Interdisziplinäres Projekt

Berechnung von Kopplungen auf Orbifolds

Martin von Gagern

1.5. – 15.6.2007

Inhaltsverzeichnis

1 Aufgabenstellung

Ziel dieses interdisziplinären Projektes ist es, für ein gegebenes stringtheoretisches Modell herauszufinden, welche der modellierten Felder miteinander gekoppelt sind. Die Felder sind dabei charakterisiert durch einen Satz von rationalen Quantenzahlen. Die an einer Kopplung beteiligten Felder erfüllen bestimmte Symmetriebedingungen, die sich als Bedingungen an die Summen der entsprechenden Quantenzahlen ausdrücken lassen.

1.1 Problemklasse

Wenn man von der physikalischen Bedeutung weg abstrahiert, stellt sich die Aufgabe wie folgt dar: gegeben ein Menge von Vektoren, finde eine Multimenge von Elementen dieser Menge, deren Summe einen bestimmten Ergebnisvektor ergibt, wobei für einzelne Vektorkomponenten Moduloarithmetik gilt. Ignorieren wir die Modulo-Komponenten vorerst, so stellt der Rest ein lineares Gleichungssystem dar, dessen Koeffizienten *ganzzahlig* und *nicht negativ* sein müssen.

$$Ax = b \qquad x \geq 0 \qquad x \in \mathbb{Z}^n \tag{1}$$

Diese Darstellung des Problems hat deutliche Ähnlichkeiten mit der bekannten Problemstellung „Integer Programming“. Dabei handelt es sich um Optimierungsprobleme unter Berücksichtigung einer Reihe von beschränkenden Ungleichungen.

$$\max c^T x \qquad \text{so dass } Ax \leq b \qquad x \geq 0 \qquad x \in \mathbb{Z}^n \tag{2}$$

Selbst ohne die Optimierungsbedingung ist bereits die Fragestellung, ob überhaupt eine Lösung dieses Systems existiert, $\mathcal{NP}$-vollständig[5], gehört also zur Problemklasse $\mathcal{NP}$ der nichtdeterministisch polynomiellen Probleme. Für diese Probleme ist noch kein Algorithmus bekannt, der diese in Polynomialzeit lösen würde, also mit einem Zeitaufwand, der durch ein Polynom in der Eingabelänge beschränkt ist. Meist wird davon ausgegangen, dass ein solcher Algorithmus nicht existiert ($\mathcal{P} \neq \mathcal{NP}$). Da es möglich ist, Ungleichungen unter Hinzunahme weiterer Variablen in Gleichungen umzuformulieren, wäre ein Algorithmus zur Lösung von Gleichungssystemen auch zum Lösen von Ungleichungssystemen in der Lage. Einen Polynomialzeitalgorithmus zum Lösen der Gleichungssysteme zu finden ist also höchst unwahrscheinlich.

1.2 Formulierung als IP

Soweit die schlechte Nachricht. Die gute Nachricht ist, dass Optimierungsprobleme eine hohe Bedeutung in vielen Bereichen haben, und viel Forschung dazu betrieben wurde und wird. Um davon profitieren zu können, müssen wir das vorliegende Problem in die gleiche Form bringen.

	f_1	f_2	f_3	f_4	f_5	Ziel
q_1	1	0	0	1	0	$= 1$
q_2	0	1	0	1	0	$= 1$
q_3	0	0	1	1	0	$= 1$
q_4	1	1	1	2	1	$= 0 \pmod 2$
q_γ	0	0	0	$\frac{1}{2}$	1	siehe Abschnitt 1.2.3 auf Seite 4

Tabelle 1: Beispiel einer Probleminstanz

Ich werde das in Tabelle 1 dargestellte Beispiel verwenden, um die einzelnen Schritte zu verdeutlichen. Dieses Beispiel ist ohne phsyikalische Signifikanz, dafür einfach zum Rechnen.

Zunächst wird das Gleichungssystem in eine Matrixschreibweise gebracht, was die Darstellung der folgenden Umformungen etwas vereinheitlicht.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \pmod 2 \end{pmatrix} \qquad x_i \in \mathbb{N} \tag{3}$$

1.2.1 Modulo

Die Modulo-Operation addiert zu einer Zahl ein ganzzahliges Vielfaches des Modulo-Divisors.

$$a_i x = b_i \pmod m \quad \longrightarrow \quad a_i x + mk = b_i \quad k \in \mathbb{Z} \tag{4}$$

Dies kann durch Hinzunahme eines Vektors mit dem Eintrag m in der Zeile i als einzigem Eintrag ausgedrückt werden:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 2 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ k \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \qquad x_i \in \mathbb{N}, k \in \mathbb{Z} \tag{5}$$

Die neue Variable k darf auch negativ sein. Dies kann beispielsweise durch die Differenz zweier nichtnegativer Variablen ausgedrückt werden. Die meisten Systeme für lineare Optimierung erlauben jedoch auch einzelne Variablen ohne Nichtnegativitätsbeschränkung, so dass keine weiteren Schritte zur Behandlung verschiedener Vorzeichen von k erforderlich sind.

1.2.2 Gleichungen

Es ist prinzipiell möglich, jede Gleichung durch zwei Ungleichungen zu ersetzen.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 \\ 1 & 1 & 1 & 2 & 1 & 2 \\ -1 & -1 & -1 & -2 & -1 & -2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ k \end{pmatrix} \leq \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \qquad x_i \in \mathbb{N}, k \in \mathbb{Z} \tag{6}$$

Die meisten Programme zur Lösung linearer Optimierungsprobleme erlauben jedoch die Angabe sowol einer unteren als auch einer oberen Schranke für jede Zeile sowie jede Variable.

$$\begin{array}{rrrrrrrrr|rr} 1 \leq & 1\,x_1 & & & +1\,x_4 & & & \leq 1 & & & \\ 1 \leq & & 1\,x_2 & & +1\,x_4 & & & \leq 1 & & 0 \leq x_i \leq \infty & x_i \in \mathbb{Z} \\ 1 \leq & & & 1\,x_3 & +1\,x_4 & & & \leq 1 & & -\infty \leq\ k\ \leq \infty & k \in \mathbb{Z} \\ 0 \leq & 1\,x_1 & +1\,x_2 & +1\,x_3 & +2\,x_4 & +1\,x_5 & +2\,k & \leq 0 & & & \end{array} \tag{7}$$

