Albert-Ludwigs-Universität Freiburg • D-79085 Freiburg

Albert-Ludwigs-
Universität Freiburg

Studienarbeit

Binäre lineare Optimierung mit K*BMDs

Ralf Wimmer

24. Juni 2003

Inhaltsverzeichnis

1 Einleitung

In dieser Studienarbeit will ich untersuchen, wie ein spezieller Typ von binären Entscheidungsdiagrammen, die K*BMDs, dazu verwendet werden können, um binäre lineare Optimierungsprobleme zu lösen.
Die bisherigen Branch-and-Bound Algorithmen scheinen dann bei diesem NP-vollständigen Problem schlecht abzuschneiden, wenn Symmetrien auftreten. Da K*BMDs klein sind, wenn Symmetrien vorhanden sind, besteht die Hoffnung, daß mein Algorithmus in diesen Fällen eine bessere Laufzeit besitzt als andere Ansätze.

Die Studienarbeit ist folgendermaßen gegliedert:
Im nächsten Kapitel definiere ich, was ein binäres lineares Optimierungsproblem ist und stelle einige Verallgemeinerungen vor, die auf das binäre Problem zurückführbar sind. Die Komplextiät des Problems wird untersucht. Außerdem werden Syntax und Semantik von K*BMDs beschrieben.
In Kapitel 3 wird der Algorithmus dargestellt, der zur Lösung dieses Optimierungsproblems verwendet werden kann. Ich erläutere die einzelnen Teile des Algorithmus' und stelle die Probleme dar, die auftreten können. Es wird auch eine Lösungsmöglichkeit präsentiert.
Da der Algorithmus an einigen Stellen noch sehr ungenau ist und viel Spielraum läßt, werden in Kapitel 4 Heuristiken vorgestellt, die dazu beitragen sollen, die Laufzeit zu verbessern.
Kapitel 5 enthält eine Beschreibung der Implementierung in C++, wobei ich auf das DD-Package von Marc Herbstritt aufbaue.
In Kapitel 6 beschreibe ich sowohl das Standardeingabeformat MPS für ILP-Solver sowie mein eigenes vereinfachtes Eingabeformat.
Abschließend führe ich in Kapitel 7 einige experimentalle Egebnisse auf und stelle einen Vergleich zum ILP-Solver an, der in [5] beschrieben ist.

2 Die Grundlagen

2.1 Binäre lineare Optimierung

Definition (Binäres lineares Optimierungsproblem, BILP):
Finde $x_1, \ldots, x_n \in \{0, 1\}$, so daß der Wert der Zielfunktion (*goal*)

$$g(x_1, \ldots, x_n) = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

minimal wird unter der Voraussetzung, daß folgende Ungleichungen (*constraints*) gelten:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + b_1 &\leq 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n + b_2 &\leq 0 \\ \vdots \qquad\qquad & \qquad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n + b_m &\leq 0. \end{aligned}$$

Dabei seien a_{ij}, b_i und c_j $(i = 1, \ldots, m,\ j = 1, \ldots, n)$ ganze Zahlen.

Beispiel:

Das folgende ist ein typisches Beispiel für ein BILP:

$$\begin{aligned} -2x_1 + x_2 - 5x_3 + 6x_4 + x_5 - x_6 + 4 &\leq 0 \\ -x_2 + 2x_3 - 3x_4 \qquad - x_6 \qquad &\leq 0 \\ g(x_1, \ldots, x_6) = -3x_1 + 2x_2 + 2x_3 \qquad - x_5 - 2x_6 + 6 &\rightsquigarrow \min \end{aligned}$$

Es besitzt die Lösung

Variable	x_1	x_2	x_3	x_4	x_5	x_6
Wert	1	1	1	0	1	1

$$g(1,1,1,0,1,1) = 4$$

2.1.1 BILP als Spezialfall

- Das oben definierte binäre lineare Optimierungsproblem (BILP) ist ein Spezialfall des ganzzahligen linearen Optimierungsproblems (ILP), bei dem die $x_i \in \mathbb{Z}$ sind. Ist der Wertebereich der x_i beschränkt, so kann das ILP auf das BILP zurückgeführt werden: Sei $-2^{k-1} \leq x_i \leq 2^{k-1} - 1$. Dann ersetze x_i durch

$$\sum_{j=1}^{k-1} 2^i x_{ij} - 2^n x_{ik}$$

  mit $x_{ij} \in \{0, 1\}$. Man erhält als Ergebnis die Zweierkomplementdarstellung der Lösung des ILP.

- Wir haben oben das BILP als Minimierungsproblem formuliert. Ein Maximierungsproblem kann in ein Minimierungsproblem umgewandelt werden, in dem die Vorzeichen der Koeffizienten in der Goal-Funktion geändert werden.

- Wir könnten beim BILP als Constraints andere lineare Ungleichungen und Gleichungen zulassen. Dies ist aber nicht nötig, da diese auf die oben geforderte Form gebracht werden können:

$$\begin{aligned} Ax \leq b \quad &\rightsquigarrow \quad Ax - b \leq 0 \\ Ax < b \quad &\rightsquigarrow \quad Ax - b + 1 \leq 0 \\ Ax \geq b \quad &\rightsquigarrow \quad -Ax + b \leq 0 \\ Ax > b \quad &\rightsquigarrow \quad -Ax + b - 1 \leq 0 \\ Ax = b \quad &\rightsquigarrow \quad Ax - b \leq 0 \quad \text{und} \quad -Ax + b \leq 0 \end{aligned}$$

2.1.2 Die Komplexität von BILP

Satz:
Das Problem, eine Variablenbelegung zu finden, so daß alle Constraints erfüllt sind, ist NP-vollständig.

Beweis: [3]

Sei $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$. Gesucht ist ein Vektor $c \in \{0, 1\}^n$ mit $Ac \leq b$.

