanke (du).

FILE FORMAT:

The first input file may be a MATRIX or a GROEBNER file; these two types are called "toric files". The second input file has to be a PROBLEM file.

If the PROBLEM file contains a positive number of problems ,i.e. right hand vectors or initial solutions, solve_IP appends its solutions to a SOLUTION file named like the first input file with extensions replaced by

```
.sol.<alg>
```

where sol stands for solution and <alg> is the abbreviation of the used algorithm as above. When called with a MATRIX file, solve_IP produces a GROEBNER file named like the MATRIX file with extensions replaced by

```
.GB.<alg>
```

where GB stands for GROEBNER.

A MATRIX file should look as follows:

```
MATRIX

columns:
<number of columns>

cost vector:
<coefficients of the cost vector>

rows:
<number of rows>
```

```
  matrix:
  <matrix coefficients>

  positive row space vector:
  <coefficients of row space vector>

The last two lines are only needed when solve_IP is called with the
algorithms of Hosten/Sturmfels or Bigatti/La Scala/Robbiano, i.e. the
options

-alg hs
or
-alg blr

The other algorithms ignore these lines.

Example:

  MATRIX

  columns:
  3

  cost vector:
  1 1 1

  rows:
  2

  matrix:
  1 2 3
  4 5 6

  positive row space vector:
  1 2 3

A GROEBNER file looks as follows:

  GROEBNER

  computed with algorithm:
  <algorithm name abbreviation>       (* abbreviations as above *)
  from file(s):                       (* the following four lines are
  <name of respective MATRIX file>       optional *)
  computation time:
  <computation time in seconds>
```

```
  term ordering:
  elimination block
  <number of elimination variables>
  <LEX / DEG_LEX                      (* only if number of elimination
  / DEG_REV_LEX>                         variables >0 *)
  weighted block
  <number of weighted variables>
  <W_LEX / W_REV_LEX                  (* number of weighted variables
  / W_DEG_LEX / W_DEG_REV_LEX>           should always be >0 *)
  <weight_vector>

  size:
  <number of elements>

  Groebner basis:
  <basis elements>

  <settings for the Buchberger
   algorithm and compiler settings>  (* optional *)

The Groebner basis consists always of binomials of the form x^a - x^b
where x^a and x^b are relatively prime. Such a binomial can be
represented by the vector a-b. The basis elements in the GROEBNER file
are given by the coefficients of this vector representation.
The settings for Buchberger's algorithm and the compiler flags are
produced when the GROEBNER file is generated by a call of solve_IP
with the verbose output option

-v, --verbose

Example (not belonging to the example above):

  GROEBNER

  computed with algorithm:
  du

  term ordering:
  elimination block:
  0
  weighted block:
  3
  W_LEX
  1 2 3

  size:
```

```
  1

  Groebner basis:
  2 3 -2     (*  x^2 * y^3 - z^2  *)

A PROBLEM file should look as follows:

  PROBLEM

  vector size:
  <vector dimension>

  number of instances:
  <number of vectors>

  right-hand or initial solution vectors:
  <vector coefficients>

A SOLUTION file looks as follows:

  SOLUTION

  computed with algorithm:
  <algorithm name abbreviation>
  from file:                                  (* the GROEBNER file *)
  <file name>

  <right-hand/initial solution> vector:
  <vector as in the problem file>
  solvable:                                   (* only if right-hand
  <YES/NO>                                       vector is given *)
  optimal solution:                           (* only in the case of
  <optimal solution vector>         existence *)
  value:                                      (* only in the case of
  <optimal cost>                 existence *)
  computation time:                           (* only for reduction *)
  <reduction time in seconds>

  ...                                         (* further vectors,
                                                 same format *)

OPTIONS:

 -alg       <alg>,
--algorithm <alg>          algorithm to use for solving the given IP
                           instances; <alg> may be
             ct            for Conti/Traverso,
```

```
             pct          for positive Conti/Traverso,
             ect          for Conti/Traverso with elimination,
             pt           for Pottier,
             hs           for Hosten/Sturmfels,
             du           for Di Biase/Urbanke,
             blr          for Bigatti-LaScal-Robbiano.

 -p         <number>      percentage of new generators to cause an
                          autoreduction during Buchberger's algorithm;
                          <number> may be an arbitrary float, a
                          negative value allows no intermediate
                          autoreductions
  default is
                          -p 12.0

 -S [RP] [M] [B] [M] [2]  criteria to use in Buchberger's algorithm
                          to discard unneccessary S-pairs
             RP           relatively prime leading terms
             M            Gebauer-Moeller criterion M
             F            Gebauer-Moeller criterion F
             B            Gebauer-Moeller criterion B
             2            Buchberger's second criterion
  default is
                          -S RP M B

 -v,
--verbose                 verbose output mode; writes the settings for
                          Buchberger's algorithm and the compiler
                          flags into the output GROEBNER file

-V <number>,
--version <number>        version of Buchberger's algorithm to use;
                          <number> may be 1, 1a, 2 or 3
  default is
                          -V 1

When called with a GROEBNER file, these options do not affect
computation and are ignored by solve_IP.
```

A.4.3 change_cost.hlp

```
USAGE: change_cost [options] groebner_file new_cost_file

DESCRIPTION:

change_cost is a program for recomputing the reduced Groebner basis of
a toric ideal with respect to a term ordering with modified
```

weights. This can be used for varying the cost function in integer programming (see the help for solve_IP for more information).
The program simply restarts Buchberger's algorithm on an already computed toric ideal using the modified term ordering.

FILE FORMAT:

The first input file has to be a GROEBNER file; the second input file should to be a NEW_COST file. For convenience, also MATRIX files are accepted as second input files. For their format see the help for the solve_IP or the toric_ideal program.

change_cost writes the new Groebner basis into a GROEBNER file named like the second input file with extensions replaced by

.GB.<alg>

where GB stands for GROEBNER and <alg> is the abbreviation of the algorithm used for computing the input GROEBNER file (see the help for solve_IP or toric_ideal for an explaination).

A GROEBNER file looks as follows:

```
  GROEBNER

  computed with algorithm:
  <algorithm name abbreviation>       (* abbreviations as above *)
  from file(s):                       (* the following four lines are
  <name of respective MATRIX file>       optional *)
  computation time:
  <computation time in seconds>

  term ordering:
  elimination block
  <number of elimination variables>
  <LEX / DEG_LEX                      (* only if number of elimination
  / DEG_REV_LEX>                         variables >0 *)
  weighted block
  <number of weighted variables>
  <W_LEX / W_REV_LEX                  (* number of weighted variables
  / W_DEG_LEX / W_DEG_REV_LEX>           should always be >0 *)
  <weight_vector>

  size:
  <number of elements>

  Groebner basis:
```

```
  <basis elements>

  <settings for the Buchberger
   algorithm and compiler settings>  (* optional *)

The Groebner basis consists always of binomials of the form x^a - x^b
where x^a and x^b are relatively prime. Such a binomial can be
represented by the vector a-b. The basis elements in the GROEBNER file
are given by the coefficients of this vector representation.
The settings for Buchberger's algorithm and the compiler flags are
produced when the GROEBNER file is generated by a call of solve_IP
with the verbose output option

-v, --verbose

Example:

  GROEBNER

  computed with algorithm:
  du

  term ordering:
  elimination block:
  0
  weighted block:
  3
  W_LEX
  1 2 3

  size:
  1

  Groebner basis:
  2 3 -2     (*  x^2 * y^3 - z^2  *)

A NEW_COST file should look as follows:

  NEW_COST

  variables:
  <number of weighted variables>

  cost vector:
  <coefficients of the new weight vector>

OPTIONS:
```

```
 -p          <number>      percentage of new generators to cause an
                           autoreduction during Buchberger's algorithm;
                           <number> may be an arbitrary float, a
                           negative value allows no intermediate
                           autoreductions
  default is
                           -p 12.0

 -S [RP] [M] [B] [M] [2]   criteria to use in Buchberger's algorithm
                           to discard unneccessary S-pairs
             RP            relatively prime leading terms
             M             Gebauer-Moeller criterion M
             F             Gebauer-Moeller criterion F
             B             Gebauer-Moeller criterion B
             2             Buchberger's second criterion
  default is
                           -S RP M B

 -v,
--verbose                  verbose output mode; writes the settings for
                           Buchberger's algorithm and the compiler
                           flags into the output GROEBNER file

-V <number>,
--version <number>         version of Buchberger's algorithm to use;
                           <number> may be 1, 1a, 2 or 3
  default is
                           -V 1
```

A.5 Bibliotheken für SINGULAR

Die vorläufige Anbindung an SINGULAR geschieht über zwei Bibliotheken.