1.2.3 Spezialfall Gamma

Eine Besondere Bedingung gilt für q_γ. In dieser Zeile darf nicht ein einzelner Bruch der einzige von Null verschiedene Wert sein. Lauter Nuller sind zulässig, ebenso eine einzelne Ganzzahl oder mehrere beliebige von Null verschiedene Werte. Auch diese zusätzliche Bedingung lässt sich in das Ungleichungssystem codieren. Dazu ordnen wir den q_γ in jedem Vektor jeweils eine Zahl n_γ zu, die zweischen Null, Bruch und Ganzzahl unterscheidet.

$$n_\gamma = \begin{cases} 0 & \text{für } q_\gamma = 0 \\ 1 & \text{für } q_\gamma \in \mathbb{Q} \setminus \mathbb{Z} \\ 2 & \text{für } q_\gamma \in \mathbb{Z} \setminus \{0\} \end{cases} \tag{8}$$

Jetzt gilt die Bedingung, dass die Summe dieser n_γ nicht genau 1 ergeben darf. Unter Zuhilfenahme einer neuen Entscheidungsvariable x_γ mit dem Wertebereich $\{0, 1\}$ kann dies durch die folgenden beiden Ungleichungen ausgedrückt werden:

$$-2x_\gamma + \sum_a n_\gamma^{(a)} \geq 0 \tag{9}$$

$$-\infty x_\gamma + \sum_a n_\gamma^{(a)} \leq 0 \tag{10}$$

Dabei drückt $x_\gamma = 0$ den Fall aus, dass alle $n_\gamma = 0$ sind, und $x_\gamma = 1$ den Fall, dass die Summe der n_γ mindestens 2 ist.

$$\begin{array}{rrrrrrr|lll}
1 \leq & 1\,x_1 & & & +1\,x_4 & & \leq 1 & & & \\
1 \leq & & 1\,x_2 & & +1\,x_4 & & \leq 1 & & & \\
1 \leq & & & 1\,x_3 & +1\,x_4 & & \leq 1 & & & \\
0 \leq & 1\,x_1 & +1\,x_2 & +1\,x_3 & +2\,x_4 & +1\,x_5 +2\,k & \leq 0 & 0 \leq x_i \leq \infty & x_i \in \mathbb{Z} & \\
0 \leq & & & & 1\,x_4 & +2\,x_5 \quad -2\,x_\gamma & \leq \infty & -\infty \leq k \leq \infty & k \in \mathbb{Z} & (11) \\
-\infty \leq & & & & 1\,x_4 & +2\,x_5 \quad -\infty\,x_\gamma & \leq 0 & 0 \leq x_\gamma \leq 1 & x_\gamma \in \mathbb{Z} &
\end{array}$$

1.2.4 Differenzvektoren

Viele der Vektoren in der Eingabe kommen als Multipletts vor. Das ist ein Satz von meist drei Vektoren, die in den meisten Quantenzahlen übereinstimmen, in einzelnen jedoch unterschiedliche Werte aufweisen. Physikalisch beschreiben sie das gleiche Feld, daher tragen sie auch in der Eingabedatei den gleichen Namen. Dann kann man einen der Vektoren unverändert beibehalten, die anderen beiden jedoch als Unterschiede zu diesem einen Repräsentanten darstellen. Nehmen wir an, die Felder f_1, f_2 und f_3 aus dem Beispiel seinen ein solches Triplett.

$$\begin{array}{rrrrrrr|lll}
1 \leq & 1\,x_1 & -1\,x_2 & -1\,x_3 & +1\,x_4 & & \leq 1 & & & \\
1 \leq & & 1\,x_2 & & +1\,x_4 & & \leq 1 & & & \\
1 \leq & & & 1\,x_3 & +1\,x_4 & & \leq 1 & & & \\
0 \leq & 1\,x_1 & & & +2\,x_4 & +1\,x_5 +2\,k & \leq 0 & 0 \leq x_i \leq \infty & x_i \in \mathbb{Z} & \\
0 \leq & & & & 1\,x_4 & +2\,x_5 \quad -2\,x_\gamma & \leq \infty & -\infty \leq k \leq \infty & k \in \mathbb{Z} & (12) \\
-\infty \leq & & & & 1\,x_4 & +2\,x_5 \quad -\infty\,x_\gamma & \leq 0 & 0 \leq x_\gamma \leq 1 & x_\gamma \in \mathbb{Z} & \\
0 \leq & 1\,x_1 & -1\,x_2 & -1\,x_3 & & & \leq \infty & & &
\end{array}$$

Die Variable x_1 zählt jetzt alle drei Felder, die zweite und dritte Spalte wurden durch die Differenz zur ersten ersetzt. Die neu hinzugekommene Zeile stellt sicher, dass nicht mehr Differenzvektoren als „echte“ Vektoren in der Lösung enthalten sind. Somit ist für jeden Differenzvektor auch ein passender voller Vektor vorhanden, mit dem er zu einem der ursprünglichen Eingabe entsprechenden Vektor kombiniert werden kann.

Bisher wurde das System um eine Zeile erweitert und die Berechnung etwas verkompliziert, was noch kein wirklicher Gewinn ist. Eine deutliche Leistungssteigerung ergibt sich jedoch aus der Beobachtung, dass die Multipletts gewisse Muster aufweisen. Dadurch treten Sätze von Differenzvektoren mehrfach auf. Für die Lösung ist es unerheblich, welcher Differenzvektor welchem Feld zugeordnet ist, solange nur eine gültige Zuordnung existiert. Man kann also die gleichen Differenzvektoren wiederverwenden. Dazu muss nur in der zu den diesem Satz von Differenzvektoren gehörenden Zeile eine weitere 1, passend zum weiteren Hauptvektor, eingetragen werden. Die anderen Vektoren dieses Multipletts können damit wegfallen. Dadurch wird das System um beispielsweise zwei Spalten kleiner, die Zeilenzahl bleibt gleich. Einige Lösungen, die sich nur in der Ausgestaltung der Multiplett-Zuordnungen unterscheiden, treten gar nicht mehr als unterschiedliche Lösungen auf, was die Berechnung beschleunigt.

1.2.5 Vektorenzahl

Da wir von unendlich vielen zulässigen Kombinationen ausgehen, macht es Sinn, eine weitere Ungleichung hinzuzufügen, die die Zahl der ausgewählten Vektoren beschränkt. Das ist einfach eine Gleichung, bei der jede von einem ursprünglichen Vektor stammende Variable mit einem Faktor 1 eingeht, die neuen Variablen zur Codierung der Modulo-Arithmetik, der Gamma-Auswahl oder der Differenzvektoren hingegen durch einen Faktor 0 ignoriert werden. Wenn man eine derartige Beschränkung vornimmt, so kann man in Ungleichung 10 auf der vorherigen Seite das doppelte dieser Obergrenze statt ∞ verwenden, und erhält so eine äquivalente Ungleichung mit endlichen Koeffizienten.