Es ist offensichtlich, daß BILP$\in NP$ ist; man bestimme nichtdeterministisch einen Vektor c und prüfe, ob $Ac \leq b$.
Wir zeigen, daß sich SAT auf BILP reduzieren läßt. Sei

$$\alpha = z_1 \wedge z_2 \wedge \cdots \wedge z_k$$

eine Formel in konjunktiver Normalform und seien $x_1, \ldots, x_n$ die Variablen, die in α vorkommen. Wir definieren A und b folgendermaßen:

$$A_{ij} = \begin{cases} -1 & \text{falls } x_j \text{ in } z_i \text{ vorkommt} \\ 1 & \text{falls } \bar{x}_j \text{ in } z_i \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

und

$$b_i = \text{Anzahl der Variablen } x_j, \text{ wobei } \bar{x}_j \text{ in } z_i \text{ vorkommt} - 1.$$

Behauptung: α ist erfüllbar genau dann, wenn es einen Vektor $c \in \{0, 1\}^n$ gibt mit $Ac \leq b$.

„$\Rightarrow$“ Sei ψ eine Belegung mit $\psi(\alpha) = 1$. Definiere $c_j = \psi(x_j)$ für alle $j = 1, \ldots, n$. Dann ist

$$\begin{aligned} (Ac)_i = \sum_{j=1}^{n} A_{ij} c_j &= \sum_{\bar{x}_j \in z_i} \psi(x_j) - \sum_{x_j \in z_i} \psi(x_j) \\ &\leq \left(\sum_{\bar{x}_j \in z_i} 1 \right) - 1 = b_i, \end{aligned}$$

denn entweder gibt es ein $x_j \in z_i$ mit $\psi(x_j) = 1$ oder ein $\bar{x}_j \in z_i$ mit $\psi(x_j) = 0$.

„$\Leftarrow$“ Sei c ein Vektor mit $Ac \geq b$. Wir definieren eine Variablenbelegung ψ durch $\psi(x_j) = c_j$. Angenommen, es wäre $\psi(\alpha) = 0$. Dann muß es ein $i \in \{1, \ldots, k\}$ geben mit $\psi(z_i) = 0$. In diesem z_i muß für alle Variablen gelten:
Wenn x_j in z_i vorkommt, dann ist $\psi(x_j) = 0$ und wenn $\bar{x}_j$ vorkommt, dann ist $\psi(x_j) = 1$.
Somit gilt:

$$\begin{aligned} b_i \geq (Ac)_i = \sum_{j=1}^{n} A_{ij} c_j &= \sum_{\bar{x}_j \in z_i} \psi(x_j) - \sum_{x_j \in z_i} \psi(x_j) \\ &= \sum_{\bar{x}_j \in z_i} 1 > b_i \end{aligned}$$

Widerspruch!!
Also muß $\psi(\alpha) = 1$ gelten. q.e.d.

Das Optimierungsproblem BILP ist folglich ebenfalls NP-hart, da eine Lösung des BILP auch eine erfüllende Belegung für die Constraints ist.
Somit können wir kaum erwarten, einen Algorithmus zu finden, der in jedem Fall effizient ist.

2.2 K*BMDs

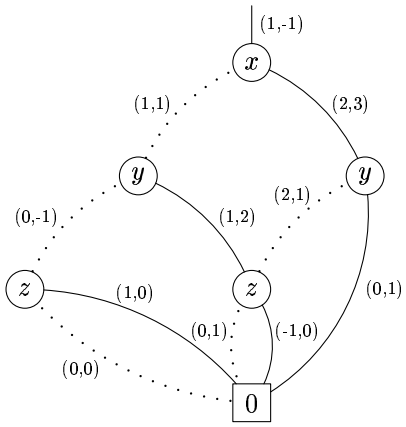
Wir wollen K*BMDs verwenden, um das oben vorgestellte BILP zu lösen. Deshalb wollen wir kurz die Syntax und die Semantik von K*BMDs definieren. Für weitere Informationen siehe [2]

Definition (Syntax von K*BMDs):
Ein K*BMD über einer Variablenmenge X_n ist ein gerichteter, azyklischer und zusammenhängender Graph $\mathcal{G} = (V, E)$ mit genau einer Wurzel zusammen mit einer Dekompositionstypliste $DTL : X_n \to \{S, nD, pD\}$, so daß gilt:

1. Jeder in V enthaltene Knoten ist entweder ein terminaler oder nicht-terminaler Knoten.
2. Jeder terminale Knoten ist mit einer Zahl $z \in \mathbb{Z}$ markiert und besitzt keine ausgehenden Kanten.
3. Jeder nicht-terminale Knoten v ist mit einer Booleschen Variablen $x_i \in X_n$ markiert und hat genau zwei ausgehende Kanten, deren Endpunkte mit $low(v)$ und $high(v) \in V$ bezeichnet werden.
4. Die Wurzel besitzt eine eingehende Kante ohne Anfangspunkt.
5. Jede Kante $e = (v_1, v_2) \in E$ ist mit $(a, m) \in \mathbb{Z} \times \mathbb{Z}$, dem additiven Gewicht a und dem multiplikativen Gewicht m markiert.

Definition (Semantik von K*BMDs):
Sei ein K*BMD über der Variablenmenge X_n gegeben.

1. Ein Terminalknoten v, der die Markierung $z \in \mathbb{Z}$ trägt, stellt die konstante Funktion $f_v(X_n) = z$ dar.
2. Einer Kante e, von deren Endknoten v die dargestellt Funktion $f_v(X_n)$ bekannt ist und die die Markierung (a, m) trägt, wird die Funktion
$$f_e(X_n) = a + m \cdot f_v(X_n)$$
zugeordnet.
3. Sei v ein nicht-terminaler Knoten, von dem die Funktionen der beiden ausgehenden Kanten e_{low} und e_{high} bekannt sind. v sei mit der Variablen x_i beschriftet.
Wir unterscheiden 3 Fälle:
   - $DTL(v) = S$: Dann ist die dargestellte Funktion gegeben durch:
   $$f_v(X_n) = (1 - x_i) \cdot f_{e_{low}}(X_n) + x_i \cdot f_{e_{high}}(X_n).$$
   - $DTL(v) = pD$: Dann wird die Funktion
   $$f_v(X_n) = f_{e_{low}}(X_n) + x_i \cdot f_{e_{high}}(X_n)$$
   dargestellt.



Zerlegungstypliste:

x	y	z
pD	nD	S

Abbildung 1: Das K*BMD für das Beispiel

- $DTL(v) = nD$: Dann stellt der Knoten v die Funktion

$$f_v(X_n) = f_{e_{low}}(X_n) + (1 - x_i) \cdot (-f_{e_{high}}(X_n))$$

dar.