A.5.1 toric.lib

```
// $Id$
//
// author : Christine Theis
//

//version="$Id$";

//////////////////////////////////////////////////////////////////////

info="
LIBRARY: toric.lib PROCEDURES FOR COMPUTING TORIC IDEALS
```

```
Let A an integral (mxn)-matrix. The toric ideal I_A of A is defined
as the ideal

    I_A:=< x^u - x^v | u,v integral and nonnegative, u-v in ker(A) >

in the ring of n variables x:=x1,...,xn.
Toric ideals play an important role in polyhedral geometry and may
also be used for integer programming. They are generated by binomials
with relatively prime monomials. Buchberger’s algorithm can be
specialized to these structures in a way that considerably speeds up
computation.

toric_ideal(intmat A, string alg);
toric_ideal(intmat A, string alg, intvec prsv);

    procedures for computing the toric ideal of A
    They return the standard basis of the toric ideal of A with
    respect to the term ordering in the actual basering.
    When calling this procedure, a ring with n variables should be
    active.
    Not all term orderings are supported: The usual global term
    orderings may be used, but no block orderings combining them.
    One may call the procedure with several different algorithms:

        - the algorithm of Conti/Traverso using elimination (ect),
        - the algorithm of Pottier (pt),
        - an algorithm of Bigatti/La Scala/Robbiano (blr),
        - the algorithm of Hosten/Sturmfels (hs),
        - the algorithm of DiBiase/Urbanke (du).

    (pt),(hs) and (du) are in general more efficient than (ect) and
    (blr) in the actual implementation.
    The argument ‘alg’ should be the abbreviation (in brackets) for an
    algorithm as above.
    If ‘alg’ is chosen to be ‘blr’ or ‘hs’, the algorithm needs a
    vector with positive coefficcients in the row space of A. If
    no row of A contains only positive entries, one must use the
    second version of toric_ideal which takes such a vector as its
    third argument.

toric_std(ideal I);

    computes the standard basis of I using the specialized Buchberger
    algorithm. The generating system by which I is given has to
    consist of binomials of the form x^u-x^v (although there are other
    generating systems of toric ideals). There is no real check if I
```

```
    is toric. If the generator list of I contains a binomial whose
    monomials are not relatively prime, the procedure outputs a
    warning. If I is generated by binomials of the above form, but not
    toric, toric_std computes an ideal ‘between’ I and its saturation
    with respect to all variables.
";

////////////////////////////////////////////////////////////////////

static proc toric_ideal_1(intmat A, string alg)
{
  ideal I;
  // to be returned

  // check suitability of actual basering
  if(nvars(basering)<ncols(A))
  {
    "ERROR: number of matrix columns must be greater or equal number
    of ring variables";
    return(I);
  }

  // check suitability of actual term ordering
  // the "module ordering" c or C is ignored
  string singular_ord=ordstr(basering);
  string test_ord;
  string external_ord="";
  int i,j;
  intvec weightvec;

  if(find(singular_ord,"lp")==1)
  {
    external_ord="W_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=0;
    }
    test_ord="lp("+string(nvars(basering))+"),";
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported; lp used for
      computation";
    }
  }
  if(external_ord=="" && find(singular_ord,"lp")==3)
  {
    external_ord="W_LEX";
```

```
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=0;
    }
    test_ord="lp("+string(nvars(basering))+"),";
    if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
    {
      "Warning: block orderings are not supported; lp used for
      computation";
    }
  }

  if(external_ord=="" && find(singular_ord,"dp")==1)
  {
    external_ord="W_REV_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=1;
    }
    test_ord="dp("+string(nvars(basering))+"),";
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported; dp used for
      computation";
    }
  }
  if(external_ord=="" && find(singular_ord,"dp")==3)
  {
    external_ord="W_REV_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=1;
    }
    test_ord="dp("+string(nvars(basering))+"),";
    if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
    {
      "Warning: block orderings are not supported; dp used for
      computation";
    }
  }

  if(external_ord=="" && find(singular_ord,"Dp")==1)
  {
    external_ord="W_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=1;
```

```
    }
    test_ord="Dp("+string(nvars(basering))+"),";
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported; Dp used for
      computation";
    }
  }
  if(external_ord=="" && find(singular_ord,"Dp")==3)
  {
    external_ord="W_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=1;
    }
    test_ord="Dp("+string(nvars(basering))+"),";
    if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
    {
      "Warning: block orderings are not supported; Dp used for
      computation";
    }
  }

  int pos;
  string number_string;

  if(external_ord=="" && find(singular_ord,"wp")==1)
  {
    external_ord="W_REV_LEX";
    pos=3;
    for(i=1;i<=nvars(basering);i++)
    {
      pos++;
      number_string="";
      while(singular_ord[pos]!=",")
      {
        number_string=number_string+singular_ord[pos];
        pos++;
      }
      execute("weightvec[i]="+number_string);
    }
    test_ord="wp("+string(weightvec)+"),";
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported;
      wp("+string(weightvec)+") used for computation";
    }
```

```
}
if(external_ord=="" && find(singular_ord,"wp")==3)
{
  external_ord="W_REV_LEX";
  pos=5;
  for(i=1;i<=nvars(basering);i++)
  {
    pos++;
    number_string="";
    while(singular_ord[pos]!=",")
    {
      number_string=number_string+singular_ord[pos];
      pos++;
    }
    execute("weightvec[i]="+number_string);
  }
  test_ord="wp("+string(weightvec)+"),";
  if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
  {
    "Warning: block orderings are not supported;
    wp("+string(weightvec)+") used for computation";
  }
}

if(external_ord=="" && find(singular_ord,"Wp")==1)
{
  external_ord="W_LEX";
  pos=3;
  for(i=1;i<=nvars(basering);i++)
  {
    pos++;
    number_string="";
    while(singular_ord[pos]!=",")
    {
      number_string=number_string+singular_ord[pos];
      pos++;
    }
    execute("weightvec[i]="+number_string);
  }
  test_ord="Wp("+string(weightvec)+"),";
  if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
  {
    "Warning: block orderings are not supported;
    Wp("+string(weightvec)+") used for computation";
  }
}
if(external_ord=="" && find(singular_ord,"Wp")==3)
```

```
{
  external_ord="W_LEX";
  pos=5;
  for(i=1;i<=nvars(basering);i++)
  {
    pos++;
    number_string="";
    while(singular_ord[pos]!=",")
    {
      number_string=number_string+singular_ord[pos];
      pos++;
    }
    execute("weightvec[i]="+number_string);
  }
  test_ord="Wp("+string(weightvec)+"),";
  if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
  {
    "Warning: block orderings are not supported;
    Wp("+string(weightvec)+") used for computation";
  }
}

if(external_ord=="")
{
  "ERROR: term ordering of actual basering not supported";
  return(I);
}

// check algorithm
if(alg=="ct" || alg=="pct")
  // these algorithms will not cause an error in the external
  // program; however, they do not compute the toric ideal of A, but
  // of an extended matrix
{
  "ERROR: algorithm not suitable";
  return(I);
}

// create temporary file with which the external program is called
int dummy;
int process=system("pid");
string matrixfile="temp_MATRIX"+string(process);
link MATRIX=":w "+matrixfile;
open(MATRIX);
write(MATRIX,"MATRIX","columns:",ncols(A),"cost vector:");
for(j=1;j<=ncols(A);j++)
{
```

```
    write(MATRIX,weightvec[j]);
  }
  write(MATRIX,"rows:",nrows(A),"matrix:");
  for(i=1;i<=nrows(A);i++)
  {
    for(j=1;j<=ncols(A);j++)
    {
      write(MATRIX,A[i,j]);
    }
  }

  // search for positive row space vector, if required by the
  // algorithm
  int found=0;
  if((alg=="blr") || (alg=="hs"))
  {
    for(i=1;i<=nrows(A);i++)
    {
      found=i;
      for(j=1;j<=ncols(A);j++)
      {
        if(A[i,j]<=0)
        {
          found=0;
        }
      }
      if(found>0)
      {
        break;
      }
    }
    if(found==0)
    {
      "ERROR: algorithm needs positive vector in the row space of the
      matrix";
      close(MATRIX);
      dummy=system("sh","rm -f "+matrixfile);
      return(I);
    }
    write(MATRIX,"positive row space vector:");
    for(j=1;j<=ncols(A);j++)
    {
      write(MATRIX,A[found,j]);
    }
  }
  close(MATRIX);
```

```
// call external program
dummy=system("sh","toric_ideal -alg "+alg+" "+matrixfile);

// read toric ideal from created file
link TORIC_IDEAL=":r "+matrixfile+".GB."+alg;
string toric_ideal=read(TORIC_IDEAL);
int generators;

pos=find(toric_ideal,"size");
pos=find(toric_ideal,":",pos);
pos++;

while(toric_ideal[pos]==" " || toric_ideal[pos]==newline)
{
  pos++;
}
number_string="";
while(toric_ideal[pos]!=" " && toric_ideal[pos]!=newline)
{
  number_string=number_string+toric_ideal[pos];
  pos++;
}
execute("generators="+number_string+";");

intvec v;
poly head;
poly tail;

pos=find(toric_ideal,"basis");
pos=find(toric_ideal,":",pos);
pos++;

for(i=1;i<=generators;i++)
{
  head=1;
  tail=1;

  for(j=1;j<=ncols(A);j++)
  {
    while(toric_ideal[pos]==" " || toric_ideal[pos]==newline)
    {
      pos++;
    }
    number_string="";
    while(toric_ideal[pos]!=" " && toric_ideal[pos]!=newline)
    {
      number_string=number_string+toric_ideal[pos];
```

```
        pos++;
      }
      execute("v[j]="+number_string+";");
      if(v[j]<0)
      {
        tail=tail*var(j)^(-v[j]);
      }
      if(v[j]>0)
      {
        head=head*var(j)^v[j];
      }
    }
    I[i]=head-tail;
  }

  // delete all created files
  dummy=system("sh","rm -f "+matrixfile);
  dummy=system("sh","rm -f "+matrixfile+".GB."+alg);

  return(I);
}

static proc toric_ideal_2(intmat A, string alg, intvec prsv)
{
  ideal I;
  // to be returned

  // check arguments
  if(size(prsv)<ncols(A))
  {
    "ERROR: number of matrix columns must equal size of positive row
    space vector";
    return(I);
  }

  // check suitability of actual basering
  if(nvars(basering)!=ncols(A))
  {
    "ERROR: number of matrix columns must be greater or equal number
    of ring variables";
    return(I);
  }

  // check suitability of actual term ordering
  // the "module ordering" c or C is ignored
  string singular_ord=ordstr(basering);
  string test_ord;
```

```
string external_ord="";
int i,j;
intvec weightvec;

if(find(singular_ord,"lp")==1)
{
  external_ord="W_LEX";
  for(i=1;i<=nvars(basering);i++)
  {
    weightvec[i]=0;
  }
  test_ord="lp("+string(nvars(basering))+"),";
  if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
  {
    "Warning: block orderings are not supported; lp used for
    computation";
  }
}
if(external_ord=="" && find(singular_ord,"lp")==3)
{
  external_ord="W_LEX";
  for(i=1;i<=nvars(basering);i++)
  {
    weightvec[i]=0;
  }
  test_ord="lp("+string(nvars(basering))+"),";
  if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
  {
    "Warning: block orderings are not supported; lp used for
    computation";
  }
}

if(external_ord=="" && find(singular_ord,"dp")==1)
{
  external_ord="W_REV_LEX";
  for(i=1;i<=nvars(basering);i++)
  {
    weightvec[i]=1;
  }
  test_ord="dp("+string(nvars(basering))+"),";
  if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
  {
    "Warning: block orderings are not supported; dp used for
    computation";
  }
}
```

```
if(external_ord=="" && find(singular_ord,"dp")==3)
{
  external_ord="W_REV_LEX";
  for(i=1;i<=nvars(basering);i++)
  {
    weightvec[i]=1;
  }
  test_ord="dp("+string(nvars(basering))+"),";
  if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
  {
    "Warning: block orderings are not supported; dp used for
    computation";
  }
}

if(external_ord=="" && find(singular_ord,"Dp")==1)
{
  external_ord="W_LEX";
  for(i=1;i<=nvars(basering);i++)
  {
    weightvec[i]=1;
  }
  test_ord="Dp("+string(nvars(basering))+"),";
  if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
  {
    "Warning: block orderings are not supported; Dp used for
    computation";
  }
}
if(external_ord=="" && find(singular_ord,"Dp")==3)
{
  external_ord="W_LEX";
  for(i=1;i<=nvars(basering);i++)
  {
    weightvec[i]=1;
  }
  test_ord="Dp("+string(nvars(basering))+"),";
  if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
  {
    "Warning: block orderings are not supported; Dp used for
    computation";
  }
}

int pos;
string number_string;
```

```
if(external_ord=="" && find(singular_ord,"wp")==1)
{
  external_ord="W_REV_LEX";
  pos=3;
  for(i=1;i<=nvars(basering);i++)
  {
    pos++;
    number_string="";
    while(singular_ord[pos]!=",")
    {
      number_string=number_string+singular_ord[pos];
      pos++;
    }
    execute("weightvec[i]="+number_string);
  }
  test_ord="wp("+string(weightvec)+"),";
  if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
  {
    "Warning: block orderings are not supported;