Wenn ich in unserem Beispiel die Zahl der Vektoren auf maximal zehn und mindestens einen beschränke, erhalte ich folgende Form:

$$\begin{array}{rrrrrrr|rr}
1 \le & 1\,x_1 & & & +1\,x_4 & +1\,x_5 & & \le 10 & \\
1 \le & 1\,x_1 & -1\,x_2 & -1\,x_3 & +1\,x_4 & & & \le 1 & \\
1 \le & & 1\,x_2 & & +1\,x_4 & & & \le 1 & \\
1 \le & & & 1\,x_3 & +1\,x_4 & & & \le 1 & \\
0 \le & 1\,x_1 & & & +2\,x_4 & +1\,x_5 & +2\,k & \le 0 & \\
0 \le & & & & 1\,x_4 & +2\,x_5 & -2\,x_\gamma & \le \infty & \\
-\infty \le & & & & 1\,x_4 & +2\,x_5 & -20\,x_\gamma & \le 0 & \\
0 \le & 1\,x_1 & -1\,x_2 & -1\,x_3 & & & & \le \infty &
\end{array}
\quad\Bigg|\quad
\begin{array}{rr}
0 \le x_i \le \infty & x_i \in \mathbb{Z} \\
-\infty \le k \le \infty & k \in \mathbb{Z} \\
0 \le x_\gamma \le 1 & x_\gamma \in \mathbb{Z}
\end{array}
\tag{13}$$

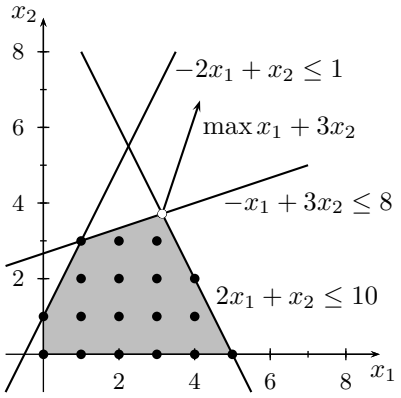
2 Lösungsansätze

Es ist also möglich, die Probleminstanzen der Kopplungen auf Orbifolds in äquivalente „Integer Programs“ zu übersetzen. Jetzt stellt sich das Problem, dazu alle Lösungen zu finden. Ich habe verschiedene Strategien untersucht, und die meisten davon wieder verworfen. Ich möchte hier dennoch jeweils anreißen, wie sie funktionieren, und worin das Problem bei der Anwendung auf die vorliegende Fragestellung besteht.

2.1 Mixed Integer Programming

Verzichtet man in Ungleichung 2 auf Seite 2 auf die Ganzzahligkeitsbedingung, so erhält man ein lineares Optimierungsproblem, ein sogenanntes „Linear Program“. Für diese gibt es effiziente Lösungsmethoden. In der Praxis wird meist der sogenannte *Simplex-Algorithmus* verwendet.[4]

Die n Variablen in x spannen einen n-dimensionalen Vektorraum auf, und die Ungleichungen beschreiben Halbräume darin. Die zulässigen Lösungen liegen im Schnitt dieser Halbräume (in Abbildung 1 grau hinterlegt). Der Simplex-Algorithmus wandert auf der Oberfläche dieses Polytops von Ecke zu Ecke, entsprechend der durch das Optimierungskriterium vorgegebenen Richtung (Pfeil) hin zu einer optimalen Lösung ($\circ$ in der Abbildung).



$$\max \begin{pmatrix} 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$\begin{pmatrix} -2 & 1 \\ -1 & 3 \\ 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 8 \\ 10 \end{pmatrix}$$

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Abbildung 1: Darstellung eines linearen Programms in zwei Variablen.

Führt man zumindest für einzelne Variablen eine zusätzliche Ganzzahligkeitsbedingung ein (in der Abbildung $\bullet$), erhält man ein sogenanntes „Mixed Integer Program“. Da die Ecken des Polytops im Allgemeinen nicht auf ganzzahligen Koordinaten liegen, ist dieser Algorithmus allein nicht zur Lösung von Problemen mit Ganzzahligkeitsbedingungen geeignet. Der Typische Ansatz zur Lösung heißt „Branch, Bound and Cut“.

Branch bezeichnet die Verzweigung an einer Fallunterscheidung. Wenn beispielsweise die reelle Optimallösung den Wert $x_5 = 7{,}2$ enthält, fügt man das eine mal die Ungleichung $x_5 \leq 7$ ein und sucht dort eine ganzzahlige Optimallösung, und dann fügt man statt dessen die Ungleichung $x_5 \geq 8$ ein und sucht erneut rekursiv. So entsteht ein Suchbaum.

Bound ist hilfreich für die Optimierung. Wenn eine zulässige Lösung in einem Teil des Suchbaums gefunden wurde, kann global eine Ungleichung hinzugefügt werden, die besagt, dass jede weitere Lösung im Sinne des Optimierungskriteriums mindestens so gut sein muss. Dadurch kann man womöglich ganze Teilbäume vorzeitig abbrechen, weil die beste reelle Lösung dort bereits schlechter ist.

Cut bezeichnet das Trennen der reellen Optimallösung vom Raum der zulässigen ganzzahligen Lösungen. Wenn beispielsweise eine der Gleichungen die Form $2x_1 + 2x_2 \leq 3$ hat, und von der optimalen Lösung mit Gleichheit erfüllt wird, so kann man die Gleichung durch zwei teilen und runden, erhält damit $x_1 + x_2 \leq 1$, was von allen ganzzahligen Lösungen weiterhin erfüllt wird, von der reellen Optimallösung jedoch bereits verletzt wird. Dadurch verringert man den Abstand zwischen zulässigen reellen und ganzzahligen Lösungen.

Meine Lösung des Problems basiert ebenfalls auf dem Simplex-Algorithmus, ich werde darauf in Abschnitt 3.2 auf Seite 12 noch genauer eingehen. An dieser Stelle ist bereits zu bemerken, dass das Bounding zum Finden einer optimalen Lösung zwar ein sehr nützliches Hilfsmittel, zum finden aller Lösungen hingegen nur hinderlich ist.

2.2 OPBDP

Zum Namen: OPBDP[2] ist ein **O**ptimierer, der den auf **p**seudo-**b**oolsche Aussagen verallgemeinerten **D**avis-**P**utnam-Algorithmus ausführt.