Die vom gesamten K*BMD dargestellte Funktion ist gegeben durch die Funktion, die an der eingehenden Kante der Wurzel dargestellt wird.

Beispiel:

Wir wollen am Beispiel von Abbildung 1 zeigen, wie durch ein K*BMD eine Pseudo-Boolesche Funktion[1] dargestellt werden kann. Wir berechnen dabei die von den einzelnen Knoten dargestellten Funktionen ausgehend vom Terminalknoten zur Wurzel.

- Der terminale 0-Knoten stellt die Funktion $f_0(x, y, z) = 0$ dar.
- Der linke, mit z beschriftete Knoten besitzt Shannon-Zerlegung (S) und stellt somit die Funktion

$$f_{z_l}(x, y, z) = (1 - z) \cdot (0 + 0 \cdot 0) + z \cdot (1 + 0 \cdot 0) = z$$

dar.

- Dem rechten z-Knoten wird die Funktion

$$f_{z_r}(x, y, z) = (1 - z) \cdot (0 + 1 \cdot 0) + z \cdot (-1 + 0 \cdot 0) = -z$$

zugeordnet.

- Der linke y-Knoten besitzt den Zerlegungstyp nD und repräsentiert die Funktion:

$$\begin{aligned} f_{y_l}(x, y, z) &= 0 - 1 \cdot f_{z_l}(x, y, z) + (1 - y) \cdot (-(1 + 2 \cdot f_{z_r}(x, y, z))) \\ &= 0 - z + (1 - y)(-1 + 2z) = -1 + z + y - 2yz \end{aligned}$$

[1] Eine Pseudo-Boolesche Funktion ist eine Funktion $f : \{0, 1\}^n \to \mathbb{Z}$.

- Der rechte y-Knoten stellt die Funktion

$$f_{y_r}(x, y, z) = 2 + 1 \cdot (-z) + y \cdot (0 + 1 \cdot 0) = 2 - z$$

  dar.

- Die Funktion, die am x-Knoten dargestellt ist, lautet

$$\begin{aligned} f_x(x, y, z) &= 1 + 1 \cdot f_{y_l}(x, y, z) + x \cdot (2 + 3 \cdot f_{y_r}(x, y, z)) \\ &= 1 + (-1 + y + z - 2yz) + 2x + 3x(2 - z) \\ &= 8x + y + z - 2yz - 3xz \end{aligned}$$

Damit stellt das komplette K*BMD die Funktion

$$f(x, y, z) = 1 - 1 \cdot f_x(x, y, z) = 1 - 8x - y - z + 2yz + 3xz$$

dar.

2.2.1 Darstellung und Operationen auf K*BMDs

Als nächstes sollten wir uns überlegen, wie man ein K*BMD darstellen kann und welche Operationen auf ihnen wichtig sind.

Ein K*BMD kann dargestellt werden als ein Tripel $\langle a, m, G \rangle$. Dabei ist

- G der zugrundeliegende Graph.
- a ist das additive Gewicht der eingehenden Kante.
- m ist das multiplikative Gewicht der eingehenden Kante.
- $var(G)$ ist die Variable, mit der der Wurzelknoten von G beschriftet ist.
- $\langle a_l, m_l, G_l \rangle = lowSon(G)$ ist der low-Sohn des K*BMDs. Dieser ist wieder ein vollständiges K*BMD.
- $\langle a_r, m_r, G_r \rangle = highSon(G)$ ist der high-Sohn des K*BMDs. Auch hierbei handelt es sich wieder um ein vollständiges K*BMD.

Die Operationen *lowSon* und *highSon* sind nur definiert, wenn das K*BMD nicht nur aus einem Blatt besteht.

Da wir später häufig das Maximum und Minimum der durch K*BMDs dargestellten Funktionen berechnen müssen, beschränken wir uns auf die Shannon-Zerlegung. Dann kann die Bestimmung dieser Werte in linearer Zeit bzgl. der Zahl der inneren Knoten erfolgen.

3 Der Algorithmus

3.1 Vorverarbeitung

Bevor der eigentliche Algorithmus zur Lösung von BILP ausgeführt wird, wird mittels eines Präprozessors geprüft, ob der Wert einiger Variablen in Voraus festgestellt werden kann. Betrachten wir dazu folgendes Beispiel für einen Constraint:

$$12x_1 - 5x_2 + 2x_3 - 3x_4 + x_5 + 4 \leq 0$$

Nehmen wir an, wir würden x_1 mit 1 belegen. Dann gibt es keine Belegung für $x_2, \ldots, x_5$, so daß der Constraint erfüllt ist. Deshalb muß bei jeder Lösung gelten: $x_1 = 0$.
Entsprechend muß, wie man leicht nachrechnet, $x_2 = 1$ gelten.
Folglich können wir in *allen* Constraints $x_1 = 0$ und $x_2 = 1$ setzen.

Der folgende Algorithmus (siehe auch [1]) versucht, Variablen zu finden, deren Wert, wie oben beschrieben, festgelegt werden kann:

Algorithmus preProcess
Input: Matrix (a_{ij}) der Koeffizienten, rechte Seiten b
Output: `true`, falls die Suche erfolgreich war, sonst `false`

```
1   ergebnis = false;
2   // Über jeden Constaint iterieren.
3   for i = 1 to m do {
4     // Summe der negativen Koeffizienten berechnen.
5     sum = b_i;
6     for j = 1 to n do {
7       if (a_ij < 0) sum = sum + a_ij; }

8     // Variablen x_j suchen, deren Wert schon feststeht.
9     for j = 1 to n do {
10      // Unterscheidung: a_ij ≤ 0 oder a_ij > 0.
11      if (a_ij ≤ 0) {
12        if (sum - a_ij > 0) {
13          print('x_j = 1');
14          ergebnis = true;
15          for k=1 to m do { b_k = b_k + a_kj; a_kj = 0; }
16        }
17      } else {                                  // a_ij > 0
18        if (sum + a_ij > 0) {
19          print('x_j = 0');
20          ergebnis = true;
21          for k=1 to m do { a_kj = 0; }
22        }
23      }
```

```
24      }
25    }
26    return ergebnis;
```

Dieser Algorithmus wird solange ausgeführt, bis sein Rückgabewert `false` ist.