    wp("+string(weightvec)+") used for computation";
  }
}
if(external_ord=="" && find(singular_ord,"wp")==3)
{
  external_ord="W_REV_LEX";
  pos=5;
  for(i=1;i<=nvars(basering);i++)
  {
    pos++;
    number_string="";
    while(singular_ord[pos]!=",")
    {
      number_string=number_string+singular_ord[pos];
      pos++;
    }
    execute("weightvec[i]="+number_string);
  }
  test_ord="wp("+string(weightvec)+"),";
  if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
  {
    "Warning: block orderings are not supported;
    wp("+string(weightvec)+") used for computation";
  }
}

if(external_ord=="" && find(singular_ord,"Wp")==1)
{
```

```
    external_ord="W_LEX";
    pos=3;
    for(i=1;i<=nvars(basering);i++)
    {
      pos++;
      number_string="";
      while(singular_ord[pos]!=",")
      {
        number_string=number_string+singular_ord[pos];
        pos++;
      }
      execute("weightvec[i]="+number_string);
    }
    test_ord="Wp("+string(weightvec)+"),";
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported;
      Wp("+string(weightvec)+") used for computation";
    }
  }
  if(external_ord=="" && find(singular_ord,"Wp")==3)
  {
    external_ord="W_LEX";
    pos=5;
    for(i=1;i<=nvars(basering);i++)
    {
      pos++;
      number_string="";
      while(singular_ord[pos]!=",")
      {
        number_string=number_string+singular_ord[pos];
        pos++;
      }
      execute("weightvec[i]="+number_string);
    }
    test_ord="Wp("+string(weightvec)+"),";
    if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
    {
      "Warning: block orderings are not supported;
      Wp("+string(weightvec)+") used for computation";
    }
  }

  if(external_ord=="")
  {
    "ERROR: term ordering of actual basering not supported";
    return(I);
```

```
}

// check algorithm
if(alg=="ct" || alg=="pct")
  // these algorithms will not cause an error in the external
  // program; however, they do not compute the toric ideal of A, but
  // of an extended matrix
{
  "ERROR: algorithm not suitable";
  return(I);
}

// create temporary file with that the external program is called
int dummy;
int process=system("pid");
string matrixfile="temp_MATRIX"+string(process);
link MATRIX=":w "+matrixfile;
open(MATRIX);
write(MATRIX,"MATRIX","columns:",ncols(A),"cost vector:");
for(j=1;j<=ncols(A);j++)
{
  write(MATRIX,weightvec[j]);
}
write(MATRIX,"rows:",nrows(A),"matrix:");
for(i=1;i<=nrows(A);i++)
{
  for(j=1;j<=ncols(A);j++)
  {
    write(MATRIX,A[i,j]);
  }
}

// enter positive row space vector, if required by the algorithm
if((alg=="blr") || (alg=="hs"))
{
  write(MATRIX,"positive row space vector:");
  for(j=1;j<=ncols(A);j++)
  {
    write(MATRIX,prsv[j]);
  }
}
close(MATRIX);

// call external program
dummy=system("sh","toric_ideal -alg "+alg+" "+matrixfile);

// read toric ideal from created file
```

```
link TORIC_IDEAL=":r "+matrixfile+".GB."+alg;
string toric_ideal=read(TORIC_IDEAL);

int generators;
pos=find(toric_ideal,"size");
pos=find(toric_ideal,":",pos);
pos++;

while(toric_ideal[pos]==" " || toric_ideal[pos]==newline)
{
  pos++;
}
number_string="";
while(toric_ideal[pos]!=" " && toric_ideal[pos]!=newline)
{
  number_string=number_string+toric_ideal[pos];
  pos++;
}
execute("generators="+number_string+";");

intvec v;
poly head;
poly tail;

pos=find(toric_ideal,"basis");
pos=find(toric_ideal,":",pos);
pos++;

for(i=1;i<=generators;i++)
{
  head=1;
  tail=1;

  for(j=1;j<=ncols(A);j++)
  {
    while(toric_ideal[pos]==" " || toric_ideal[pos]==newline)
    {
      pos++;
    }
    number_string="";
    while(toric_ideal[pos]!=" " && toric_ideal[pos]!=newline)
    {
      number_string=number_string+toric_ideal[pos];
      pos++;
    }
    execute("v[j]="+number_string+";");
    if(v[j]<0)
```

```
      {
        tail=tail*var(j)^(-v[j]);
      }
      if(v[j]>0)
      {
        head=head*var(j)^v[j];
      }
    }
    I[i]=head-tail;
  }

  // delete all created files
  dummy=system("sh","rm -f "+matrixfile);
  dummy=system("sh","rm -f "+matrixfile+".GB."+alg);

  return(I);
}

proc toric_ideal
"USAGE:
toric_ideal(A,alg);            A intmat, alg string
toric_ideal(A,alg,prsv);       A intmat, alg string, prsv intvec
RETURN:  toric ideal of A as explained in toric.lib
         return type = ideal
EXAMPLE: example toric_ideal;  shows an example"
{
  if(size(#)==2)
  {
    return(toric_ideal_1(#[1],#[2]));
  }
  else
  {
    return(toric_ideal_2(#[1],#[2],#[3]));
  }
}

example
{
"EXAMPLE"; echo=2;

ring r=0,(x,y,z),dp;

// call with two arguments
intmat A[2][3]=1,1,0,0,1,1;
A;

ideal I=toric_ideal(A,"du");
```

```
I;

I=toric_ideal(A,"blr");
I;

// call with three arguments
intvec prsv=1,2,1;
I=toric_ideal(A,"blr",prsv);
I;
}

proc toric_std(ideal I)
"USAGE:   toric_std(I);        I ideal
RETURN:  standard basis of I as explained in toric.lib
         return type = ideal
EXAMPLE: example toric_std;   shows an example"
{
  ideal J;
  // to be returned

  // check suitability of actual term ordering
  // the "module ordering" c or C is ignored
  string singular_ord=ordstr(basering);
  string test_ord;
  string external_ord="";
  int i,j;
  intvec weightvec;

  if(find(singular_ord,"lp")==1)
  {
    external_ord="W_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=0;
    }
    test_ord="lp("+string(nvars(basering))+"),";
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported; lp used for
      computation";
    }
  }
  if(external_ord=="" && find(singular_ord,"lp")==3)
  {
    external_ord="W_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
```

```
      weightvec[i]=0;
    }
    test_ord="lp("+string(nvars(basering))+"),";
    if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
    {
      "Warning: block orderings are not supported; lp used for
      computation";
    }
  }

  if(external_ord=="" && find(singular_ord,"dp")==1)
  {
    external_ord="W_REV_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=1;
    }
    test_ord="dp("+string(nvars(basering))+"),";
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported; dp used for
      computation";
    }
  }
  if(external_ord=="" && find(singular_ord,"dp")==3)
  {
    external_ord="W_REV_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=1;
    }
    test_ord="dp("+string(nvars(basering))+"),";
    if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
    {
      "Warning: block orderings are not supported; dp used for
      computation";
    }
  }

  if(external_ord=="" && find(singular_ord,"Dp")==1)
  {
    external_ord="W_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=1;
    }
    test_ord="Dp("+string(nvars(basering))+"),";
```

```
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported; Dp used for
      computation";
    }
  }
  if(external_ord=="" && find(singular_ord,"Dp")==3)
  {
    external_ord="W_LEX";
    for(i=1;i<=nvars(basering);i++)
    {
      weightvec[i]=1;
    }
    test_ord="Dp("+string(nvars(basering))+"),";
    if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
    {
      "Warning: block orderings are not supported; Dp used for
      computation";
    }
  }

  int pos;
  string number_string;

  if(external_ord=="" && find(singular_ord,"wp")==1)
  {
    external_ord="W_REV_LEX";
    pos=3;
    for(i=1;i<=nvars(basering);i++)
    {
      pos++;
      number_string="";
      while(singular_ord[pos]!="," && singular_ord[pos]!=")")
      {
        number_string=number_string+singular_ord[pos];
        pos++;
      }
      execute("weightvec["+string(i)+"]="+number_string+";");
    }
    test_ord="wp("+string(weightvec)+"),";
    if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
    {
      "Warning: block orderings are not supported;
      wp("+string(weightvec)+") used for computation";
    }
  }
  if(external_ord=="" && find(singular_ord,"wp")==3)
```

```
{
  external_ord="W_REV_LEX";
  pos=5;
  for(i=1;i<=nvars(basering);i++)
  {
    pos++;
    number_string="";
    while(singular_ord[pos]!=",")
    {
      number_string=number_string+singular_ord[pos];
      pos++;
    }
    execute("weightvec[i]="+number_string);
  }
  test_ord="wp("+string(weightvec)+"),";
  if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
  {
    "Warning: block orderings are not supported;
    wp("+string(weightvec)+") used for computation";
  }
}

if(external_ord=="" && find(singular_ord,"Wp")==1)
{
  external_ord="W_LEX";
  pos=3;
  for(i=1;i<=nvars(basering);i++)
  {
    pos++;
    number_string="";
    while(singular_ord[pos]!=",")
    {
      number_string=number_string+singular_ord[pos];
      pos++;
    }
    execute("weightvec[i]="+number_string);
  }
  test_ord="Wp("+string(weightvec)+"),";
  if(singular_ord!=(test_ord+"C") && singular_ord!=(test_ord+"c"))
  {
    "Warning: block orderings are not supported;
    Wp("+string(weightvec)+") used for computation";
  }
}
if(external_ord=="" && find(singular_ord,"Wp")==3)
{
  external_ord="W_LEX";
```

```
  pos=5;
  for(i=1;i<=nvars(basering);i++)
  {
    pos++;
    number_string="";
    while(singular_ord[pos]!=",")
    {
      number_string=number_string+singular_ord[pos];
      pos++;
    }
    execute("weightvec[i]="+number_string);
  }
  test_ord="Wp("+string(weightvec)+"),";
  if(singular_ord!=("C"+test_ord) && singular_ord!=("c"+test_ord))
  {
    "Warning: block orderings are not supported;
    Wp("+string(weightvec)+") used for computation";
  }
}

if(external_ord=="")
{
  "ERROR: term ordering of actual basering not supported";
  return(I);
}

// create first temporary file with which the external program is
// called
int dummy;
int process=system("pid");
string groebnerfile="temp_GROEBNER"+string(process);
link GROEBNER=":w "+groebnerfile;
open(GROEBNER);
write(GROEBNER,"GROEBNER","computed with algorithm:","pt",
               "term ordering:","elimination block",0,
               "weighted block",nvars(basering),external_ord);
// algorithm is totally unimportant, only required by the external
// program
for(i=1;i<=nvars(basering);i++)
{
  write(GROEBNER,weightvec[i]);
}
write(GROEBNER,"size:",size(I),"Groebner basis:");

poly head;
poly tail;
poly rest;
```

```
intvec v;

for(j=1;j<=size(I);j++)
{
  // test suitability of generator j
  rest=I[j];
  head=lead(rest);
  rest=rest-head;
  tail=lead(rest);
  rest=rest-tail;

  if(head==0 && tail==0 && rest!=0)
  {
    "ERROR: generator "+string(j)+" of input ideal is no binomial";
    close(GROEBNER);
    dummy=system("sh","rm -f "+groebnerfile);
    return(J);
  }

  if(leadcoef(tail)!=-leadcoef(head))
    // generator is no difference of monomials (or a constant multiple)
  {
    "ERROR: generator "+string(j)+" of input ideal is no difference
    of monomials";
    close(GROEBNER);
    dummy=system("sh","rm -f "+groebnerfile);
    return(J);
  }
  if(gcd(head,tail)!=1)
  {
    "Warning: monomials of generator "+string(j)+" of input ideal
    are not relatively prime";
  }

  // write vector representation of generator j into the file
  v=leadexp(head)-leadexp(tail);
  for(i=1;i<=nvars(basering);i++)
  {
    write(GROEBNER,v[i]);
  }
}
close(GROEBNER);

// create second temporary file
string newcostfile="temp_NEW_COST"+string(process);
link NEW_COST=":w "+newcostfile;
open(NEW_COST);
```

```
write(NEW_COST,"NEW_COST","variables:",nvars(basering),
               "cost vector:");
for(i=1;i<=nvars(basering);i++)
{
  write(NEW_COST,weightvec[i]);
}

// call external program
dummy=system("sh","change_cost "+groebnerfile+" "+newcostfile);

// read toric standard basis from created file
link TORIC_IDEAL=":r "+newcostfile+".GB.pt";
string toric_ideal=read(TORIC_IDEAL);

int generators;
pos=find(toric_ideal,"size");
pos=find(toric_ideal,":",pos);
pos++;

while(toric_ideal[pos]==" " || toric_ideal[pos]==newline)
{
  pos++;
}
number_string="";
while(toric_ideal[pos]!=" " && toric_ideal[pos]!=newline)
{
  number_string=number_string+toric_ideal[pos];
  pos++;
}
execute("generators="+number_string+";");

pos=find(toric_ideal,"basis");
pos=find(toric_ideal,":",pos);
pos++;

for(j=1;j<=generators;j++)
{
  head=1;
  tail=1;

  for(i=1;i<=nvars(basering);i++)
  {
    while(toric_ideal[pos]==" " || toric_ideal[pos]==newline)
    {
      pos++;
    }
    number_string="";
```

```
      while(toric_ideal[pos]!=" " && toric_ideal[pos]!=newline)
      {
        number_string=number_string+toric_ideal[pos];
        pos++;
      }
      execute("v[i]="+number_string+";");
      if(v[i]<0)
      {
        tail=tail*var(i)^(-v[i]);
      }
      if(v[i]>0)
      {
        head=head*var(i)^v[i];
      }
    }
    J[j]=head-tail;
  }

  // delete all created files
  dummy=system("sh","rm -f "+groebnerfile);
  dummy=system("sh","rm -f "+groebnerfile+".GB.pt");
  dummy=system("sh","rm -f "+newcostfile);

  return(J);
}

example
{
"EXAMPLE"; echo=2;

ring r=0,(x,y,z),wp(3,2,1);

// call with toric ideal (of the matrix A=(1,1,1))
ideal I=x-y,x-z;
ideal J=toric_std(I);
J;

// call with the same ideal, but badly chosen generators:
// not only binomials
I=x-y,2x-y-z;
J=toric_std(I);
// binomials whose monomials are not relatively prime
I=x-y,xy-yz,y-z;
J=toric_std(I);
J;

// call with a non-toric ideal that seems to be toric
```

```
I=x-yz,xy-z;
J=toric_std(I);
J;
// comparison with real standard basis and saturation
ideal H=std(I);
H;
LIB "elim.lib";
sat(H,xyz);
}
```