Der Ansatz stammt aus der Aussagenlogik. Eine logische Aussage kann in konjunktiver Normalform dargestellt werden, das heißt als Konjunktion (Und-Verknüpfung) von Disjunktionen (Oder-Verknüpfungen) von Literalen (Aussagenvariablen oder ihren Inversen). Ein Beispiel:

$$(A \vee \bar{C} \vee D) \wedge (B \vee \bar{C} \vee \bar{D}) \tag{14}$$

Wenn man den Aussagen die Zahlen 0 für Falsch und 1 für Wahr zuordnet, kann man die Inversion als $\bar{A} = (1 - A)$ formulieren, und die Konjunktion als $A \vee B = A + B$. Eine einzelne Klausel (Klammer) ist in diesem Fall wahr, wenn ihr Wert größer als 0 ist. Im Beispiel sieht das dann so aus:

$$\begin{aligned} A + (1 - C) + D &\geq 1 \\ B + (1 - C) + (1 - D) &\geq 1 \end{aligned}$$

Dies lässt sich in die uns vertraute Form eines linearen Ungleichungssystems übersetzen.

$$\begin{pmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & -1 & -1 \end{pmatrix} \cdot \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} \geq \begin{pmatrix} 0 \\ -1 \end{pmatrix} \tag{15}$$

Pseudo-boolsche Ungleichungen verallgemeinern dieses Konzept, indem sie vor den einzelnen Literalen beliebige ganzzahlige Koeffizienten zulassen, nicht nur 0 und 1. Auch die rechte Seite der Ungleichung darf eine beliebige ganze Zahl sein. Dadurch lassen sich also im wesentlichen Klauseln formulieren wie „drei dieser fünf Aussagen sollen wahr sein, wobei die eine doppelt zählt“. Bestehen bleibt die Einschränkung, dass die einzelnen Variablen nur 0 und 1 sein dürfen.

Der Davis-Putnam-Algorithmus ist ein Verfahren, um Lösungen klassischer aussagenlogischer Probleme zu finden. Er besteht im wesentlichen aus zwei Schritten.

Propagation feststehender Werte. Sobald eine Klausel ein einzelnes Literal enthält, steht der Wert dieser Variablen fest. Sie kann in allen Klauseln durch eine entsprechende Konstante ersetzt werden. Dadurch können weitere Klauseln mit einem einzigen Literal entstehen, so dass auch dieses festgelegt werden kann. Klauseln ganz ohne freie Variablen sind entweder wahr und können weggelassen werden, oder falsch falls das System unerfüllbar ist.

Fallunterscheidung wenn keine Klausel nur ein Literal enthält. Dann muss eine Variable ausgewählt werden, und rekursiv für beide möglichen Belegungen nach Lösungen gesucht werden. Eine geschickte Heuristik wählt Variablen, deren Festlegung in der Folge möglichst viele andere Variablen auch fest legt.

Der Davis-Putnam-Algorithmus kann auf pseudo-boolsche Ungleichungen verallgemeinert werden. Mit dieser Verallgemeinerung wird er mitunter zum Lösen bestimmter Optimierungsprobleme herangezogen, wobei auch hier Bounding verwendet wird, also das Einführen einer zusätzlichen Gleichung die fordert, dass alle neuen Lösungen im Sinne des Optimierungskriteriums besser sein müssen als die beste bisher gefundene Lösung.

Für die vorliegende Problemstellung erweist es sich als vorteilhaft, dass OPBDP ohne Bounding auf recht einfache Weise alle zulässigen Lösungen aufzählt. Auch die komplett kombinatorische Herangehensweise, die mit ganzzahliger Arithmetik auskommt, ist von Vorteil. Als gravierendes Hindernis hingegen stellt sich die Beschränkung auf binäre Variablen heraus, also Variablen aus $\{0, 1\}$. Eine mögliche Lösung ist Binarisierunug, also das Ersetzen von einer Variablen x durch eine Summe $x_0 + 2x_1 + 4x_2 + \cdots + 2^n x_n$ mit $x_i \in \{0, 1\}$. Nachdem die Gesamtzahl der Vektoren nach oben hin beschränkt wird, ist auch die Zahl der nötigen Bits festgelegt. Eine Formulierung des Problems als OPBDP-Instanz ist also möglich, so dass die existierende Implementierung[3] getestet werden kann.

Testläufe haben jedoch ergeben, dass der OPBDP-Algorithmus im Vergleich zu anderen Algorithmen zu langsam läuft. Offenbar ist die Propagation eher eingeschränkt, die Festlegung einer Variablen legt also kaum andere fest, was zu vielen Fallunterscheidungen führt.

Kernursache könnte bereits die Art des Problems sein. Die meisten Einschränkungen sind Gleichungen mit einer kleinen rechten Seite, oft Null. Eher selten weicht eine gegebene Teilmenge von Vektoren so stark von diesem Ziel ab, dass es nicht mit wenigen zusätzlichen Vektoren noch zu erreichen wäre. Das Erfüllen einer einzelnen Gleichung ist also vergleichsweise einfach, erst das gleichzeitige Erfüllen vieler Gleichungen wird schwer. Weiter erschwert werden dürfte die Sache für OPBDP durch die Binarisierung, da die Koeffizienten dadurch stark unterschiedliche Werte haben und die niederwertigeren Bits erst sinnvoll festgelegt werden können, wenn die höherwertigen feststehen.

2.3 Basisreduktion

Die Lösungsmenge eines ganzzahligen Gleichungssystems, dessen Matrix n Spalten und Rang m hat, also m linear unabhängige Zeilen, ist ein Gitter mit $n - m$ Dimensionen. Mittels einer Basisreduktion kann man eine einen Basis dieses Gitters finden, also Vektoren, mit denen man von einem Gitterpunkt zum nächsten kommen kann[1]. Eine geeignete Basisreduktion ist die LLL-Reduktion[6]. Sie basiert auf der Berechnung einer Hermit-Normalform und ist beispielsweise in LiDIA[12] implementiert. Das transformierte Beispiel aus Gleichung 5 auf Seite 3 sieht so aus:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 2 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ y_1 \\ y_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{16}$$