Nach der Vorstellung des Algorithmus' zur Vorverarbeitung der Eingabedaten stellen wir den eigentlichen Lösungsalgorithmus auf der Basis von K*BMDs und OBDDs vor. Dieser wurde für EVBDDs von Y.-T. Lai et al. in [5] beschrieben.

3.2 LeqZero

Im ersten Schritt konstruieren wir aus den linken Seiten der Constraints und aus der Goal-Funktion jeweils K*BMDs. Da wir hier nur lineare Funktionen betrachten, besitzen alle entstehenden K*BMDs lineare Größe in der Anzahl der Variablen, von denen sie jeweils abhängen. Die Constraint-DDs müssen dann in die charakteristischen Funktionen

$$\chi_i(x_1, \ldots, x_n) = \begin{cases} 1 & \text{falls } a_{i1}x_1 + \cdots + a_{in}x_n + b_i \leq 0 \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

umgewandelt werden. Dies erfolgt mit Hilfe des Algorithmus `LessEqualZeroSymbolic`, der nachfolgend angegeben ist:

Algorithmus leqZero
Input: K*BMD $\langle a, m, G\rangle$
Output: OBDD H

```
1   if (max(⟨a, m, G⟩) ≤ 0) return 1;
2   if (min(⟨a, m, G⟩) > 0) return 0;
3   if (compTableLookup(⟨a, m, G⟩, ans) return ans;

4   ⟨a_l, m_l, G_l⟩ = lowSon(G);
5   ⟨a_h, m_h, G_h⟩ = highSon(G);
6   H_l = leqZero(⟨a + m · a_l, a + m · m_l, G_l⟩);
7   H_h = leqZero(⟨a + m · a_h, a + m · m_h, G_h⟩);
8   if (H_l == H_h) return H_h;
9   H = ite(var(G), H_h, H_l);
10  compTableInsert(⟨a, m, G, ⟩, H);
11  return H;
```

Erläuterungen:
Die Zeilen 1 bis 3 überprüfen, ob ein Basisfall vorliegt, bei dem die Lösung sofort angegeben werden kann.

1. Ist das Maximum der dargestellten Funktion kleiner gleich Null, dann ist die Ungleichung immer erfüllt. Es wird die konstante 1-Funktion zurückgegeben.
2. Ist das Minimum größer als Null, dann ist der Constraint unerfüllbar. Es wird die konstante 0-Funktion zurückgegeben.
3. Es wird überprüft, ob das Ergebnis in der ComputedTable enthalten ist. Wenn ja, dann wird dieses zurückgegeben.

In den Zeilen 4 bis 7 wird das K*BMD in die beiden Kofaktoren zerlegt. Auf diese wird rekursiv der Algorithmus angewendet. In Zeile 9 wird mit Hilfe des ITE-Operators („if-then-else“) die gesamte Lösung zusammengesetzt (zur Funktionsweise von ITE siehe [2]). Sie wird in Zeile 10 in die ComputedTable eingefügt und dann zurückgegeben.

Mit dem Algorithmus `leqZero` können wir die charakteristischen Funktionen $\chi_1, \ldots, \chi_n$ der Constraints bestimmen. Im nächsten Schritt werden diese charakteristischen Funktionen mit AND verknüpft. Dies geschieht mit dem ITE-Operator auf folgende Weise:
Seien G_1 und G_2 zwei OBDDs, die mit AND verknüpft werden sollen. Dann liefert der folgende Aufruf von ITE die AND-Verknüpfung der beiden OBDDs:

`ITE(`G_1`, `G_2`, 0)`

Nachdem wir so alle Constraints verknüpft haben, erhalten wir eine Darstellung der charakteristischen Funktion χ, die die zulässigen Werte der Variablen beschreibt.

Wir haben jetzt noch die Goal-Funktion und die charakteristische Funktion übrig. Auf diese wenden wir den Algorithmus `minimize`[5] an:

3.3 Minimize

Algorithmus Minimize
Input:

- Goal $\langle a_g, m_g, G_g \rangle$
- Charakterist. Fkt. G_χ
- Der Wert der bisher besten Lösung bound

Output: TRUE, wenn eine bessere Lösung gefunden wurde, FALSE sonst.

```
// Basisfälle überprüfen
1   if (isZero(Gχ)) return FALSE;
```

```
2   if (min(⟨a_g, m_g, G_g⟩) ≥ bound) return FALSE;
3   if (isConstant(⟨a_g, m_g, G_g⟩)) {
4     bound = a_g;
5     return TRUE; }
6   if (isOne(G_χ)) {
7     bound = min(⟨a_g, m_g, G_g⟩);
8     return TRUE; }

// In der ComputedTable nachschauen
9   localBound = bound - a_g;
10  if (compTableLookUp(⟨0, m_g, G_g⟩, G_χ), entry) {
11    if (entry.value < entry.bound) {
12      if (entry.value < localBound) {
13        bound = entry.value + a_g;
14        return TRUE;
15      } else return FALSE;
16    } else {
17      if (localBound ≤ entry.bound) return FALSE;
18  }

// Aufspalten in Cofaktoren
19  if (level(var(G_g) <= level(var(G_χ)) topVar = var(G_g);
20  else topVar = var(G_χ);
21  ⟨a_gl, m_gl, G_gl⟩ = cofactor(⟨0, m_g, G_g⟩, topVar, 0);
22  ⟨a_gh, m_gh, G_gh⟩ = cofactor(⟨0, m_g, G_g⟩, topVar, 1);
23  G_χl = cofactor(G_χ, topVar, 0);
24  G_χh = cofactor(G_χ, topVar, 1);

// Rekursiv aufrufen
25  entry.bound = localBound;
26  if (min(⟨a_gl, m_gl, G_gl⟩) < min(⟨a_gh, m_gh, G_gh⟩) {
27    lRet = Minimize(⟨a_gl, m_gl, G_gl⟩, G_χl, localBound);
28    hRet = Minimize(⟨a_gh, m_gh, G_gh⟩, G_χh, localBound);
29  } else {
30    hRet = Minimize(⟨a_gh, m_gh, G_gh⟩, G_χh, localBound);
31    lRet = Minimize(⟨a_gl, m_gl, G_gl⟩, G_χl, localBound);
32  }
// Bessere Lösung gefunden?
33  if (lRet || hRet) {
34    bound = localBound + a_g;
35    entry.value = localBound;
36    compTableInsert(⟨0, m_g, G_g⟩, G_χ, entry);
37    return TRUE;
38  } else {
39    entry.value = entry.bound;
40    compTableInsert(⟨0, m_g, G_g⟩, G_χ, entry);
41    return FALSE;
```

```
42  };
```