A.5.2 IP.lib

```
// $Id$
//
// author : Christine Theis
//

//version="$Id$";

///////////////////////////////////////////////////////////////////

info="
LIBRARY: IP.lib PROCEDURES FOR INTEGER PROGRAMMING USING
                        GROEBNER BASIS METHODS

Let A an integral (mxn)-matrix, b a vector with m integral
coefficients and c a vector with n nonnegative integral
coefficients. The solution of the IP-problem

(*)    minimize{ cx | Ax=b,
                     all components of x are nonnegative integers }

proceeds in two steps:

1. We compute the toric ideal of A and its Groebner basis with respect
   to a term ordering refining the cost function c (such an ordering
   exists because c is nonnegative).

2. We reduce the right-hand vector b or an initial solution of the
   problem modulo this ideal.

For these purposes, we can use seven different algorithms:
The algorithm of Conti/Traverso (ct) can compute an optimal solution
of the IP-problem directly from the right-hand vector b. The same is
true for its `positive' variant (pct) which can only be applied if A
and b have nonnegative entries.
All other algorithms need initial solutions of the IP-problem. Except
```

```
from the Conti-Traverso algorithm with elimination (ect),
they are in general more efficient than the algorithm mentionned
before. These are the algorithms of Pottier (pt), Bigatti/La
Scala/Robbiano (blr), Hosten/Sturmfels (hs) and Di Biase/Urbanke (du).

solve_IP(intmat A, intvec bx, intvec c, string alg);
solve_IP(intmat A, list bx, intvec c, string alg);
solve_IP(intmat A, intvec bx, intvec c, string alg, intvec prsv);
solve_IP(intmat A, list bx, intvec c, string alg, intvec prsv);

procedures for solving the IP-problem (*)
    They return the solution(s) of the given problem(s) or the
    message 'not solvable'.
    'alg' may be one of the algorithm abbreviations as above.
    If 'alg' is chosen to be 'ct' or 'pct', bx is read as the right
    hand vector b of the system Ax=b. b should then be an intvec of
    size m where m is the number of rows of A. Furthermore, bx and
    A should be nonnegative if 'pct' is used.
If 'alg' is chosen to be 'ect','pt','blr','hs' or 'du', bx is
read as an initial solution x of the system Ax=b. bx should
then be a nonnegative intvec of size n where n is the number
of columns of A.
    If 'alg' is chosen to be 'blr' or 'hs', the algorithm needs a
    vector with positive coefficcients in the row space of A. If
    no row of A contains only positive entries, one must use the
    versions of solve_IP which take such a vector prsv as
    argument.
solve_IP may also be called with a list bx of intvecs instead
of a single intvec.
";

///////////////////////////////////////////////////////////////////////

static proc solve_IP_1(intmat A, intvec bx, intvec c, string alg)
{
  intvec v;
  // to be returned

  // check arguments as far as necessary
  // other inconsistencies are detected by the external program
  if(size(c)!=ncols(A))
  {
    "ERROR: number of matrix columns must equal size of cost vector";
    return(v);
  }

  // create first temporary file with which the external program is
```

```
// called
int process=system("pid");
string matrixfile="temp_MATRIX"+string(process);
link MATRIX=":w "+matrixfile;
open(MATRIX);
write(MATRIX,"MATRIX","columns:",ncols(A),"cost vector:");
int i,j;
for(j=1;j<=ncols(A);j++)
{
  write(MATRIX,c[j]);
}
write(MATRIX,"rows:",nrows(A),"matrix:");
for(i=1;i<=nrows(A);i++)
{
  for(j=1;j<=ncols(A);j++)
  {
    write(MATRIX,A[i,j]);
  }
}

// search for positive row space vector, if required by the
// algorithm
int found=0;
if((alg=="blr") || (alg=="hs"))
{
  for(i=1;i<=nrows(A);i++)
  {
    found=i;
    for(j=1;j<=ncols(A);j++)
    {
      if(A[i,j]<=0)
      {
        found=0;
      }
    }
    if(found>0)
    {
      break;
    }
  }
  if(found==0)
  {
    "ERROR: algorithm needs positive vector in the row space of the
    matrix";
    close(MATRIX);
    system("sh","rm -f "+matrixfile);
    return(v);
```

```
    }
    write(MATRIX,"positive row space vector:");
    for(j=1;j<=ncols(A);j++)
    {
      write(MATRIX,A[found,j]);
    }
  }
  close(MATRIX);

  // create second temporary file for the external program
  string problemfile="temp_PROBLEM"+string(process);
  link PROBLEM=":w "+problemfile;
  open(PROBLEM);
  write(PROBLEM,"PROBLEM","vector size:",size(bx),
                "number of instances:",1,
                "right-hand or initial solution vectors:");
  for(i=1;i<=size(bx);i++)
  {
    write(PROBLEM,bx[i]);
  }
  close(PROBLEM);

  // call external program
  int dummy=system("sh","solve_IP -alg "+alg+" "+matrixfile+" "
                      +problemfile);

  // read solution from created file
  link SOLUTION=":r "+matrixfile+".sol."+alg;
  string solution=read(SOLUTION);
  int pos;
  string s;
  if(alg=="ct" || alg=="pct")
  {
    pos=find(solution,"NO");
    if(pos!=0)
    {
      "not solvable";
    }
    else
    {
      pos=find(solution,"YES");
      pos=find(solution,":",pos);
      pos++;
      for(j=1;j<=ncols(A);j++)
      {
        while(solution[pos]==" " || solution[pos]==newline)
        {
```

```
          pos++;
        }
        s="";
        while(solution[pos]!=" " && solution[pos]!=newline)
        {
          s=s+solution[pos];
          pos++;
        }
        execute("v[j]="+s+";");
      }
    }
  }
  else
  {
    pos=find(solution,"optimal");
    pos=find(solution,":",pos);
    pos++;
    for(j=1;j<=ncols(A);j++)
    {
      while(solution[pos]==" " || solution[pos]==newline)
      {
        pos++;
      }
      s="";
      while(solution[pos]!=" " && solution[pos]!=newline)
      {
        s=s+solution[pos];
        pos++;
      }
      execute("v[j]="+s+";");
    }
  }

  // delete all created files
  dummy=system("sh","rm -f "+matrixfile);
  dummy=system("sh","rm -f "+matrixfile+".GB."+alg);
  dummy=system("sh","rm -f "+problemfile);
  dummy=system("sh","rm -f "+matrixfile+".sol."+alg);

  return(v);
}

static proc solve_IP_2(intmat A, list bx, intvec c, string alg)
{
  list l;;
  // to be returned
```

```
// check arguments as far as necessary
// other inconsistencies are detected by the external program
if(size(c)!=ncols(A))
{
  "ERROR: number of matrix columns must equal size of cost vector";
  return(l);
}
int k;
for(k=2;k<=size(bx);k++)
{
  if(size(bx[k])!=size(bx[1]))
  {
    "ERROR: size of all right-hand vectors must be equal";
    return(l);
  }
}

// create first temporary file with which the external program is
// called
int process=system("pid");
string matrixfile="temp_MATRIX"+string(process);
link MATRIX=":w "+matrixfile;
open(MATRIX);
write(MATRIX,"MATRIX","columns:",ncols(A),"cost vector:");
int i,j;
for(j=1;j<=ncols(A);j++)
{
  write(MATRIX,c[j]);
}
write(MATRIX,"rows:",nrows(A),"matrix:");
for(i=1;i<=nrows(A);i++)
{
  for(j=1;j<=ncols(A);j++)
  {
    write(MATRIX,A[i,j]);
  }
}

// search for positive row space vector, if required by the
// algorithm
int found=0;
if((alg=="blr") || (alg=="hs"))
{
  for(i=1;i<=nrows(A);i++)
  {
    found=i;
    for(j=1;j<=ncols(A);j++)
```

```
    {
      if(A[i,j]<=0)
      {
        found=0;
      }
    }
    if(found>0)
    {
      break;
    }
  }
  if(found==0)
  {
    "ERROR: algorithm needs positive vector in the row space of the
    matrix";
    close(MATRIX);
    system("sh","rm -f "+matrixfile);
    return(1);
  }
  write(MATRIX,"positive row space vector:");
  for(j=1;j<=ncols(A);j++)
  {
    write(MATRIX,A[found,j]);
  }
}
close(MATRIX);

// create second temporary file for the external program
string problemfile="temp_PROBLEM"+string(process);
link PROBLEM=":w "+problemfile;
open(PROBLEM);
write(PROBLEM,"PROBLEM","vector size:",size(bx[1]),
              "number of instances:",size(bx),
              "right-hand or initial solution vectors:");
for(k=1;k<=size(bx);k++)
{
  for(i=1;i<=size(bx[1]);i++)
  {
    write(PROBLEM,bx[k][i]);
  }
}
close(PROBLEM);

// call external program
int dummy=system("sh","solve_IP -alg "+alg+" "+matrixfile+" "
                      +problemfile);
```

```
// read solution from created file
link SOLUTION=":r "+matrixfile+".sol."+alg;
string solution=read(SOLUTION);
intvec v;
int pos,pos1,pos2;
string s;
if(alg=="ct" || alg=="pct")
{
  pos=1;
  for(k=1;k<=size(bx);k++)
  {
    pos1=find(solution,"NO",pos);
    pos2=find(solution,"YES",pos);
    if(pos1!=0 && (pos1<pos2 || pos2==0))
      // first problem not solvable
    {
      pos=find(solution,":",pos1);
      l=insert(l,"not solvable",size(l));
    }
    else
      // first problem solvable
    {
      pos=find(solution,":",pos2);
      pos++;
      for(j=1;j<=ncols(A);j++)
      {
        while(solution[pos]==" " || solution[pos]==newline)
        {
          pos++;
        }
        s="";
        while(solution[pos]!=" " && solution[pos]!=newline)
        {
          s=s+solution[pos];
          pos++;
        }
        execute("v[j]="+s+";");
      }
      l=insert(l,v,size(l));
    }
  }
}
else
{
  pos=1;
  for(k=1;k<=size(bx);k++)
  {
```

```
      pos=find(solution,"optimal",pos);
      pos=find(solution,":",pos);
      pos++;
      for(j=1;j<=ncols(A);j++)
      {
        while(solution[pos]==" " || solution[pos]==newline)
        {
          pos++;
        }
        s="";
        while(solution[pos]!=" " && solution[pos]!=newline)
        {
          s=s+solution[pos];
          pos++;
        }
        execute("v[j]="+s+";");
      }
      l=insert(l,v,size(l));
    }
  }

  // delete all created files
  dummy=system("sh","rm -f "+matrixfile);
  dummy=system("sh","rm -f "+matrixfile+".GB."+alg);
  dummy=system("sh","rm -f "+problemfile);
  dummy=system("sh","rm -f "+matrixfile+".sol."+alg);

  return(l);
}

static proc solve_IP_3(intmat A, intvec bx, intvec c, string alg,
                       intvec prsv)
{
  intvec v;
  // to be returned

  // check arguments as far as necessary
  // other inconsistencies are detected by the external program
  if(size(c)!=ncols(A))
  {
    "ERROR: number of matrix columns must equal size of cost vector";
    return(v);
  }
  if(size(prsv)!=ncols(A))
  {
    "ERROR: number of matrix columns must equal size of positive row
    space vector";
```

```
  return(v);
}

// create first temporary file with which the external program is
// called
int process=system("pid");
string matrixfile="temp_MATRIX"+string(process);
link MATRIX=":w "+matrixfile;
open(MATRIX);
write(MATRIX,"MATRIX","columns:",ncols(A),"cost vector:");
int i,j;
for(j=1;j<=ncols(A);j++)
{
  write(MATRIX,c[j]);
}
write(MATRIX,"rows:",nrows(A),"matrix:");
for(i=1;i<=nrows(A);i++)
{
  for(j=1;j<=ncols(A);j++)
  {
    write(MATRIX,A[i,j]);
  }
}

// enter positive row space vector, if required by the algorithm
if((alg=="blr") || (alg=="hs"))
{
  write(MATRIX,"positive row space vector:");
  for(j=1;j<=ncols(A);j++)
  {
    write(MATRIX,prsv[j]);
  }
}
close(MATRIX);

// create second temporary file for the external program
string problemfile="temp_PROBLEM"+string(process);
link PROBLEM=":w "+problemfile;
open(PROBLEM);
write(PROBLEM,"PROBLEM","vector size:",size(bx),
              "number of instances:",1,
              "right-hand or initial solution vectors:");
for(i=1;i<=size(bx);i++)
{
  write(PROBLEM,bx[i]);
}
close(PROBLEM);
```

```
// call external program
int dummy=system("sh","solve_IP -alg "+alg+" "+matrixfile+" "
                      +problemfile);

// read solution from created file
link SOLUTION=":r "+matrixfile+".sol."+alg;
string solution=read(SOLUTION);
int pos;
string s;
if(alg=="ct" || alg=="pct")
{
  pos=find(solution,"NO");
  if(pos!=0)
  {
    "not solvable";
  }
  else
  {
    pos=find(solution,"YES");
    pos=find(solution,":",pos);
    pos++;
    for(j=1;j<=ncols(A);j++)
    {
      while(solution[pos]==" " || solution[pos]==newline)
      {
        pos++;
      }
      s="";
      while(solution[pos]!=" " && solution[pos]!=newline)
      {
        s=s+solution[pos];
        pos++;
      }
      execute("v[j]="+s+";");
    }
  }
}
else
{
  pos=find(solution,"optimal");
  pos=find(solution,":",pos);
  pos++;
  for(j=1;j<=ncols(A);j++)
  {
    while(solution[pos]==" " || solution[pos]==newline)
    {
```

```
        pos++;
      }
      s="";
      while(solution[pos]!=" " && solution[pos]!=newline)
      {
        s=s+solution[pos];
        pos++;
      }
      execute("v[j]="+s+";");
    }
  }

  // delete all created files
  dummy=system("sh","rm -f "+matrixfile);
  dummy=system("sh","rm -f "+matrixfile+".GB."+alg);
  dummy=system("sh","rm -f "+problemfile);
  dummy=system("sh","rm -f "+matrixfile+".sol."+alg);

  return(v);
}



static proc solve_IP_4(intmat A, list bx, intvec c, string alg,
                       intvec prsv)
{
  list l;
  // to be returned

  // check arguments as far as necessary
  // other inconsistencies are detected by the external program
  if(size(c)!=ncols(A))
  {
    "ERROR: number of matrix columns must equal size of cost vector";
    return(l);
  }
  if(size(prsv)!=ncols(A))
  {
    "ERROR: number of matrix columns must equal size of positive row
    space vector";
    return(v);
  }
  int k;
  for(k=2;k<=size(bx);k++)
  {
    if(size(bx[k])!=size(bx[1]))
    {
```

```
      "ERROR: size of all right-hand vectors must be equal";
      return(1);
    }
  }

  // create first temporary file with which the external program is
  // called
  int process=system("pid");
  string matrixfile="temp_MATRIX"+string(process);
  link MATRIX=":w "+matrixfile;
  open(MATRIX);
  write(MATRIX,"MATRIX","columns:",ncols(A),"cost vector:");
  int i,j;
  for(j=1;j<=ncols(A);j++)
  {
    write(MATRIX,c[j]);
  }
  write(MATRIX,"rows:",nrows(A),"matrix:");
  for(i=1;i<=nrows(A);i++)
  {
    for(j=1;j<=ncols(A);j++)
    {
      write(MATRIX,A[i,j]);
    }
  }

  // enter positive row space vector if required by the algorithm
  if((alg=="blr") || (alg=="hs"))
  {
    write(MATRIX,"positive row space vector:");
    for(j=1;j<=ncols(A);j++)
    {
      write(MATRIX,prsv[j]);
    }
  }
  close(MATRIX);

  // create second temporary file for the external program
  string problemfile="temp_PROBLEM"+string(process);
  link PROBLEM=":w "+problemfile;
  open(PROBLEM);
  write(PROBLEM,"PROBLEM","vector size:",size(bx[1]),
                "number of instances:",size(bx),
                "right-hand or initial solution vectors:");
  for(k=1;k<=size(bx);k++)
  {
    for(i=1;i<=size(bx[1]);i++)
```

```
    {
      write(PROBLEM,bx[k][i]);
    }
  }
  close(PROBLEM);

  // call external program
  int dummy=system("sh","solve_IP -alg "+alg+" "+matrixfile+" "
                      +problemfile);

  // read solution from created file
  link SOLUTION=":r "+matrixfile+".sol."+alg;
  string solution=read(SOLUTION);
  intvec v;
  int pos,pos1,pos2;
  string s;
  if(alg=="ct" || alg=="pct")
  {
    pos=1;
    for(k=1;k<=size(bx);k++)
    {
      pos1=find(solution,"NO",pos);
      pos2=find(solution,"YES",pos);
      if(pos1!=0 && (pos1<pos2 || pos2==0))
        // first problem not solvable
      {
        pos=find(solution,":",pos1);
        l=insert(l,"not solvable",size(l));
      }
      else
        // first problem solvable
      {
        pos=find(solution,":",pos2);
        pos++;
        for(j=1;j<=ncols(A);j++)
        {
          while(solution[pos]==" " || solution[pos]==newline)
          {
            pos++;
          }
          s="";
          while(solution[pos]!=" " && solution[pos]!=newline)
          {
            s=s+solution[pos];
            pos++;
          }
          execute("v[j]="+s+";");
```

```
        }
        l=insert(l,v,size(l));
      }
    }
  }
  else
  {
    pos=1;
    for(k=1;k<=size(bx);k++)
    {
      pos=find(solution,"optimal",pos);
      pos=find(solution,":",pos);
      pos++;
      for(j=1;j<=ncols(A);j++)
      {
        while(solution[pos]==" " || solution[pos]==newline)
        {
          pos++;
        }
        s="";
        while(solution[pos]!=" " && solution[pos]!=newline)
        {
          s=s+solution[pos];
          pos++;
        }
        execute("v[j]="+s+";");
      }
      l=insert(l,v,size(l));
    }
  }

  // delete all created files
  dummy=system("sh","rm -f "+matrixfile);
  dummy=system("sh","rm -f "+matrixfile+".GB."+alg);
  dummy=system("sh","rm -f "+problemfile);
  dummy=system("sh","rm -f "+matrixfile+".sol."+alg);

  return(l);
}

proc solve_IP
"USAGE:
solve_IP(A,bx,c,alg);       A intmat, bx intvec, c intvec, alg string
solve_IP(A,bx,c,alg);       A intmat, bx list of intvec, c intvec,
                            alg string
solve_IP(A,bx,c,alg,prsv);  A intmat, bx intvec, c intvec, alg string,
                            prsv intvec
```

```
solve_IP(A,bx,c,alg,prsv);  A intmat, bx list of intvec, c intvec,
                            alg string, prsv intvec
RETURN:  solution of the associated integer programming problem as
         explained in IP.lib
         return type = type of bx
EXAMPLE: example solve_IP;  shows an example"
{
  if(size(#)==4)
  {
    if(typeof(#[2])=="intvec")
    {
      return(solve_IP_1(#[1],#[2],#[3],#[4]));
    }
    else
    {
      return(solve_IP_2(#[1],#[2],#[3],#[4]));
    }
  }
  else
  {
    if(typeof(#[2])=="intvec")
    {
      return(solve_IP_3(#[1],#[2],#[3],#[4],#[5]));
    }
    else
    {
      return(solve_IP_4(#[1],#[2],#[3],#[4],#[5]));
    }
  }
}

example
{
"EXAMPLE"; echo=2;

// call with single right-hand vector
intmat A[2][3]=1,1,0,0,1,1;
A;
intvec b1=1,1;
b1;
intvec c=2,2,1;
c;
intvec solution_vector=solve_IP(A,b1,c,"pct");
solution_vector;

// call with list of right-hand vectors
intvec b2=-1,1;
```

```
list l=b1,b2;
l;
list solution_list=solve_IP(A,l,c,"ct");
solution_list;

// call with single initial solution vector
A=2,1,-1,-1,1,2;
A;
b1=3,4,5;
solution_vector=solve_IP(A,b1,c,"du");

// call with single initial solution vector and algorithm needing a
// positive row space vector
solution_vector=solve_IP(A,b1,c,"hs");

// call with single initial solution vector and positive row space
// vector
intvec prsv=1,2,1;
prsv;
solution_vector=solve_IP(A,b1,c,"hs",prsv);
solution_vector;

// call with list of initial solution vectors and positive row space
// vector
b2=7,8,0;
l=b1,b2;
l;
solution_list=solve_IP(A,l,c,"blr",prsv);
solution_list;
}
```