Der Tatsache, dass die meisten Zeilen Gleichungen beschreiben, ist mit diesem Lösungsansatz bereits Rechnung getragen, und die Ganzzahligkeit übersetzt sich direkt in eine entsprechende Ganzzahligkeit der Koeffizienten vor den neuen Basisvektoren. Dafür gestaltet sich jetzt die Nichtnegativitätsbedingung zum Problem. Sie muss in Ungleichungen über die neuen Koeffizienten übersetzt werden. Die Übersetzung geht zwar einfach, das sich ergebende Ungleichungssystem hat jedoch n Zeilen und $n - m$ Spalten, da die Lösung durch $n - m$ Koeffizienten bestimmt wird und dabei jede der n ursprünglichen Variablen nicht negativ werden darf.

$$n \times m \quad \longrightarrow \quad n \times (n - m) \tag{17}$$

Die Gleichungssysteme zu dem vorliegenden Problem sind schwach bestimmt, mit einer verhältnismäßig kleinen Anzahl von Gleichungen, jedoch einer großen Anzahl von Variablen. Es gilt also $m \ll n$. Dadurch ist das nach der Umformung erhaltene Ungleichungssystem deutlich größer als das ursprüngliche. Laufzeitmessungen ergeben, dass Berechnungen mit verschiedenen anderen hier diskutierten Ansätzen auf dem transformierten Problem deutlich langsamer Ergebnisse liefern als auf der ursprünglichen Formulierung. Das transformierte Problem scheint also nicht auf eine Weise „gutartiger“ zu sein, die die zur Verfügung stehenden Algorithmen zu nutzen wüssten. Für schwach eingeschränkte Gleichungssysteme scheint dieser Ansatz unbrauchbar.

2.4 Endliche Automaten

Endliche Automaten sind ein Werkzeug aus der Theorie formaler Sprachen, sie finden auch im Compilerbau Anwendung. Ein endlicher Automat ist eine Menge von *Zuständen* sowie *Transitionen* zwischen diesen Zuständen. Einer der Zustände ist als *Startzustand* ausgezeichnet, und eine Teilmenge der Zustände ist die Menge *akzeptierender* Zustände. Die Transitionen sind gerichtete Kanten zwischen zwei Zuständen und mit einem Symbol der Eingabe versehen. Der Automat startet im Startzustand, führt entsprechend der Eingabe nacheinander Transitionen aus, und wenn er sich am Ende in einem akzeptierenden Zustand befindet, so akzeptiert er die Eingabe.

Es ist einfach, für eine einzelne Quantenzahl des vorliegenden Problems einen endlichen Automaten zu konstruieren. Dabei stellt der Zustand die bisherige Zwischensumme dar, Transitionen beschreiben die Änderung dieser Summe durch einen bestimmten Vektor. Modulo-Arithmetik lässt sich direkt ausdrücken und beschränkt auf natürliche Weise die Zahl der Zustände, in anderen Fällen muss der maximale Wert einer Quantenzahl durch eine obere Schranke für die Zahl der auszuwählenden Vektoren beschränkt werden.

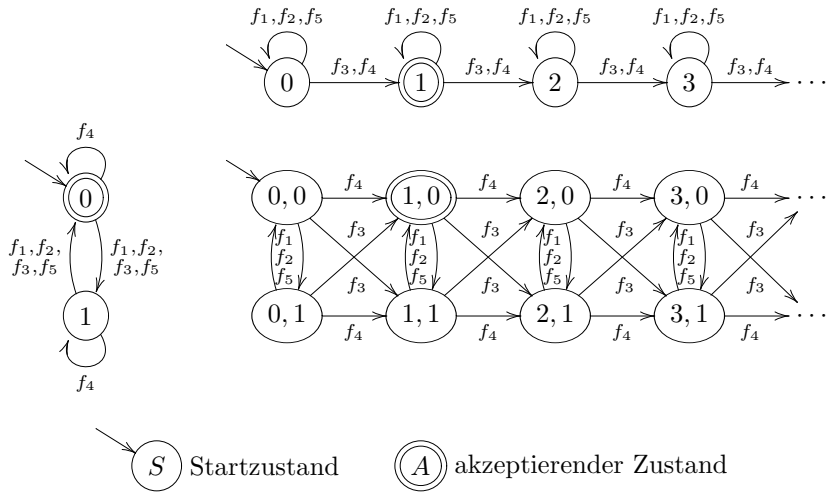


Abbildung 2: Kombination der Automaten zu q_3 (oben) und q_4 (links)

Das Problem besteht jetzt darin, die Automaten der einzelnen Quantenzahlen zu kombinieren, so dass eine Eingabe nur akzeptiert wird, wenn sie jeden Teilautomaten erfüllt. Damit wäre es möglich, effizient alle akzeptierten Eingaben aufzuzählen. Der typische Ansatz zur Kombination mehrerer Automaten besteht darin, als Zusände des neuen Automaten alle Tupel von Zuständen

der Teilautomaten zu verwenden. Die Zustandsmenge des Ergebnisses ist also das kartesische Produkt der Zustandsmengen der Eingaben. Eine Kante existiert im Produktgraphen genau dann, wenn jede Eingabe eine entsprechende Kante aufweist.

Das Problem ist, dass die Größe der Automaten exponentiell mit der Anzahl der Quantenzahlen wächst, als direkte Folge des kartesischen Produktes. Man kann in Zwischenschritten den Automaten vereinfachen, beispielsweise unerreichbare Zustände entfernen. Bei der konkret vorliegenden Fragestellung ist jedoch mit einer Nennenswerten Ausdünnung erst bei Kombination einer recht großen Anzahl von einzelnen Automaten zu erwarten, während das massive Anwachsen des Problems davor schon diesen Ansatz ungeeignet macht.

In der Literatur ist diese Art der Kombination als „Finite State Automata Intersection“ bekannt und zwangsläufig ebenfalls $\mathcal{NP}$-vollständig.[5, 7] Es sind auch kaum Algorithmen bekannt, die für allgemeine reale Probleminstanzen wesentlich bessere Effizienz aufweisen als der oben beschriebene Ansatz über das kartesische Produkt.

3 Implementierung

3.1 `mqn2mps` – Formulierung des LP

Die tatsächlich implementierte Lösung basiert auf der Modellierung als lineares Optimierungsproblem. Für stärkere Flexibilität habe ich die Formulierung des linearen Problems von der Auflistung aller Lösungen getrennt und in einem eigenständigen Programm implementiert. Dieses trägt den Namen `mqn2mps`. MQN steht dabei etwa für „Mathematica Quantum Number“ und ist mein Name für das Eingabeformat, das an das Ausgabeformat eines Mathematika-Programms angelehnt ist. MPS ist ein in der linearen Optimierung weit verbreitetes Austauschformat, so dass die erzeugten Problemformulierungen leicht mit anderen Lösungsprogrammen bearbeitet werden können. Es ist jedoch auch die Erzeugung verschiedener anderer Ausgabeformate vorgesehen.