Erläuterungen:
Der Algorithmus besteht aus folgenden Teilen:

- Überprüfen der Basisfälle (Zeilen 1 bis 8)
- Nachschauen in der ComputedTable (Zeilen 9 bis 18)
- Zerlegen der K*BMDs in Cofaktoren (Zeilen 19 bis 24)
- Rekursiver Aufruf (Zeilen 25 bis 32)
- Zusammensetzen der Lösung (Zeilen 33 bis 42)

Im folgenden gehe ich auf die einzelnen Teile ein:

- *Überprüfen der Basisfälle:*
  Zeile 1 prüft, ob die charakteristische Funktion konstant 0 ist. Dann gibt es keine Lösung, und es wird folglich FALSE zurückgegeben.
  Wenn das Minimum größer oder gleich `bound` ist, dann wird in Zeile 2 FALSE zurückgeliefert.
  Als nächstes wird in den Zeilen 3 bis 5 geprüft, ob die goal-Funktion konstant ist. Wenn ja, dann muß dieser Wert kleiner als `bound` sein, sonst wäre der Fall von Zeile 2 eingetreten. Es wird `bound` auf diesen konstanten Wert gesetzt und TRUE zurückgeliefert.
  Als letztes prüfen die Zeilen 6 bis 8, ob die charakteristische Funktion konstant 1 ist. Wenn ja, dann wird `bound` auf das Minimum der goal-Funktion gesetzt und TRUE zurückgegeben.

- *Nachschauen in der ComputedTable:*
  Um die Trefferrate in der ComputedTable zu erhöhen, wird das additive Gewicht a_g der Goal-Funktion auf 0 normiert. Dazu wird die Variable `localBound` eingeführt.
  Ein Eintrag in der ComputedTable besteht aus dem K*BMD für die Goal- und dem OBDD für die charakteristische Funktion sowie einem Feld `entry`. In `entry.bound` wird der Wert von `bound` zur Zeit des Aufrufs gespeichert und in `entry.value` der verbesserte Wert von `bound`, falls ein besserer gefunden wurde, und sonst derselbe Wert wie in `entry.bound`.
  Wenn also `entry.value < entry.bound` ist, dann war der Aufruf letztes Mal erfolgreich. Wenn der damals gefundene Wert kleiner als `localBound` ist, dann haben wir Erfolg gehabt und können `bound` auf `entry.value +` a_g setzen und TRUE zurückgeben. Falls `entry.value` nicht kleiner als `localBound` ist, gibt es keine bessere Lösung als `bound`. Wir geben dann FALSE zurück.
  Es kann jedoch auch noch sein, daß der Aufruf letztes Mal mit einem anderen Wert für `bound` ausgeführt wurde, insbesondere auch mit einem kleineren Wert. Möglicherweise wurde dann keine bessere Lösung gefunden, weil `bound` beim Aufruf bereits zu klein war. In diesem Fall müssen wir weitersuchen.

- *Zerlegen der K*BMDs in Cofaktoren:*
  Wir wollen die K*BMDs nach der Variablen in Cofaktoren aufteilen, die in Goal und Constraint auf dem obersten besetzten Level vorkommt. Diese Variable ermitteln wir in den Zeilen 19 und 20. In den Zeilen 21 bis 24 werden das Goal-K*BMD und das OBDD für die charakteristische Funktion nach der soeben bestimmten Variablen in Cofaktoren aufgeteilt.

- *Rekursiver Aufruf:*
  Wir wollen zuerst den rekursiven Aufruf für die Seite mit dem kleineren Minimum ausführen. Wir hoffen dadurch, auf der anderen Seite möglichst viel durch Basisfälle abdecken zu können.

- *Zusammensetzen der Lösung:*
  Wenn einer der beiden rekursiven Aufrufe ein besseres Ergebnis geliefert hat, dann wird die Schranke angepaßt (Zeile 34) und ein Eintrag in die ComputedTable gemacht (Zeilen 35f).
  Falls keine bessere Lösung gefunden wurde, wird dies ebenfalls in der ComputedTable vermerkt (Zeilen 39f).

Probleme des Algorithmus':
So wie der Algorithmus vorgestellt wurde, gibt es zwei Probleme:

1. Auch wenn die K*BMDs für die linke Seite der Constraints immer lineare Größe haben, kann es passieren, daß die OBDDs für die charakteristischen Funktionen χ_i exponentiell groß werden.

2. Selbst wenn die OBDDs für die χ_i klein sind, kann durch die AND-Verknüpfung ein exponentiell großes OBDD für χ entstehen.

Um das erste Problem zu lösen, führen wir einen Parameter `n_supp` ein. Die Constraints werden nur dann in die charakteristischen Funktionen umgewandelt, wenn sie weniger als `n_supp` Knoten enthalten. Sonst werden sie in Cofaktoren zerlegt. Diese werden kann getrennt behandelt.
Entsprechend verfährt man mit dem zweiten Problem. Nur wenn die Zahl der Knoten in den charakteristischen Funktionen kleiner als ein Parameter `c_size` ist, werden sie zu einem OBDD verknüpft. Ansonsten erfolgt eine Aufteilung in Cofaktoren, und der Algorithmus wird rekursiv auf die beiden Hälften angewendet. Das zweite Problem wird von folgendem Algorithmus gelöst:

3.4 Ilp_min

Algorithmus ilp_min
Input:

- goal – das K*BMD für die Goal-Funktion

- constr – eine Menge von OBDDs für die charakteristischen Funktionen
- LB – eine untere Schranke für die möglichen Lösungen
- UB – der Wert der bisher besten Lösung
- c_size – maximale Anzahl von Knoten in den χ_i.