Anhang B

Testmaterial

B.1 Matrizen und Kostenfunktionen

Dieser Abschnitt listet die Eingabedateien auf, die den Tests zur Berechnung torischer Ideale in Kapitel 6 zugrundeliegen. Es sind alles Dateien des Typs, der im Quellcode mit `MATRIX` bezeichnet wird; als Überschrift der einzelnen Abschnitte haben wir den Dateinamen gewählt.

Zur Namensgebung: Die Datei mit Namen `matrix_3x9_-5_5` enthält die in Kapitel 6 mit $r(3 \times 9, -5, 5)$ bezeichnete Instanz, also eine 3×9-Matrix mit zufälligen Koeffizienten im Bereich von -5 bis 5; die Datei `trans_5x7_10` die Instanz $t(5, 7)$, also die Matrix des 5×7-Transportproblems mit Kostenvektorkoeffizienten zwischen 0 und 10.

B.1.1 matrix_1x11_0_1

```
MATRIX

columns:
11

cost vector:
   1   0   1   1   1   1   0   0   1   1   0
rows:
1

matrix:
   1   0   1   1   0   0   0   0   0   1   0

positive row space vector:
```

B.1.2 matrix_1x11_0_5

```
MATRIX

columns:
```

```
11

cost vector:
   3   4   2   4   5   3   5   2   3   0   4
rows:
1

matrix:
   2   1   2   2   1   3   5   4   5   4   3

positive row space vector:
   2   1   2   2   1   3   5   4   5   4   3
```

B.1.3 matrix_1x11_-5_5

```
MATRIX

columns:
11

cost vector:
   2   4   1   1   5   2   2   4   4   4   0
rows:
1

matrix:
  -5   5  -5  -1  -5   3   5   2   2  -4   2

positive row space vector:
```

B.1.4 matrix_1x11_0_15

```
MATRIX

columns:
11

cost vector:
  10   4   0   6   0   3   5  12  11   2   7
rows:
1

matrix:
   4   4  12  11   9   0  10   6   3  14   5

positive row space vector:
```

B.1.5 matrix_1x11_-15_15

```
MATRIX

columns:
11

cost vector:
   8   5   8   4   6   2   4   8   9   8   7
rows:
1

matrix:
 -12 -12  -5  -7 -14  -3   1  12  12   0 -11

positive row space vector:
```

B.1.6 matrix_3x9_0_1

```
MATRIX

columns:
9

cost vector:
   1   1   0   0   0   1   1   1   1
rows:
3

matrix:
   0   0   0   1   1   1   0   1   0
   1   1   1   1   0   1   0   0   1
   0   1   0   1   0   0   1   0   0

positive row space vector:
   1   2   1   3   1   2   1   1   1
```

B.1.7 matrix_3x9_0_5

```
MATRIX

columns:
9

cost vector:
   3   1   3   5   3   5   3   1   3
rows:
3
```

```
matrix:
   4   5   3   0   5   4   5   5   2
   4   3   2   5   3   5   0   0   4
   5   4   3   0   5   2   1   2   3

positive row space vector:
   8   8   5   5   8   9   5   5   6
```

B.1.8 matrix_3x9_-5_5

```
MATRIX

columns:
9

cost vector:
   4   5   4   0   5   5   2   4   3
rows:
3

matrix:
   3   5  -1   3  -2   1  -3  -4   5
  -2  -5   5  -5  -4  -4  -4   2  -5
   3  -2  -4  -3  -1   3  -1   5   4

positive row space vector:
```

B.1.9 matrix_3x9_0_15

```
MATRIX

columns:
9

cost vector:
   1  12   9   7   3   7   9   0   3
rows:
3

matrix:
   3   4   3   9   4   7  14   0   2
   0   8   6   5   4   1  14   4  12
   4   8  10   9   9   6   3   0  10

positive row space vector:
   3  12   9  14   8   8  28   4  14
```

B.1.10 matrix_3x9_-15_15

```
MATRIX

columns:
9

cost vector:
  12   5   9  14   7   9   3   1  13
rows:
3

matrix:
  -1   3   9  14  13 -12  11   7  -4
  -4   9  -3  -3 -14   5  -3  -2 -10
  -9  -6   3  -5  -7  -8  12 -15 -11

positive row space vector:
```

B.1.11 matrix_5x7_0_1

```
MATRIX

columns:
7

cost vector:
   0   1   1   1   0   1   0
rows:
5

matrix:
   1   0   1   1   1   0   1
   0   1   0   1   0   0   1
   0   1   0   0   0   0   0
   1   1   0   1   0   0   0
   0   1   0   0   0   0   1

positive row space vector:
```

B.1.12 matrix_5x7_0_5

```
MATRIX

columns:
7

cost vector:
```

```
   1   5   5   4   2   2   1
rows:
5

matrix:
   2   1   4   2   4   0   4
   5   2   3   3   5   1   3
   4   4   5   5   4   5   5
   0   1   2   1   5   1   3
   1   2   2   3   3   0   3

positive row space vector:
   5   2   3   3   5   1   3
```

B.1.13 matrix_5x7_-5_5

```
MATRIX

columns:
7

cost vector:
   4   0   0   5   2   4   0
rows:
5

matrix:
  -3  -1  -5  -1  -2   3   0
  -2   2   0  -3   4  -1   5
   2  -4  -5   4  -4  -1  -1
   0  -1  -3  -4  -5  -2  -4
   2  -1   1   4   1  -1  -5

positive row space vector:
```

B.1.14 matrix_5x7_0_15

```
MATRIX

columns:
7

cost vector:
  10   9  10  14  12  14   1
rows:
5
```

```
matrix:
   5   3   8   4   3  10   4
  11   1   9   0   2   7   4
  14  11  12   9   5   5  15
   8   5   9   2  14   3   0
  10   2   2   0   5  10   4

positive row space vector:
   5   3   8   4   3  10   4
```

B.1.15 matrix_5x7_-15_15

```
MATRIX

columns:
7

cost vector:
  12   3   4   9  15  10  15
rows:
5

matrix:
   4   9 -15  -1   0  -6   8
  -9   6 -10  -8  -7   0   3
  -2   6  13 -14 -15 -12  -9
  11 -12  -4  -3  -3   4   1
 -11  -1  -4   8 -10  -6 -11

positive row space vector:
```

B.1.16 trans_3x4_10

```
MATRIX

columns:
12

cost vector:
  10   8   5   8   4   9   0   1   4   7   9  10
rows:
7

matrix:
 1 1 1 0 0 0 0 0 0 0 0 0
 0 0 0 1 1 1 0 0 0 0 0 0
 0 0 0 0 0 0 1 1 1 0 0 0
```

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

positive row space vector:
 1 1 1 1 1 1 1 1 1 1 1 1
```

B.1.17 trans_5x7_10

```
MATRIX

columns:
35

cost vector:
   8   7   9   4   5   8   0   1   1   4   0   3   4  10   8  10   1
   3   3   9   9   6   7   3   4   5   2   6  10   0   3   5   6   0
   9
rows:
12

matrix:
 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0
 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0
 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0
 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0
 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1

positive row space vector:
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

B.1.18 trans_5x7_50

```
MATRIX

columns:
35

cost vector:
```

```
  50   6  39  35  34   6  23  35  29  41  30  12   4  10  34  16  13
  29  33   0  40  43  11  15  37  35  48  11  12  46  26  11  26  39
  46
rows:
12

matrix:
 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0
 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0
 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0
 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0
 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1

positive row space vector:
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

B.2 Torische Ideale

Als Beispiel führen wir die drei Gröbnerbasen auf, die das Programm `toric_ideal` zur Instanz $r(1 \times 11, 0, 5)$ berechnet, wenn es mit dem allgemeinen Conti-Traverso-Algorithmus (`matrix_1x11_0_5.GB.ct`), dem nichtnegativen Conti-Traverso-Algorithmus (`matrix_1x11_0_5.GB.pct`) und einem der restlichen Algorithmen (`matrix_1x11_0_5.GB.pt`) augerufen wird (die Ausgaben dieser restlichen Algorithmen stimmen bis auf Laufzeitangaben überein).

B.2.1 matrix_1x11_0_5.GB.ct

```
GROEBNER

computed with algorithm:
ct
from file(s):
matrix_1x11_0_5
computation time
  0.01 sec

term ordering:
elimination block
2
LEX
```

```
weighted block
11
W_LEX
  3.00  4.00  2.00  4.00  5.00  3.00  5.00  2.00  3.00  0.00  4.00
size:
23

Groebner basis:
    0    0    0    0    0   -1    0    0    0    0    1    0    0
    0    0   -1    0    0    0    0    0    2   -2    0    0    0
    0    0    0    0    0   -1    0    0    0    1    0    0    1
    0    0    0    0    0    0    0    0    1   -1    0    0    1
    0   -1    0    0    0    0    0    0    0    0    0    1    0
    1    0   -1    0    0    0    0    0    0    0    0    0    0
    0    1    0    0    0    0    0    0   -1    1    0    0    0
    0    1   -1    0    0    0    0    0    1   -1    0    0    0
    0    2   -1    0    0    0    0    0    0    0    0    0    0
    0    1    0    0    0    0    0    0    0    0    0    0    1
    0    0    2    0    0    0    0    0    0   -1    0    0    0
    0    0    1    0    0   -1    0    0    1   -1    0    0    0
    0   -1    1    0    0    0    0    0    0    0    0    0    1
    0    1    1    0    0   -1    0    0    0    0    0    0    0
    0    0   -1    1    0    0    0    0    0    0    0    0    0
    0   -1    0    0    1    0    0    0    0    0    0    0    0
    0    0    0    0    0    1    0    0    1   -2    0    0    0
    0    0   -1    0    0    2    0    0    0   -1    0    0    0
    0    0   -1    0    0    1    0    0    0    0    0    0    1
    0    1    0    0    0    1    0    0    0   -1    0    0    0
    0    0    1    0    0    1    0    0   -1    0    0    0    0
    0    0    0    0    0    0    1    0   -1    0    0    0    0
    0    0    0    0    0    0    0    1    0   -1    0    0    0

settings for the Buchberger algorithm:
version:
1
S-pair criteria:
relatively prime leading terms
criterion M
criterion B

interreduction frequency:
12.00 %

compiler settings:
SUPPORT_DRIVEN_METHODS_EXTENDED
List Support Variables:
8
```

```
SUPPORT_VARIABLES_FIRST
simply linked lists
```

B.2.2 matrix_1x11_0_5.GB.pct