Das Programm gliedert sich grob in drei Aufgabenkomplexe:

Eingabe erfolgt mit einem generierten Parser. Die Beschreibung der Eingabesprache findet sich in `parser.yy`, die Definition der lexikalischen Tokens in `lexer.lex`. Aus diesen Dateien wird mittels `bison` und `flex` der Parser erzeugt.

Koordination obliegt der Klasse `driver`. Dort wird die Kommandozeile ausgewertet, der Parser gestartet, die Daten aus der Eingabe gesammelt, und die Formulierung für die Ausgabe angestoßen.

Problemformulierung und Ausgabe ist in verschiedenen Klassen für verschiedene Ausgabeformate aufgeteilt. Gemeinsame Basisklasse ist `mipgen` für die allgemeine Generierung eines „Mixed Integer Programs“. Wichtigste abgeleitete Klasse ist `mpsgen` zur Erzeugung des angesprochenen MPS-Formats.

3.1.1 Eingabeformat

Das Eingabeformat MQN ist in Tabelle 2 auf der nächsten Seite dargestellt. Der *Name* ist optional und kann zusammen mit dem Gleichheitszeichen weggelassen werden. Das MPS-Format beispielsweise erlaubt die Angabe eines Namens für das Problem, daher ist eine entsprechende Angabe auch in der Eingabe vorgesehen. Der Name ist eine Zeichenkette, und Zeichenketten sollten in Anführungszeichen gesetzt werden, selbst wenn einfache Worte auch ohne Anführungszeichen als Zeichenketten verwendet werden dürfen. Die Details finden sich in `lexer.lex`.

Dateiformat	::=	[*Name* ‚=‘] ‚{‘ *Quantenzahlnamen* ‚,‘ *Bedingungen* ‚,‘ *Vektoren* ‚}‘
Name	::=	*Zeichenfolge*
Zeichenfolge	::=	‚"‘ *Text* ‚"‘ | *Wort*
Quantenzahlnamen	::=	‚{‘ *Quantenzahlname* [‚,‘ *Quantenzahlname* . . .] ‚}‘
Quantenzahlname	::=	*Zeichenfolge*
Bedingungen	::=	‚{‘ *Bedingung* [‚,‘ *Bedingung* . . .] ‚}‘
Bedingung	::=	‚{‘ *Quantenzahlname* ‚,‘ *Zielwert* ‚,‘ *Modulo* ‚}‘
Zielwert	::=	*Wert*
Wert	::=	*Bruch* | *Asterisk*
Asterisk	::=	‚*‘ | ‚"*"‘
Bruch	::=	[‚+‘ | ‚−‘] *NatürlicheZahl* [‚/‘ *NatürlicheZahl*]
Modulo	::=	*NatürlicheZahl*
Vektoren	::=	‚{‘ *Vektor* [‚,‘ *Vektor* . . .] ‚}‘
Vektor	::=	‚{‘ *Vektorname* ‚,‘ *Quantenzahlwert* [‚,‘ *Quantenzahlwert* . . .] ‚}‘
Vektorname	::=	*Zeichenfolge*
Quantenzahlwert	::=	*Wert*

Tabelle 2: Syntax des MQN-Formats

```
tiny={
{"q1", "q2", "q3", "q4", "gamma"},
{{"q1", 1, 0}, {"q2", 1, 0}, {"q3", 1, 0}, {"q4", 0, 2}},
{{"f123", 1, 0, 0, 1, 0},
 {"f123", 0, 1, 0, 1, 0},
 {"f123", 0, 0, 1, 1, 0},
 {"f4", 1, 1, 1, 2, 1/2},
 {"f5", 0, 0, 0, 1, 1}}}
```

Abbildung 3: Das Beispiel aus Tabelle 1 auf Seite 2 als MQN-Datei mit Multiplett

Die eigentliche Problembeschreibung ist eine Liste, also eine Folge von Elementen, die in geschweifte Klammern eingeschlossen und untereinander durch Kommata getrennt sind. Die Elemente sind ihrerseits wieder Listen in dieser Form. Die Liste *Quantenzahlen* gibt die Namen der einzelnen Quantenzahlen an, und zwar in der Reihenfolge, in der sie in den Verktoren aufgelistet werden. Darauf folgt die Liste *Bedingungen*, deren einzelne Elemente jeweils eine Quantenzahl benenne, für die sie gelten, dann einen Zielwert angeben, den die Summe erfüllen muss, und einen Divisor für die Modulo-Arithmetik. Soll nicht Modulo gerechnet werden, wir der Divisor 0 angegeben. Als letzte Liste kommen die eigentlichen Vektoren, von denen jeder einzelne wieder eine Liste aus einem Namen und einer Folge von Werten ist. Als Wert wird neben den üblichen rationalen Zahlen auch der spezielle Wert `*` akzeptiert; dieser wird intern als 0 interpretiert.

Im `parser` werden die einzelnen Elemente dieser Syntax erkannt und im `driver` gesammelt. Dabei ist zu beachten, dass alle Werte mit der Ausnahme der natürlichen Zahlen immer als Zeiger auf Objekte weitergereicht werden. Eine Funktion, die einen solchen Zeiger als Parameter übergeben bekommt, ist im Allgemeinen für die weitere Speicherverwaltung des entsprechenden Objektes verantwortlich. Es gibt daher einige Konstruktoren im Code, die Zeiger statt Werden als Parameter haben und die referenzierten Objekte nach der Auswertung löschen.

3.1.2 Problemformulierung

Das Formulieren des Problems findet in der Klasse `mipgen` und den für die verschiedenen Ausgabeformate abgeleiteten Klassen statt. Die einzelnen Schritte sind in getrennten Methoden implementiert.

`modfields` erzeugt Feldvektoren für die Modulo-Arithmetik. Für jede Bedingung, die mit Modulo-Arithmetik berechnet wird, wird ein Vektor mit einem dem Divisor entsprechenden Wert erzeugt. Ein Vektor der Eingabe wird im Programm immer als `field` bezeichnet, um Konflikte mit der Datenstruktur `vector` zu vermeiden, daher auch der Name dieser Methode.

`multiply_k` wird nur ausgeführt, wenn die Option `--multiply-k` (`-k`) angegeben wurde, und multipliziert alle Felder, deren Name mit `n_` beginnt, mit dem Feld `k`.

`append_gamma` wird nur ausgeführt, wenn die Option `--gamma` (`-g`) angegeben wurde, und erzeugt zwei Ungleichungen, die die Sonderbedingung für q_γ ausdrücken.