Output: TRUE, falls eine Lösung gefunden wurde, FALSE sonst.

```
1   if (max(goal) < LB ) return FALSE;
2   if (min(goal) ≥ UB) return FALSE;
3   if (∃ c ∈ constr : Minimize(goal, c, UB)==FALSE) return FALSE;
4   new_constr = conjunction_constr(constr, c_size));
5   if (|new_constr| == 1) return Minimize(goal, new_constr, UB);
6   else {
7     ⟨goal_l, constr_l, goal_h, constr_h⟩ = divide_problem(goal, new_constr);
8     l_ret = ilp_min(goal_l, constr_l, LB, UB, c_size);
9     h_ret = ilp_min(goal_h, constr_l, LB, UB, c_size);
10    return (l_ret || h_ret);
11  }
```

Der Algorithmus für das erste Problem ist ganz analog zu `ilp_min`. Außerdem wandelt `ilp_convert` die K*BMDs für die linken Seiten der Constraints in OBDDs für die charakteristischen Funktionen um.

3.5 Ilp_convert

Algorithmus ilp_convert
Input:

- goal – das K*BMD für die Goal-Funktion
- constr – eine Menge von K*BMDs für die linken Seiten der Constraints
- LB – eine untere Schranke für die möglichen Lösungen
- UB – der Wert der bisher besten Lösung
- c_size – maximale Anzahl von Knoten in den χ_i
- n_supp – maximale Anzahl von Knoten in den K*BMDs von constr

Output: TRUE, falls eine Lösung gefunden wurde, FALSE sonst.

```
1  if (∀ c ∈ constr : size(c) < n_supp) {
2    charakt = { leqZero(c) | c ∈ constr};
```

```
3     return ilp_min(goal, charakt, LB, UB, c_size);
4   } else {
5     ⟨goal_l, constr_l, goal_h, constr_h⟩ = divide_problem(goal, new_constr);
6     l_ret = ilp_convert(goal_l, constr_l, LB, UB, c_size, n_supp);
7     h_ret = ilp_convert(goal_h, constr_h, LB, UB, c_size, n_supp);
8     return (l_ret || h_ret);
9   }
```

4 Heuristiken zur Laufzeitverbesserung

Die in Kapitel 3 angegebenen Algorithmen sind an zwei wichtigen Stellen ungenau. Zum einen sagen sie nichts über die verwendete Variablenordnung aus und zum anderen lassen sie die Reihenfolge, in der die einzelnen charakteristischen Funktionen χ_i mit AND verknüpft werden, offen. Die Wahl, die an diesen Stellen getroffen wird, hat einen großen Einfluß auf die Laufzeit des Algorithmus.

4.1 Auswahl der Variablenordnung

Die Variablenordnung soll so gewählt werden, daß „wichtige“ Variablen möglichst weit vorne in der Ordnung vorkommen.
Dazu haben wir zwei Heuristiken gefunden, die diese Aufgabe erfüllen:

1. Als Heuristik für eine Variable wird die Anzahl der Constraints, in denen diese Variable vorkommt, verwendet:

$$heur_1(x_i) = |\{\chi_j \mid x_i \in \mathrm{supp}(\chi_j)\}|$$

2. Zusätzlich zu der Anzahl der Constraints, in denen eine Variable vorkommt, wird noch berücksichtigt, wieviele andere Variable in dem Constraint vorkommen. Je größer die Supportmenge eines Constraints ist, der von einer Variablen x_i abhängt, desto größer soll der Beitrag zur Heuristik sein.

$$heur_2(x_i) = \sum_{\substack{1 \le j \le m \\ x_i \in \mathrm{supp}(\chi_j)}} 2 - \frac{1}{|\mathrm{supp}(\chi_j)| + 1}$$

Die Variablenordnung wird durch absteigende Sortierung der Variablen nach dem Wert der verwendeten Heuristik festegelegt.
Da erfahrungsgemäß viele Variablen in gleich vielen Constraints vorkommen, verwenden wir die zweite Heuristik, um differenziertere Werte zu erhalten.

4.2 Reihenfolge bei der AND-Verknüpfung

Es können drei verschiedene Heuristiken zur Festlegung der Reihenfolge der Constraints bei der AND-Verknüpfung verwendet werden:

- SizeConjunctionOrder
- SATConjunctionOrder
- SupportConjunctionOrder

Sämtliche Heuristiken zur AND-Verknüpfung sind von einer abstrakten Basisklasse `ConjunctionOrder` abgeleitet, die folgende Methoden zur Verfügung stellt:

- `int size()` Gibt die Anzahl der noch vorhandenen OBDDs zurück.

- `bool empty()` Prüft, ob OBDDs vorhanden sind oder ob die List leer ist.
- `EPtr next()` Liefert ein OBDD zurück, das als Operand bei der nächsten AND-Verknüpfung dienen soll. Es wird automatisch aus der Liste entfernt.
- `EPtr next(EPtr last)` Liefert das nächste OBDD in Abhängigkeit vom als Parameter übergebenen OBDD.
- `void add(EPtr dd)` Fügt das als Parameter übergebene OBDD zur Liste der noch zu verknüpfenden OBDDs hinzu.

Die verschiedenen Heuristiken unterscheiden sich vor allem in der Implementierung von `next`.

4.2.1 SizeConjunctionOrder

Sowohl bei `next()` als auch bei `next(EPtr last)` wird aus der Liste immer das OBDD mit den wenigsten Knoten ausgewählt.

4.2.2 SATConjunctionOrder

Bei beiden `next`-Methoden wird dasjenige OBDD ausgewählt, das die wenigsten erfüllenden Belegungen besitzt, also das OBDD, bei dem die Wahrscheinlichkeit, daß die dargestellte Funktion den Wert 0 annimmt, am größten ist.