```
GROEBNER

computed with algorithm:
pct
from file(s):
matrix_1x11_0_5
computation time
  0.00 sec

term ordering:
elimination block
1
LEX
weighted block
11
W_LEX
  3.00  4.00  2.00  4.00  5.00  3.00  5.00  2.00  3.00  0.00  4.00

size:
18

Groebner basis:
    0    0    0    0    0   -1    0    0    0    0    1    0
    0    0   -1    0    0    0    0    0    2   -2    0    0
    0   -1    0    0    0    0    0    0    0    0    0    1
    1    0   -1    0    0    0    0    0    0    0    0    0
    0    1    0    0    0    0    0    0   -1    1    0    0
    0    1   -1    0    0    0    0    0    1   -1    0    0
    0    2   -1    0    0    0    0    0    0    0    0    0
    0    0    2    0    0    0    0    0    0   -1    0    0
    0    0    1    0    0   -1    0    0    1   -1    0    0
    0    1    1    0    0   -1    0    0    0    0    0    0
    0    0   -1    1    0    0    0    0    0    0    0    0
    0   -1    0    0    1    0    0    0    0    0    0    0
    0    0    0    0    0    1    0    0    1   -2    0    0
    0    0   -1    0    0    2    0    0    0   -1    0    0
    0    1    0    0    0    1    0    0    0   -1    0    0
    0    0    1    0    0    1    0    0   -1    0    0    0
    0    0    0    0    0    0    1    0   -1    0    0    0
    0    0    0    0    0    0    0    1    0   -1    0    0

settings for the Buchberger algorithm:
```

```
version:
1
S-pair criteria:
relatively prime leading terms
criterion M
criterion B

interreduction frequency:
12.00 %

compiler settings:
SUPPORT_DRIVEN_METHODS_EXTENDED
List Support Variables:
8
SUPPORT_VARIABLES_FIRST
simply linked lists
```

B.2.3 matrix_1x11_0_5.GB.pt

```
GROEBNER

computed with algorithm:
pt
from file(s):
matrix_1x11_0_5
computation time
  0.00 sec

term ordering:
elimination block
0
weighted block
11
W_LEX
  3.00  4.00  2.00  4.00  5.00  3.00  5.00  2.00  3.00  0.00  4.00
size:
17

Groebner basis:
    0    0    0    0    0   -1    0    0    0    0    1
    0    0   -1    0    0    0    0    0    2   -2    0
    1    0   -1    0    0    0    0    0    0    0    0
    0    1    0    0    0    0    0    0   -1    1    0
    0    1   -1    0    0    0    0    0    1   -1    0
    0    2   -1    0    0    0    0    0    0    0    0
    0    0    2    0    0    0    0    0    0   -1    0
    0    0    1    0    0   -1    0    0    1   -1    0
```

```
0   1   1   0   0  -1   0   0   0   0   0
0   0  -1   1   0   0   0   0   0   0   0
0  -1   0   0   1   0   0   0   0   0   0
0   0   0   0   0   1   0   0   1  -2   0
0   0  -1   0   0   2   0   0   0  -1   0
0   1   0   0   0   1   0   0   0  -1   0
0   0   1   0   0   1   0   0  -1   0   0
0   0   0   0   0   0   1   0  -1   0   0
0   0   0   0   0   0   0   1   0  -1   0

settings for the Buchberger algorithm:
version:
1
S-pair criteria:
relatively prime leading terms
criterion M
criterion B

interreduction frequency:
12.00 %

compiler settings:
SUPPORT_DRIVEN_METHODS_EXTENDED
List Support Variables:
8
SUPPORT_VARIABLES_FIRST
simply linked lists
```

B.3 Ganzzahlige Optimierungsprobleme

Die folgenden Seiten enthalten die IP-Instanzen, die Abschnitt 6.5 zugrundeliegen. Zu jeder Matrix aus Abschnitt B.2 gehören zwei oder drei Dateien vom Typ `PROBLEM`, mit denen das Programm `solve_IP` aufgerufen wird. Die Dateien mit Endung `.ct` bzw. `.pct` enthalten fünf Begrenzungsvektoren für den allgemeinen bzw. nichtnegativen Conti-Traverso-Algorithmus, die mit Endung `.pt` fünf Initiallösungen für die anderen Algorithmen. Für die beiden 5×7-Transportprobleme $t(5,7)$ und $t(5,7)_{50}$ wurden diesselben Vektoren benutzt.

Zur Namensgebung: Die Datei `problems_1x11_0_1.ct` enthält die Instanzen zur Gröbnerbasis in `matrix_1x11_0_1.GB.ct` usw.

Als Beispiel geben wir zusätzlich die `.sol.`-Dateien zur Instanz $r(1\times 11, 0, 5)$ an, welche die Optimallösungen zu den Begrenzungsvektoren bzw. Initiallösungen der `PROBLEM`-Dateien enthalten.

B.3.1 problems_1x11_0_1.ct

```
PROBLEM
```

```
vector size:
1

number of instances:
5

right hand or initial solution vectors:
   192
   134
  -100
    95
    31
```

B.3.2 problems_1x11_0_5.ct

```
PROBLEM

vector size:
1

number of instances:
5

right hand or initial solution vectors:
 11704
  8016
  -443
  -392
   -87
```

B.3.3 problems_1x11_-5_5.ct

```
PROBLEM

vector size:
1

number of instances:
5

right hand or initial solution vectors:
   663
 -1630
  -443
  -392
   -87
```

B.3.4 problems_1x11_0_15.ct

```
PROBLEM

vector size:
1

number of instances:
5

right hand or initial solution vectors:
 49730
 47047
   968
  -301
  1304
```

B.3.5 problems_1x11_-15_15.ct

```
PROBLEM

vector size:
1

number of instances:
5

right hand or initial solution vectors:
-51418
 -3992
   968
  -301
  1304
```

B.3.6 problems_3x9_0_1.ct

```
PROBLEM

vector size:
3

number of instances:
5

right hand or initial solution vectors:
   215   316   139
   219   401   206
   -90     5   -47
```

```
   24   -67   -92
  -38    78    53
```

B.3.7 problems_3x9_0_5.ct

```
PROBLEM

vector size:
3

number of instances:
5

right hand or initial solution vectors:
 11192  9883  9364
  7816  5124  5941
    27   398  -272
   279  -275   319
  -358   349  -476
```

B.3.8 problems_3x9_-5_5.ct

```
PROBLEM

vector size:
3

number of instances:
5

right hand or initial solution vectors:
  4051 -7249   813
   895 -4288  2170
    27   398  -272
   279  -275   319
  -358   349  -476
```

B.3.9 problems_3x9_0_15.ct

```
PROBLEM

vector size:
3

number of instances:
5

right hand or initial solution vectors:
```

```
 39102 50543 70384
 23024 39218 47280
 -1130   872   790
  -603 -1178  -512
 -1388   210   131
```

B.3.10 problems_3x9_-15_15.ct

```
PROBLEM

vector size:
3

number of instances:
5

right hand or initial solution vectors:
 39549-32281-57411
 22079-19588-44824
 -1130   872   790
  -603 -1178  -512
 -1388   210   131
```

B.3.11 problems_5x7_0_1.ct

```
PROBLEM

vector size:
5

number of instances:
5

right hand or initial solution vectors:
   170   156    72   100   146
   219   127    77   158   109
   -26    87    81    98    83
    83    15    34    89    48
   -61    48   -71    99    -7
```

B.3.12 problems_5x7_0_5.ct

```
PROBLEM

vector size:
5

number of instances:
```

```
5

right hand or initial solution vectors:
  4620  6479  8787  3245  3864
  3809  5830  7841  3408  3146
   187   217   188  -193    94
   427   143  -483   138   323
   -18  -316    25   443  -339
```

B.3.13 problems_5x7_-5_5.ct

```
PROBLEM

vector size:
5

number of instances:
5

right hand or initial solution vectors:
 -3541   777 -2907 -4660  1263
 -1947  1560 -3159 -4257   377
   187   217   188  -193    94
   427   143  -483   138   323
   -18  -316    25   443  -339
```

B.3.14 problems_5x7_0_15.ct

```
PROBLEM

vector size:
5

number of instances:
5

right hand or initial solution vectors:
 38256 31415 75630 34737 31188
 38847 33835 71140 33634 31636
  -511  1069  -886  -116 -1180
  1400  1456 -1206  1065   989
   779   280   808  -819  -855
```

B.3.15 problems_5x7_-15_15.ct

```
PROBLEM

vector size:
```

```
5

number of instances:
5

right hand or initial solution vectors:
  4804-17875-33653-12201-27879
 -4157-23196-31775 -5029-27466
  -511  1069  -886  -116 -1180
  1400  1456 -1206  1065   989
   779   280   808  -819  -855
```

B.3.16 problems_trans_3x4.ct

```
PROBLEM

vector size:
7

number of instances:
5

right hand or initial solution vectors:
   143    43    61   108    80   166   109
   201   228   152   181   261   242   259
    -9   -26    -6  -100   -76   -52   -70
   -98     7   -15   -10   -29    16     4
   -80  -100   -74   -39   -28    -3    33
```

B.3.17 problems_trans_5x7.ct

```
PROBLEM

vector size:
12

number of instances:
5

right hand or initial solution vectors:
   203   195   236   188   235   251   335   195   335   518   353   242
   213   264   288   218   279   238   172   305   397   323   402   245
   -24   -85    11    15    11    35   -48    84    12    85   -25    36
   -22   -75    11    27   -45    13    84   -10    53   -95   -94   -94
   -75    57   -67   -64   -72    30    19   -45    -4   -70    20    57
```

B.3.18 problems_1x11_0_1.pct

```
PROBLEM

vector size:
1

number of instances:
5

right hand or initial solution vectors:
   192
   134
    72
    66
    63
```

B.3.19 problems_1x11_0_5.pct

```
PROBLEM

vector size:
1

number of instances:
5

right hand or initial solution vectors:
 11704
  8016
    74
    52
   147
```

B.3.20 problems_1x11_0_15.pct

```
PROBLEM

vector size:
1

number of instances:
5

right hand or initial solution vectors:
 49730
 47047
  1434
```

```
   33
 1220
```

B.3.21 problems_3x9_0_1.pct

```
PROBLEM

vector size:
3

number of instances:
5

right hand or initial solution vectors:
   215   316   139
   219   401   206
    13    11    30
     0    27    94
    66    25    64
```

B.3.22 problems_3x9_0_5.pct

```
PROBLEM

vector size:
3

number of instances:
5

right hand or initial solution vectors:
 11192  9883  9364
  7816  5124  5941
   100   471   434
   316   265   467
   350   335   408
```

B.3.23 problems_3x9_0_15.pct

```
PROBLEM

vector size:
3

number of instances:
5

right hand or initial solution vectors:
```

```
 39102 50543 70384
 23024 39218 47280
   963    59  1355
   741    55   434
  1124  1309  1196
```

B.3.24 problems_5x7_0_1.pct

```
PROBLEM

vector size:
5

number of instances:
5

right hand or initial solution vectors:
   170   156    72   100   146
   219   127    77   158   109
    59     4    72    37    76
    86    25    19     0    14
    24    15    73    55    93
```

B.3.25 problems_5x7_0_5.pct

```
PROBLEM

vector size:
5

number of instances:
5

right hand or initial solution vectors:
  4620  6479  8787  3245  3864
  3809  5830  7841  3408  3146
   345   109   360   215    21
   165   266   200    23   423
   231   438   450   281    30
```

B.3.26 problems_5x7_0_15.pct

```
PROBLEM

vector size:
5

number of instances:
```

```
5

right hand or initial solution vectors:
 38256 31415 75630 34737 31188
 38847 33835 71140 33634 31636
   147   960   818   973  1214
   727  1289  1342  1066  1411
  1003  1325   620   947   881
```

B.3.27 problems_trans_3x4.pct

```
PROBLEM

vector size:
7

number of instances:
5

right hand or initial solution vectors:
   143    43    61   108    80   166   109
   201   228   152   181   261   242   259
    46    39    12    68    41    28    31
     0    14    45    91    43    12    58
    17    53    26    41     0    19    92
```

B.3.28 problems_trans_5x7.pct

```
PROBLEM

vector size:
12

number of instances:
5

right hand or initial solution vectors:
   203   195   236   188   235   251   335   195   335   518   353   242
   213   264   288   218   279   238   172   305   397   323   402   245
    31    60    42     1    17    46    41    84    54     8    97    93
    20    64     0    14    61     0    28    72    57    71    50    81
    89    70     7    96    70    26    87     1    87    95    69    70
```

B.3.29 problems_1x11_0_1.pt

```
PROBLEM

vector size:
```

```
11

number of instances:
5

right hand or initial solution vectors:
    32    32    54    12    52    56     8    30    44    94    44
    39    65    19    51    91     1     5    89    34    25    58
    20    51    38    65    30     7    20    10    51    18    43
    71    97    61    26     5    57    70    65     0    75    29
    86    93    87    87    64    75    88    89   100     7    40
```

B.3.30 problems_1x11_0_5.pt