`comdenom` bringt alle Werte einer Zeile auf einen gemeinsamen Nenner.

`get_indices` ermittelt die zu den einzelnen Bedingungen gehörenden Spaltennummern.

`set_bounds` trägt die unteren und oberen Grenzen der einzelnen Zeilen ein.

`copy_row_names` legt eine Liste mit Zeilennamen an.

`collect_fields` fasst mehrere Felder mit gleichem Namen zusammen, um die in Abschnitt 1.2.4 auf Seite 4 beschriebene Codierung der Multipletts durch Differenzvektoren zu formulieren.

`copy_col_names` kopiert einfach nur die Namen der Vektoren aus den `field`-Datenstrukturen in einen eigenständigen `vector`.

`make_matrix` erstellt eine Datenstruktur für die Matrix, passend zum Ausgabeformat der abgeleiteten Klasse. Die Basisklasse verwendet geschachtelte `vector`-Strukturen.

`fill_matrix` trägt die Zähler der Koeffizienten in die Matrix ein. Die dabei verwendete Methode `set_element` kann von der abgeleiteten Klasse passend zur verwendeten Datenstruktur überschrieben werden.

`write` schreibt die fertige Datenstruktur in die Ausgabedatei.

3.2 `bande` – Finden der Lösungen

Nun da das Problem als MPS formuliert ist, kann die Lösung in Angriff genommen werden. Es gibt einige Programme und Bibliotheken, die nach dem Branch Bound and Cut-Prinzip Optimierungsprobleme lösen. Wider erwarten hatte ich massive Probleme, diesen Implementierungen das Bounding abzugewöhnen, so dass ich damit mehrere Lösungen nacheinander finden könnte. Eine einzige Lösung zu finden geht also halbwegs schnell, aber alle Lösungen (bis zu einer vorgegebenen Größe) zu finden war nicht effizient möglich. Nachdem ich viel Zeit verwendet hate auf den Versuch, bestehende Löser von Optimierung zu Enumerierung der Lösungen umzubauen, habe ich ein eigenständiges Programm dazu geschrieben.

Dieses Programm trägt den Namen `bande`, für **B**ranch **and E**numerate. Das problematische Bounding fällt weg, und zur Einfachheit habe ich auch vorerst keine Schnittebenenverfahren implementiert, da diese meist doch recht zeitaufwendig sind und sich am derzeitigen Optimum

orientieren. Zentraler Punkt dieses Programms ist die Klasse `BranchControl`, die den Ablauf des Algorithmus weitgehend steuert.

Zu Beginn eines jeden rekursiven Aufrufs wird das aktuelle lineare Programm mit einem LP-Löser gelöst. Falls es keine zulässigen reellen Lösungen gibt, kann es auch keine zulässigen ganzzahligen Lösungen geben, und der aktuelle Zweig wird verworfen. Ansonsten wird die untere und obere Schranke jeder einzelnen Variablen untersucht. Sind diese gleich (und ganzzahlig, da ich nur ganzzahlige Schranken verwende), so ist die entsprechende Variable festgelegt. Sind alle Variablen festgelegt, so liegt eine ganzzahlige Lösung vor. Andernfalls wird eine noch nicht festgelegte Variable x_i sowie eine geeignete Schranke b_i gewählt, und rekursiv die Fälle $x_i \leq b_i$ sowie $x_i \geq b_i+1$ durch Anpassen der unteren bzw. oberen Schranke dieser Variablen abgehandelt.

Die Heuristik zur Auswahl der geeigneten Verzweigungsvariable und Schranke dürfte großen Einfluss auf die Effizienz des Programms haben. Die von mir implementierte Heuristik wählt diejenige Variable, die in der aktuellen reellen Optimallösung am weitesten von der nächsten ganzen Zahl entfernt ist. Sind alle Variablen (fast) ganzzahlig, wird eine Variable gewählt, bei der der Unterschied zwischen oberer und unterer Schranke minimal ist, die also am nächsten dran ist, endgültig festgelegt zu werden.

Am Anfang des Programms wird eine zufällige (aber deterministisch reproduzierbare) Zielfunktion gewählt, um eine Optimierungsrichtung vorzugeben. Dadurch werden einige Mehrdeutigkeiten vermieden. Da schon das Finden einer zulässigen Lösung ähnlich aufwendig ist wie das finden einer optimalen Lösung sollte die zusätzliche und streng genommen nicht zwingend erforderliche Optimierung schlimmstenfalls einen Faktor zwei zur Programmlaufzeit beitragen.

Eine Besonderheit der Implementierung ist der `UndoManager`. Jede Änderung, die in einem rekursiven Zweig an der Problemformulierung vorgenommen wird, muss wieder rückgängig gemacht werden, wenn ein anderer Zweig besucht wird. Der `UndoManager` sammelt all diese Änderungen in Objekten, die von `UndoBase` abgeleitet sind. Dadurch kann das Programm leicht erweitert werden, da jedes neue Modul einfach seine Änderungen beim UndoManager registrieren kann und sich danach keine Gedanken mehr darum machen muss.

Auch in anderer Hinsicht ist das Programm auf gute Erweiterbarkeit und Modularität ausgelegt. Als Schnittstelle zum LP-Löser wird Osi[13] verwendet, das Open Solver Interface aus dem COIN-OR-Projekt[10]. Derzeit verwendet dies den ebenfalls aus COIN-OR stammenden Löser Clp[9], aber der Austausch dieses Lösers gegen einen anderen von OSI unterstützten gestaltet sich einfach. Die Funktionalität von OSI wird noch einmal in einer Klasse `LinearProgram` gekapselt, sie sich auch um den `UndoManager` kümmert. Um nicht immer wieder mit Rundungsproblemen umgehen zu müssen, wird diese Klasse wiederum in eine Klasse `IntegerProgram` verpackt, bei der alle Schranken ganzzahlig sind.

3.3 Installation und Anwendung

Das Programm sollte sich unter GNU/Linux ganz einfach durch Ausführen von `make` im ausgepackten Verzeichnis compilieren lassen. Die Programme `bison` und `flex` müssen installiert sein, um den Parser für das Eingabeformat zu erzeugen. Osi[13] wird automatisch heruntergeladen und im Arbeitsverzeichnis installiert. Am Ende finden sich im Unterverzeichnis `bin` vier Programme, von denen eines, `clp`, von Osi installiert wurde. Zwei weitere sind die oben beschriebenen `mqn2mps` und `bande`. Diese wurden statisch gegen die Osi-Bibliotheken gelinkt und können daher beliebig verschoben werden.