4.2.3 SupportConjunctionOrder

Diese Heuristik ist etwas komplizierter als die beiden anderen. Es wurden folgende zwei Punkte berücksichtigt:

1. Zwei OBDDs sollen dann miteinander verknüpft werden, wenn sie von denselben oder zumindest von ähnlichen Variablen abhängen.
2. Verknüpfe nach Möglichkeit OBDDs mit ähnlicher Größe und nicht sehr kleine OBDDs mit sehr großen. Dabei ist die Beobachtung wichtig, daß die anfänglichen OBDDs in der Regel klein sind, aber mit zunehmender Zahl an Verknüpfungen schnell an Größe gewinnen.

Der zweite Punkt wird realisiert, indem wir mit zwei Listen von Constraints arbeiten: aus der ersten Liste werden die OBDDs entnommen, mit AND verknüpft, und das Ergebnis dieser Operation wird in die zweite Liste eingefügt. Ist die erste Liste leer, werden die Listen vertauscht.

Zu jedem Constraint χ_i, der in der ersten Liste enthalten ist, wird die Supportmenge $\mathrm{supp}(\chi_i)$ berechnet, die folgendermaßen definiert ist:

$$\begin{aligned}\mathrm{supp}(f) = \{x_i \mid \exists a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n \in \mathbb{B}^{n-1} : \\ f(a_1, \ldots, a_{i-1}, 0, a_{i+1}, \ldots, a_n) \neq f(a_1, \ldots, a_{i-1}, 1, a_{i+1}, \ldots, a_n)\}\end{aligned}$$

Die Methode `next()` liefert den Constraint aus der ersten Liste zurück, bei dem die Supportmenge die geringste Mächtigkeit hat.

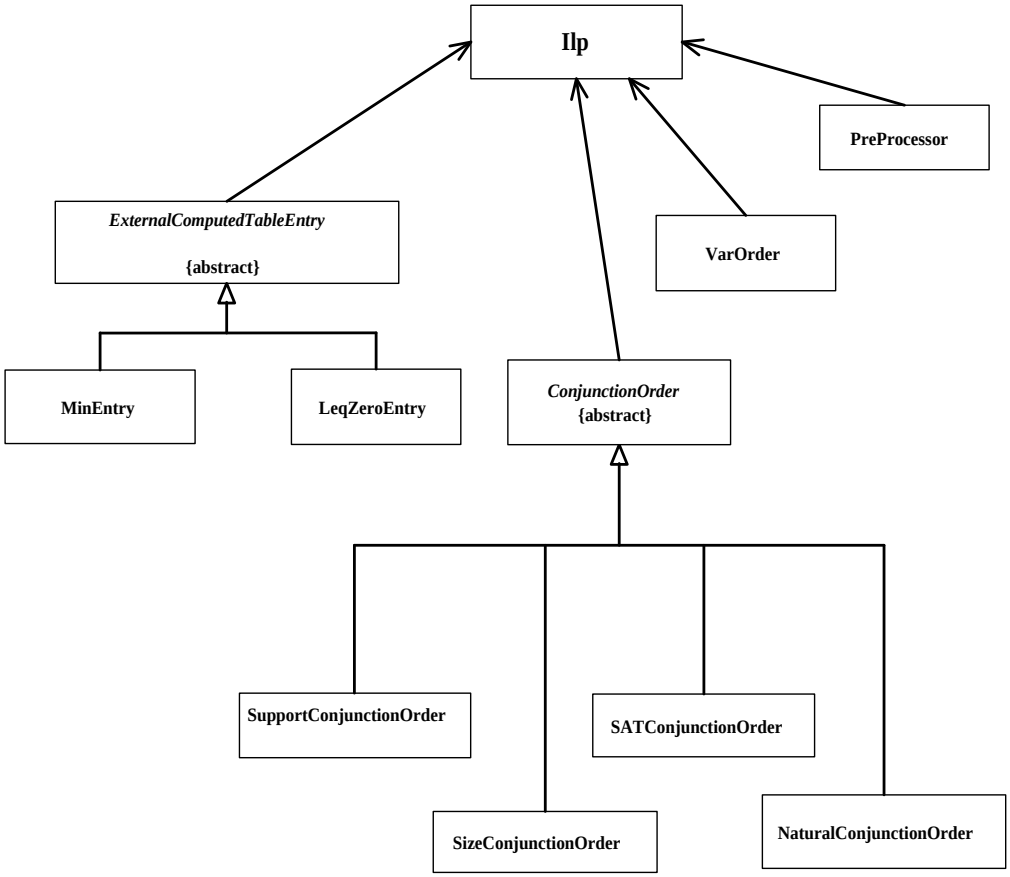


Abbildung 2: Das Klassendiagramm von ILP

`next(EPtr last)` wählt aus der ersten Liste so das OBDD aus, daß es mit dem als Parameter übergebenen OBDD möglichst viel Variablen gemeinsam hat, d. h. daß

$$|\operatorname{supp}(op_1) \cap \operatorname{supp}(op_2)| \quad \rightsquigarrow \quad \max.$$

5 Die Implementierug

Das Ziel der Studienarbeit ist es, die in Kapitel 3 vorgestellten Algorithmen zu implementieren. Die Implementierung des ILP-Solvers in C++ baut auf dem Word-Level-Package von Marc Herbstritt auf. Dieses stellt die notwendigen Funktionen zum Aufbau und zur Verknüpfung von DDs bereit.
Für die Goal-Funktion und die linken Seiten der Constraints werden K*BMDs mit reiner Shannon-Zerlegung verwendet. Die charakteristischen Funktionen werden durch OBDDs dargestellt, da für OBDDs effizientere Algorithmen existieren. Außerdem sind K*BMDs für boolesche Funktionen isomorph zu OBDDs, so daß sich kein Nachteil bezüglich der Größe ergibt.

5.1 Die verwendeten Klassen

Die von mir programmierten Klassen und deren Abhängigkeiten sind im Klassendiagramm in Abbildung 2 dargestellt. Diese Klassen haben die folgende Funktion:

- `VarOrder` dient dazu, eine vernünftige Variablenordnung für die K*BMDs und OBDDs zu finden (siehe Kapitel 4.1).
- `LeqZeroEntry` ist der Typ der Einträge in der ComputedTable, die verwendet wird, um die Umwandlung der Constraints in die charakteristischen Funktionen zu beschleunigen.
- Einträge vom Typ `MinEntry` werden in der ComputedTable von Minimize verwendet, um die Berechnung der Lösung zu beschleunigen.
- `ConjunctionOrder` und alle abgeleiteten Klassen stellen die Implementierungen der verschiedenen Heuristiken zur AND-Verknüpfung der charakteristischen Funktionen dar (siehe Kapitel 4.2).
- `Ilp` ist der eigentliche ILP-Solver. `Ilp` besitzt Methoden zum Aufbau der K*BMDs, zur Umwandlung der Constraints in die charakteristischen Funktionen und zum Finden einer (optimalen) Lösung. Dazu verwendet `Ilp` die anderen Klassen.