```
PROBLEM

vector size:
11

number of instances:
5

right hand or initial solution vectors:
   381   299   344   106   328   444   405   365   450   302   380
    61     3   385   230   450    24   195   333   497    81   341
   215   342   158   142   378   315   329     5   129   456   305
   473    61   132   417   212   243   112    13   370   174   264
   254   150   460   278   346    38    21   173   380   483   261
```

B.3.31 problems_1x11_-5_5.pt

```
PROBLEM

vector size:
11

number of instances:
5

right hand or initial solution vectors:
   381   299   344   106   328   444   405   365   450   302   380
    61     3   385   230   450    24   195   333   497    81   341
   215   342   158   142   378   315   329     5   129   456   305
   473    61   132   417   212   243   112    13   370   174   264
   254   150   460   278   346    38    21   173   380   483   261
```

B.3.32 problems_1x11_0_15.pt

```
PROBLEM

vector size:
11

number of instances:
5

right hand or initial solution vectors:
  1118   876  1230   289  1483   732   260   300   287   253   333
   353  1199   604   211    59  1245   447  1011   980  1212    59
  1269  1172   562   479   289   964  1439   149  1399  1056  1079
  1128  1399  1062   359   158  1362   647   466   195  1054   218
   853  1319   331   651   266  1342   131  1478  1456  1454  1149
```

B.3.33 problems_1x11_-15_15.pt

```
PROBLEM

vector size:
11

number of instances:
5

right hand or initial solution vectors:
  1118   876  1230   289  1483   732   260   300   287   253   333
   353  1199   604   211    59  1245   447  1011   980  1212    59
  1269  1172   562   479   289   964  1439   149  1399  1056  1079
  1128  1399  1062   359   158  1362   647   466   195  1054   218
   853  1319   331   651   266  1342   131  1478  1456  1454  1149
```

B.3.34 problems_3x9_0_1.pt

```
PROBLEM

vector size:
9

number of instances:
5

right hand or initial solution vectors:
    64    97    39    21    78    70    21    46    25
    54    76    92    84    47    57    46    31    38
    31    75    40    61    21    84    51    86    41
```

```
   19    21    37    58    86   100    97    73    44
   67    60    90    58    13    31    49    63    44
```

B.3.35 problems_3x9_0_5.pt

```
PROBLEM

vector size:
9

number of instances:
5

right hand or initial solution vectors:
  491   427   450   272   309   455   202   156   294
  478   239    59   266   213    32   183   460    62
  463   262   314    49   348   124   286   366    10
   59   126   194   133   363   120   329   134   175
   30    83   332   324    60   317   129    72    29
```

B.3.36 problems_3x9_-5_5.pt

```
PROBLEM

vector size:
9

number of instances:
5

right hand or initial solution vectors:
  491   427   450   272   309   455   202   156   294
  478   239    59   266   213    32   183   460    62
  463   262   314    49   348   124   286   366    10
   59   126   194   133   363   120   329   134   175
   30    83   332   324    60   317   129    72    29
```

B.3.37 problems_3x9_0_15.pt

```
PROBLEM

vector size:
9

number of instances:
5

right hand or initial solution vectors:
```

```
  1191  1236  1396  1077  1487   854   154   551  1311
   183  1147   901   166  1176   766    96   828  1140
   118   164   914   465   682   780  1250   598  1256
   259   996  1449   639   741  1238   589   371  1225
  1443   525   329  1253   762    30   654   929  1260
```

B.3.38 problems_3x9_-15_15.pt

```
PROBLEM

vector size:
9

number of instances:
5

right hand or initial solution vectors:
  1191  1236  1396  1077  1487   854   154   551  1311
   183  1147   901   166  1176   766    96   828  1140
   118   164   914   465   682   780  1250   598  1256
   259   996  1449   639   741  1238   589   371  1225
  1443   525   329  1253   762    30   654   929  1260
```

B.3.39 problems_5x7_0_1.pt

```
PROBLEM

vector size:
7

number of instances:
5

right hand or initial solution vectors:
    18    72    64    10     4    64    74
    63    77    15    18    91    84    32
    36    77    10    39    75    35    87
    23    22    30    37    31    65    58
    59     7    14    78    79    45    54
```

B.3.40 problems_5x7_0_5.pt

```
PROBLEM

vector size:
7

number of instances:
```

```
5

right hand or initial solution vectors:
   390   424   332   264   274   175   116
   348   369   133    90   407   345   101
     3   429   191   179   346   208    13
     3   284   140   475    57   255   272
   163   181    20   300   105   352    63
```

B.3.41 problems_5x7_-5_5.pt

```
PROBLEM

vector size:
7

number of instances:
5

right hand or initial solution vectors:
   390   424   332   264   274   175   116
   348   369   133    90   407   345   101
     3   429   191   179   346   208    13
     3   284   140   475    57   255   272
   163   181    20   300   105   352    63
```

B.3.42 problems_5x7_0_15.pt

```
PROBLEM

vector size:
7

number of instances:
5

right hand or initial solution vectors:
   770  1442   848  1368   542  1137  1207
   885   948  1044  1321   430  1252  1033
  1421   534   228   743    58   620  1235
   230  1085    60  1029   251    26   691
   375   332  1370  1145   274   717  1066
```

B.3.43 problems_5x7_-15_15.pt

```
PROBLEM

vector size:
```

```
7

number of instances:
5

right hand or initial solution vectors:
   770  1442   848  1368   542  1137  1207
   885   948  1044  1321   430  1252  1033
  1421   534   228   743    58   620  1235
   230  1085    60  1029   251    26   691
   375   332  1370  1145   274   717  1066
```

B.3.44 problems_trans_3x4.pt

```
PROBLEM

vector size:
12

number of instances:
5

right hand or initial solution vectors:
    25    89    29    19    18     6    28    26     7     8    33    67
    74    95    32    99    33    96     5    51    96    83    63    35
    62    71    39    16    10    69     8    35    23     3    55    41
    76    49    68    83    23    67    15    97    61    13    61    60
    75    33    77    71    15    39    72    43    76    77    59    53
```

B.3.45 problems_trans_5x7.pt

```
PROBLEM

vector size:
35

number of instances:
5

right hand or initial solution vectors:
    11    33    88    34    37     8    76    79    23     9    62
    46    76    43     9     2    57    70    28    31    69     4
    68    84    10    39    26    52    82    52     4    93    85
    59    94
    21    33    35    67    57    44    28    69    86    37    78
    54    94    14    48    25    83    18    59    33    28    99
    25    81    46    77    51    39    62     9    32    49    43
```

```
   33   15
  100   77    9   68   28   47   12   82    6   26   96
   98   75   13   57    7    8   55   33   55    0   76
    5    5    3   15    3   53   58   36   34   23   79
    10  57
    6   23   69   54   29   61   49   27   36   63   84
    9   71    4    8   25   71   85   97   77   88   11
   46    6   35   83    7   24   27   17   82   34   40
   16   88
   69   44    3   62   46   32   45   55    2   49   64
   94   87   14   90   63   68   68   75   75    2   23
   82   27   51   65   75   85   71   57   38   39    0
    7    1
```

B.3.46 matrix_1x11_0_5.sol.ct

```
SOLUTION

computed with algorithm:
ct
from file:
matrix_1x11_0_5.GB.ct
right hand vector:
 11704
solvable:
YES
optimal solution:
     0     0     0     0     0     0     0     0     0  2926     0
computation time
   0.000 sec

right hand vector:
  8016
solvable:
YES
optimal solution:
     0     0     0     0     0     0     0     0     0  2004     0
computation time
   0.000 sec

right hand vector:
  -443
solvable:
NO
computation time
   0.000 sec
```

```
right hand vector:
  -392
solvable:
NO
computation time
   0.000 sec

right hand vector:
   -87
solvable:
NO
computation time
   0.000 sec
```

B.3.47 matrix_1x11_0_5.sol.pct

```
SOLUTION

computed with algorithm:
pct
from file:
matrix_1x11_0_5.GB.pct
right hand vector:
 11704
solvable:
YES
optimal solution:
     0     0     0     0     0     0     0     0     0  2926     0
computation time
   0.000 sec

right hand vector:
  8016
solvable:
YES
optimal solution:
     0     0     0     0     0     0     0     0     0  2004     0
computation time
   0.000 sec

right hand vector:
    74
solvable:
YES
optimal solution:
     0     0     1     0     0     0     0     0     0    18     0
computation time
```

```
   0.000 sec

right hand vector:
    52
solvable:
YES
optimal solution:
     0     0     0     0     0     0     0     0     0    13     0
computation time
   0.000 sec

right hand vector:
   147
solvable:
YES
optimal solution:
     0     0     0     0     0     1     0     0     0    36     0
computation time
   0.000 sec
```

B.3.48 matrix_1x11_0_5.sol.pt

```
SOLUTION

computed with algorithm:
pt
from file:
matrix_1x11_0_5.GB.pt
initial solution vector:
   381   299   344   106   328   444   405   365   450   302   380
optimal solution:
     0     0     0     0     0     0     0     0     0  2926     0
computation time
   0.000 sec

initial solution vector:
    61     3   385   230   450    24   195   333   497    81   341
optimal solution:
     0     0     0     0     0     0     0     0     0  2004     0
computation time
   0.000 sec

initial solution vector:
   215   342   158   142   378   315   329     5   129   456   305
optimal solution:
     0     0     0     0     0     0     0     0     0  1936     0
computation time
```

```
   0.000 sec

initial solution vector:
   473    61   132   417   212   243   112    13   370   174   264
optimal solution:
     0     0     0     0     0     0     0     0     0  1749     0
computation time
   0.000 sec

initial solution vector:
   254   150   460   278   346    38    21   173   380   483   261
optimal solution:
     0     0     1     0     0     0     0     0     0  2001     0
computation time
   0.000 sec
```

B.4 Variation des Kostenvektors

In diesem Abschnitt findet man das Testmaterial zu Abschnitt 6.6. Wiederum für die Beispielinstanz $r(1 \times 11, 0, 5)$ geben wir die modifizierten Kostenfunktionen und die darauf umgerechneten Gröbnerbasen an. Die Dateien sind ähnlich wie in den Abschnitten B.1 und B.2 benannt: `matrix` ist im Dateinamen durch `new_cost` ersetzt.

B.4.1 new_cost_1x11_0_5

```
NEW_COST

variables:
11

cost vector:
   1   1   0   0   3   0   2   2   5   5   4
```

B.4.2 new_cost_1x11_0_5.GB.ct

```
GROEBNER

computed with algorithm:
ct
from file(s):
matrix_1x11_0_5.GB.ct, new_cost_1x11_0_5
computation time
    0.01 sec

term ordering:
elimination block
```

```
2
LEX
weighted block
11
W_LEX
  1.00  1.00  0.00  0.00  3.00  0.00  2.00  2.00  5.00  5.00  4.00
size:
16

Groebner basis:
    0    0    0    0    0   -1    0    0    0    0    1    0    0
    0    0    0   -2    0    0    0    0    0    1    0    0    0
    0    0    0   -1    0   -1    0    0    1    0    0    0    0
    0   -1    0    0    0    0    0    0    0    0    0    1    0
    1    0    0   -1    0    0    0    0    0    0    0    0    0
    0    2    0   -1    0    0    0    0    0    0    0    0    0
    0    1    0    0    0    0    0    0    0    0    0    0    1
    0    0    1   -1    0    0    0    0    0    0    0    0    0
    0    0    0    3    0   -2    0    0    0    0    0    0    0
    0   -1    0    1    0    0    0    0    0    0    0    0    1
    0    1    0    1    0   -1    0    0    0    0    0    0    0
    0   -1    0    0    1    0    0    0    0    0    0    0    0
    0    0    0   -1    0    1    0    0    0    0    0    0    1
    0    1    0   -2    0    1    0    0    0    0    0    0    0
    0    0    0   -1    0   -1    1    0    0    0    0    0    0
    0    0    0   -2    0    0    0    1    0    0    0    0    0
```

B.4.3 new_cost_1x11_0_5.GB.pct

```
GROEBNER

computed with algorithm:
pct
from file(s):
matrix_1x11_0_5.GB.pct, new_cost_1x11_0_5
computation time
    0.00 sec

term ordering:
elimination block
1
LEX
weighted block
11
W_LEX
  1.00  1.00  0.00  0.00  3.00  0.00  2.00  2.00  5.00  5.00  4.00
size:
```

```
13

Groebner basis:
    0    0    0    0    0   -1    0    0    0    0    1    0
    0    0    0   -2    0    0    0    0    0    1    0    0
    0    0    0   -1    0   -1    0    0    1    0    0    0
    0   -1    0    0    0    0    0    0    0    0    0    1
    1    0    0   -1    0    0    0    0    0    0    0    0
    0    2    0   -1    0    0    0    0    0    0    0    0
    0    0    1   -1    0    0    0    0    0    0    0    0
    0    0    0    3    0   -2    0    0    0    0    0    0
    0    1    0    1    0   -1    0    0    0    0    0    0
    0   -1    0    0    1    0    0    0    0    0    0    0
    0    1    0   -2    0    1    0    0    0    0    0    0
    0    0    0   -1    0   -1    1    0    0    0    0    0
    0    0    0   -2    0    0    0    1    0    0    0    0
```

B.4.4 new_cost_1x11_0_5.GB.pt

```
GROEBNER

computed with algorithm:
pt
from file(s):
matrix_1x11_0_5.GB.pt, new_cost_1x11_0_5
computation time
    0.00 sec

term ordering:
elimination block
0
weighted block
11
W_LEX
  1.00  1.00  0.00  0.00  3.00  0.00  2.00  2.00  5.00  5.00  4.00
size:
12