Als drittes liegt in diesem Verzeichnis das Perl-Script `reformat.pl`, das die Ausgabe nachbearbeitet. `mqn2mps` führt, wie oben beschrieben, einige zusätzliche Variablen ein, um verschiedene Aspekte der Problemaspekte zu modellieren. `bande` ist als allgemein verwendbares Programm zur Aufzählung ganzzahliger Lösungen geschrieben, und beachtet die zusätzlich eingeführten Spalten

gar nicht. Jede Ausgabezeile besteht aus Einträgen der Form *Variable=Wert*. Die Nachbearbeitung entfernt von `mqn2mps` eingeführte Variablen, die an einem führenden Unterstrich erkannt werden. Sie wandelt Werte in entsprechende Mehrfachnennungen um, sortiert die Vektoren, eliminiert mehrfach gefundene Lösung (durch unterschiedliche Differenzvektoren) und sortiert die verbleibenden Lösungen nach Vektorenzahl und dann lexikographisch.

```
bin/mqn2mps --gamma --min-order=3 --max-order=4 samples/test1.mqn
bin/bande --output=samples/test1.sol samples/test1.mps
bin/reformat.pl samples/test1.sol
```

Abbildung 4: Beispielsitzung

Abbildung 4 zeigt eine beispielhafte Anwendungssitzung, in der alle Lösungen mit drei bis vier Vektoren zu einer mitgelieferten Eingabedatei ermittelt werden. `bande` benötigt für das Finden von 461 Lösungen bei mir deutlich unter drei Minuten. Nach `reformat.pl` bleiben davon noch 216 unterschiedliche Lösungen übrig. Alle Programme verfügen über eine Online-Hilfe, die man mit dem Parameter `--help` ansehen kann und die die möglichen Kommandozeilenparameter beschreibt.

4 Diskussion

Der Vergleich mit Laufzeiten, die Herr Ratz mir genannt hat, zeigt, dass mein Programm bei kleiner Obergrenze für die Zahl der Vektoren nicht so schnell läuft wie seines, aber bei mittlerer und großer Vektorenzahl das schnellere ist. Es skaliert also besser. Interessant wäre eine Untersuchung, ob sich die Vorteile beider Programme kombinieren lassen,

Auch wenn mein Systementwicklungsprojekt hiermit abgeschlossen ist, gibt es sicher noch einige Möglichkeiten, die Leistungsfähigkeit des Programms zu optimieren. Mir sind dazu bereits einige Möglichkeiten eingefallen.

Andere LP-Löser. Während CLP ein frei verfügbarer LP-Löser ist, gibt es auf dem Markt auch einige kommerziell erhältliche Implementierungen, die in vielen Fällen deutlich schneller das Optimum finden. Besonders CPLEX[11] ist ein oft verwendeter und auch von OSI unterstützter kommerzieller Löser.

Bessere Heuristiken. Die Auswahl der Variable, die zur Verzweigung verwendet wird, sowie der Schranke, an der verzweigt wird, geschieht recht einfach und intuitiv. Es sind ganze Artikel über geschickte Auswahl von Variablen in ähnlichen Situationen geschrieben worden, und es ist daher gut vorstellbar, dass ein geschickterer Ansatz hier gewisse Leistungsgewinne herausholen kann. Durch Laufzeitmessungen auf realen Problemeinstanzen lässt sich jeder neue Vorschlag schnell evaluieren.

Problemspezifische Optimierung. `bande` ist derzeit ein allgemeines Programm zur Aufzählung von Lösungen beliebiger Integerprogramme. Womöglich kann man unter Ausnutzung der Struktur und der physikalischen Bedeutung der konkreten Probleminstanzen noch klügere Entscheidungen treffen und somit die Laufzeit verbessern.

Verteilte Berechnung. Das Verzweigungsprinzip eignet sich hervorragend zur verteilten Berechnung der Lösungen. Ein einziger Prozess könnte den Suchbaum bis zu einer gewissen Tiefe aufbauen und die Situation an den Stellen, wo er abbricht, jeweils speichern. Damit hätte man eine Reihe von Unterproblemen, die sich unabhängig voneinander bearbeiten

lassen. Es wäre möglich, das Programm beispielsweise an BOINC[8] anzupassen und somit eine verbreitete Middleware für verteilte Berechnungen zu nutzen. Erster Schritt zu jeder Art von verteilter Berechnung wäre eine Methode, den aktuellen Zustand der Berechnung zu speichern und beim nächsten Programmstart wieder aufzunehmen.

Literatur

[1] Aardal, K.; Hurkens, C.; Lenstra, A. K.: Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of operations research*, 25(3), (2000), S. 427–442. ISSN 0364-765X.

[2] Barth, P.: A Davis-Putnam Based Enumeration Algorithm for Linear pseudo-Boolean Optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Januar 1995.

[3] Barth, P.: OPBDP – A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization, August 2005.
`http://www.mpi-inf.mpg.de/departments/d2/software/opbdp/`

[4] Cormen, T. H.; et al.: *Introduction to Algorithms*, Kapitel 29: *Linear Programming*, S. 770–821. The MIT Press, 2 Auflage. ISBN 0-262-03293-7, 2001.

[5] Garey, M. R.; Johnson, D. S.: *Computers and Intractability*. W. H. Freeman and Company, 1979. ISBN 0-7167-1045-5.

[6] Lenstra, A. K.; Lenstra, H. W., Jr.; Lovász: Factoring Polynomials with Rational Coefficients. *Mathematische Annalen*, 261(4), (1982), S. 515–534.

[7] Wareham, H. T.: The Parameterized Complexity of Intersection and Composition Operations on Sets of Finite-State Automata. In: *Implementation and Application of Automata*, Band 2088 von *Lecture Notes in Computer Science*. Springer. ISSN 1611-3349, 2001. S. 302–310.

[8] BOINC – Berkeley Open Infrastructure for Network Computing.
`http://boinc.berkeley.edu/`

[9] Clp – Coin-or linear programming. Teil von COIN-OR[10].
`https://projects.coin-or.org/Clp`

[10] COIN-OR – Computational Infrastructure for Operations Research.
`http://www.coin-or.org/`

[11] ILOG CPLEX.
`http://www.ilog.com/products/cplex/`

[12] LiDIA – A C++ Library For Computational Number Theory, März 2006.
`http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/`

[13] Osi – Open Solver Interface. Teil von COIN-OR[10].
`https://projects.coin-or.org/Osi`