Groebner basis:
     0     0     0     0     0    -1     0     0     0     0     1
     0     0     0    -2     0     0     0     0     0     1     0
     0     0     0    -1     0    -1     0     0     1     0     0
     1     0     0    -1     0     0     0     0     0     0     0
     0     2     0    -1     0     0     0     0     0     0     0
     0     0     1    -1     0     0     0     0     0     0     0
     0     0     0     3     0    -2     0     0     0     0     0
     0     1     0     1     0    -1     0     0     0     0     0
```

0	-1	0	0	1	0	0	0	0	0	0
0	1	0	-2	0	1	0	0	0	0	0
0	0	0	-1	0	-1	1	0	0	0	0
0	0	0	-2	0	0	0	1	0	0	0

B.5 SINGULAR

Anhand der Testinstanz $r(1 \times 11, 0, 5)$ zeigen wir, wie sich die erweiterten torischen Ideale der Conti-Traverso-Algorithmen mit SINGULAR berechnen lassen. Die Wahl der monomialen Ordnung geschieht dort über die Ringdefinition.

Zum Beispiel an den Erzeugern mit den Nummern 16 und 5 in der Datei `singular_1x11_0_5.ct` erkennt man, daß SINGULAR im allgemeinen keine *reduzierte* Gröbnerbasis berechnet.

B.5.1 singular_1x11_0_5.ct

```
> ring r=0,(y,t,x(1..11)),(lp(2),Wp(3,4,2,4,5,3,5,2,3,0,4));
> ideal I;
> I[1]=y^2-x(1);
> I[2]=y-x(2);
> I[3]=y^2-x(3);
> I[4]=y^2-x(4);
> I[5]=y-x(5);
> I[6]=y^3-x(6);
> I[7]=y^5-x(7);
> I[8]=y^4-x(8);
> I[9]=y^5-x(9);
> I[10]=y^4-x(10);
> I[11]=y^3-x(11);
> I[12]=t*y-1;
> std(I);
_[1]=x(8)-x(10)
_[2]=x(1)-x(3)
_[3]=x(11)-x(6)
_[4]=x(4)-x(3)
_[5]=x(3)^2-x(10)
_[6]=x(2)*x(10)-x(9)
_[7]=x(7)-x(2)*x(10)
_[8]=x(5)-x(2)
_[9]=x(3)*x(9)-x(6)*x(10)
_[10]=x(3)*x(6)-x(2)*x(10)
_[11]=x(9)^2-x(3)*x(10)^2
_[12]=x(6)*x(9)-x(10)^2
_[13]=x(6)^2-x(3)*x(10)
_[14]=x(2)*x(3)-x(6)
_[15]=x(2)*x(9)-x(3)*x(10)
_[16]=x(2)*x(6)-x(3)^2
```

```
_[17]=x(2)^2-x(3)
_[18]=t*x(10)-x(11)
_[19]=t*x(3)-x(2)
_[20]=t*x(9)-x(10)
_[21]=t*x(6)-x(3)
_[22]=t*x(2)-1
_[23]=y-t*x(3)
```

B.5.2 singular_1x11_0_5.pct

```
> ring r=0,(y,x(1..11)),(lp(1),Wp(3,4,2,4,5,3,5,2,3,0,4));
> ideal I;
> I[1]=y^2-x(1);
> I[2]=y-x(2);
> I[3]=y^2-x(3);
> I[4]=y^2-x(4);
> I[5]=y-x(5);
> I[6]=y^3-x(6);
> I[7]=y^5-x(7);
> I[8]=y^4-x(8);
> I[9]=y^5-x(9);
> I[10]=y^4-x(10);
> I[11]=y^3-x(11);
> std(I);
_[1]=x(8)-x(10)
_[2]=x(1)-x(3)
_[3]=x(11)-x(6)
_[4]=x(4)-x(3)
_[5]=x(3)^2-x(10)
_[6]=x(2)*x(10)-x(9)
_[7]=x(7)-x(9)
_[8]=x(5)-x(2)
_[9]=x(3)*x(9)-x(6)*x(10)
_[10]=x(3)*x(6)-x(9)
_[11]=x(9)^2-x(3)*x(10)^2
_[12]=x(6)*x(9)-x(3)^2*x(10)
_[13]=x(6)^2-x(3)*x(10)
_[14]=x(2)*x(3)-x(6)
_[15]=x(2)*x(9)-x(6)^2
_[16]=x(2)*x(6)-x(3)^2
_[17]=x(2)^2-x(3)
_[18]=y-x(2)
```

B.6 MPL und CPLEX

Wir geben eine Möglichkeit an, die Testinstanz $r(1 \times 11, 0, 5)$ mit MPL zu modellieren, so daß sie mit CPLEX gelöst werden kann; als Begrenzungsvektor nehmen wir dabei den ersten in `problems_1x11_0_5.ct`. Außerdem drucken wir die von MPL generierte Lösungsdatei zu dieser Instanz ab; man vergleiche die Lösung mit der in `matrix_1x11_0_5.sol.ct`.

Dank an Maximal Software, Inc. für die Erlaubnis, die Dateien zu zitieren (die `.sol`-Datei wurde der Übersichtlichkeit wegen um einige Leerzeilen gekürzt).

B.6.1 matrix_1x11_0_5a.mpl

```
TITLE
    matrix_1x11_0_5a;

INDEX
    spalten = 1..11;

DECISION VARIABLES
    x[spalten];

MIN

    3 x[1] + 4 x[2] + 2  x[3] + 4  x[4] + 5 x[5] + 3 x[6] + 5 x[7] +
    2 x[8] + 3 x[9] + 0 x[10] + 4 x[11];

SUBJECT TO

    2 x[1] + 1 x[2] + 2  x[3] + 2  x[4] + 1 x[5] + 3 x[6] + 5 x[7] +
    4 x[8] + 5 x[9] + 4 x[10] + 3 x[11] = 11704;
    INTEGER x[1];
    INTEGER x[2];
    INTEGER x[3];
    INTEGER x[4];
    INTEGER x[5];
    INTEGER x[6];
    INTEGER x[7];
    INTEGER x[8];
    INTEGER x[9];
    INTEGER x[10];
    INTEGER x[11];

END
```

B.6.2 matrix_1x11_0_5a.sol

```
   MPL Modeling System   -   Copyright (c) 1988-1999, Maximal Software, Inc.
--------------------------------------------------------------------------------
```

```
MODEL STATISTICS

Problem name:       matrix_1x11_0_5a

Filename:
Date:               November 2, 1999
Time:               12:06
Parsing time:       0.05 sec

Solver:             CPLEX 300
Objective value:    0.000000000000
Iterations:         0
Integer nodes:      0
Solution time:      0.00 sec

Constraints:        1
Variables:          11
Integers:           11
Nonzeros:           11
Density:            100 %


SOLUTION RESULT

  Optimal integer solution found

    MIN Z       =         0.0000


DECISION VARIABLES

VARIABLE x[spalten] :

    spalten         Activity     Reduced Cost
  ---------------------------------------------
         1            0.0000           3.0000
         2            0.0000           4.0000
         3            0.0000           2.0000
         4            0.0000           4.0000
         5            0.0000           5.0000
         6            0.0000           3.0000
         7            0.0000           5.0000
         8            0.0000           2.0000
         9            0.0000           3.0000
        10         2926.0000           0.0000
        11            0.0000           4.0000
  ---------------------------------------------
```

```
CONSTRAINTS

PLAIN CONSTRAINTS

    Constraint Name             Slack     Shadow Price
  ------------------------------------------------------
    c1                         0.0000          0.0000
  ------------------------------------------------------

END
```

Literaturverzeichnis

[1] Bachmann, O., Schönemann, H. *Monomial representations for Gröbner bases computations.* Reports on Computer Algebra **18**, Zentrum für Computeralgebra, Universität Kaiserslautern, 1998.

[2] Bigatti, A.M. *Computation of Hilbert-Poincaré series.* Journal of Pure and Applied Algebra **199**, Seiten 237-253, 1997.

[3] Bigatti, A.M., La Scala, R., Robbiano, L. *Computing toric ideals.* Soll erscheinen im Journal of Symbolic Computation, Preprint erhältlich unter `http://cocoa.dima.unige.it/research/publications.html`, 1998.

[4] Buchberger, B. *A criterion for deleting unnecessary reductions in the construction of Gröbner bases.* Proceedings EUROSAM '79, Springer LNCS **72**, Seiten 3-21, 1979.

[5] Buchberger, B. *Gröbner bases - an algorithmic method in polynomial ideal theory.* Kapitel 6 in Bose, N.K. (Editor): *Multidimensional System Theory.* D. Reidel Publ., 1985.

[6] Cohen, H. *A Course in Computational Algebraic Number Theory.* Springer, Berlin - Heidelberg - New York, 1993.

[7] Conti, P., Traverso, C. *Buchberger algorithm and integer programming.* Proceedings AAECC-9 (New Orleans), Springer LNCS **539**, Seiten 130-139, 1991.

[8] Cormen, T.H.,Leiserson, C.E., Rivest, R.R. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[9] Cox, D., Little, J., O'Shea, D. *Ideals, Varieties and Algorithms.* Undergraduate Texts in Mathematics, Springer, New York, 1991.

[10] Dantzig, G.B., Thapa, M.N. *Linear Programming 1: Introduction.* Springer Series in Operations Research, 1997.

[11] Decker, W. *Vorlesungen zur Algebra II / Computeralgebra.* Universität des Saarlandes, Saarbrücken, Sommersemester 1997. Unveröffentlicht.

[12] Di Biase, F., Urbanke, R. *An algorithm to calculate the kernel of certain polynomial ring homomorphisms.* Experimental Mathematics **4**, Seiten 227-234, 1995.

[13] Gebauer, R., Möller, H.M. *On an installation of Buchberger's algorithm.* Journal of Symbolic Computation **6**, Seiten 275-286, Springer, New York, 1988.

[14] Grassmann, H., Greuel, G.M., Martin, B., Neumann, W.,Pfister, G., Pohl, W., Schönemann, H., Siebert, T. *On an implementation of standard bases and syzygies in SINGULAR.* Reports on Computer Algebra **1**, Zentrum für Computeralgebra, Universität Kaiserslautern, 1996.

[15] Hoşten, S., Shapiro, J. *Primary decomposition of lattice basis ideals.* Preprint, 1998.

[16] Hoşten, S., Sturmfels, B. *GRIN: An implementation of Gröbner bases for integer programming.* In Balas, E., Clausen, J. (Editoren): *Integer Programming and Combinatorial Optimization.* Springer LNCS **920**, Seiten 267-276, 1995.

[17] Hoşten, S., Thomas, R. *Gröbner bases and integer programming.* In Buchberger, B., Winkler, F. (Editoren): *Gröbner bases and Applications*, London Mathematical Society Lecture Notes Series **251**, Seiten 144-158, Cambridge University Press, 1998.

[18] Hoşten, S., Thomas, R. *Standard pairs and group relaxations in integer programming.* Journal of Pure and Applied Algebra **139**, Seiten 133-157, 1999.

[19] Kernighan, D.W., Ritchie, D.M. *The C-Programming Language.* 2. Ausgabe, Prentice Hall, Englewood Cliffs, N.J., 1988.

[20] Lang, S. *Algebra.* 3. Ausgabe, Addison-Wesley Publishing Company, 1993.

[21] LEDA: A Library of Efficient Data types and Algorithms. Informationen unter `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[22] Lewis, H.R., Papadimitriou, C.H. *Elements of the Theory of Computation.* Prentice Hall, New Jersey, 1981.

[23] Möller, H.M., Mora, T. *Upper and lower bounds for the degree of Gröbner bases.* EUROSAM '84, Springer LNCS **174**, Seiten 172-183, 1984.

[24] MPL Modeling System. Informationen unter `http://www.maximal-usa.com`.

[25] Pottier, L. *Gröbner bases of toric ideals.* Rapport de recherche 2224, INRIA Sophia Antipolis, 1994.

[26] Schrijver, A. *Theory of Linear and Integer Programming.* Wiley-Interscience Series in Discrete Mathematics and Optimization, New York, 1986.

[27] Sierksma, G. *Linear and Integer Programming.* Monographs and Textbooks in Pure and Applied Mathematics **198**, Marcel Dekker Inc., New York - Basel - Hongkong, 1996.

[28] SINGULAR: A Computer Algebra System for Polynomial Computations. Informationen unter
`http://www.mathematik.uni-kl.de/~zca/Singular/` .

[29] Stroustrup, S. *The C++ Programming Language.* 3. Ausgabe, Addison-Wesley, 1997.

[30] Sturmfels, B. *Gröbner Bases and Convex Polytopes.* University Lecture Series, Volume 8. American Mathematical Society, Providence, Rhode Island, 1996.

[31] Thomas, R. *Gröbner bases in integer programming* . Du, D.-Z., Pardalos, P.M. (Editoren): *Handbook of Combinatorial Optimization*, Kluwer Academic Publishers, Boston, MA, Seiten 533-572, 1998.

[32] Thomas, R., Weismantel, R. *Truncated Gröbner bases for integer programming.* Applicable Algebra in Engineering, Communication and Computing **8**, Seiten 241-257, 1997.

[33] Urbaniak, R., Weismantel, R., Ziegler, G.M. *A variant of the Buchberger algorithm for integer programming.* SIAM Journal of Discrete Mathematics **10** (1), Seiten 96-108, Februar 1997.

Index

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit selbständig angefertigt und nur die angegebenen Hilfsmittel benutzt habe.

Christine Theis