Alle in Kapitel 3 vorgestellten Algorithmen sind in der Klasse `Ilp` implementiert. Die Methoden besitzen dieselben Namen wie die Algorithmen, und die Implementiertung folgt möglichst genau den beschriebenen Algorithmen.

6 Eingabedateien und Kommandozeilenparameter

In diesem Abschnitt soll das Format der Eingabedateien für den Ilp-Solver beschrieben werden. Außerdem werden die Kommandozeilenoptionen zum Setzen der Parameter `c_size`, `n_supp` und `conjunctionMode` erläutert.

6.1 Das Standardformat MPS

Das Format MPS für lineare Optimierungsprobleme wurde ursprünglich von IBM entwickelt, dient aber heutzutage für die meisten Programme zur linearen Optimierung als Eingabeformat.
Ich will im Folgenden eine kurze (unvollständige) Beschreibung des Formats angeben, wie sie für binäre lineare Probleme benötigt wird. Für eine vollständige Beschreibung siehe [4].
Eine MPS-Datei besteht aus verschiedenen Abschnitten (in dieser Reihenfolge):

1. NAME
2. ROWS
3. COLUMNS
4. RHS
5. RANGES (optional)
6. BOUNDS (optional)
7. ENDATA

Das Format ist spaltenorientiert. Jede Datenzeile besteht aus 2 bis 6 Feldern, die in bestimmten Spalten stehen müssen:

	Feld 1	Feld 2	Feld 3	Feld 4	Feld 5	Feld 6
Spalten	2-3	5-12	15-22	25-36	40-47	50-61
Inhalt	Bezeichner	Name	Name	Wert	Name	Wert

Im Folgenden werde ich diese Abschnitte einzeln erläutern. Danach folgt ein Beispiel für eine solche Eingabedatei:

1. NAME
   Dieser Abschnitt besteht aus einer Zeile, in der in den Spalten 1-4 NAME und in den Spalten 15-22 der Name des Problems stehen.

2. ROWS
   Dieser Abschnitt beginnt mit dem Wort ROWS in den Spalten 1-4 und enthält für jeden Constraint und die Goal-Funktion genau eine Datenzeile.
   Im Feld 1 (Spalte 2 oder 3) steht der Typ der Ungleichung gemäß folgender Kürzel:

Typ	Kürzel	Bezeichnung
$=$	E	gleich
$\leq$	L	kleiner oder gleich
$\geq$	G	größer oder gleich
	M	Zielfunktion (goal)

   In Feld 2 (Spalten 5-12) steht ein eindeutiger Bezeichner für die Gleichung/Goal-Funktion. Die übrigen Felder sind leer.

3. COLUMNS
   In diesem Abschnitt, der mit dem Wort COLUMNS in den Spalten 1-7 beginnt, werden die Namen der Variablen, die von Null verschiedenen Koeffizienten der Ziel-Funktion und der Constraints definiert.
   Im Feld 2 steht der Name der Variablen, in Feld 3 der Name des Constraints/der Goal-Funktion, in Feld 4 der zugehörige Koeffizient. Falls zu der Variable mehr als ein Koeffizient gehört, kann in derselbe Zeile in Feld 5 ein weiterer Constraint-Bezeichner und in Feld 6 der zugehörige Koeffizient angegeben werden.

4. RHS
   Hier werden die rechten Seiten der Constraints definiert. Dieser Abschnitt wird eingeleitet durch eine Zeile mit RHS in den Spalten 1-3. Dann folgt für jeden Constraint, bei dem die rechte Seite von 0 verschieden ist, eine Datenzeile.
   In Feld 2 steht ein Bezeichner für die rechte Seite, Feld 3 enthält den Namen des Constraints, zu dem die rechte Seite gehört, und Feld 4 den Wert der rechten Seite.
   Die Felder 5 und 6 entsprechen den Feldern 3 und 4.

5. RANGES
   Dieser Abschnitt ist für Constraints der Form:

$$h_i \leq a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq u_i,$$

d. h. es existiert sowohl eine obere als auch eine untere Schranke. Der Bereich des Constraint ist $r_i := u_i - h_i$. Der Wert von u_i oder h_i wird im RHS-Abschnitt angegeben, und der Wert von r_i im Abschnitt RANGES. Daraus läßt sich der andere Wert berechnen. Wenn b_i der Wert im Abschnitt RHS und r_i der Wert im Abschnitt RANGES, dann sind u_i und h_i definiert als:

Typ	Vorzeichen von r_i	untere Schranke h_i	obere Schranke u_i
G	+ oder −	b_i	$b_i + \|r_i\|$
L	+ oder −	$b_i - \|r_i\|$	b_i
E	+	b_i	$b_i + \|r_i\|$
E	−	$b_i - \|r_i\|$	b_i

Der Bereich beginnt mit einer Zeile, die RANGES in den Spalten 1-6 enthält. Die Datenzeilen haben genau dasselbe Format wie im Abschnitt COLUMNS.
Der Abschnitt RANGES ist optional.

6. BOUNDS
   In diesem Abschnitt werden Schranken für die einzelnen Variablen angegeben. Am Anfang des Abschnittes steht BOUNDS in den Spalten 1-6. Ist für eine Variable x_i kein Eintrag vorhanden oder fehlt der BOUNDS-Abschnitt ganz, so wird $0 \leq x_i \leq \infty$ angenommen.
   Für jede Variable kann sowohl eine obere als auch eine untere Schranke angegeben werden. Fehlt ein Wert, so wird dafür der Standardwert (also 0 oder ∞) angenommen.
   Format der Datenzeilen:

   - Feld 1: Typ der Schranke:

LO	untere Schranke	$b_i \leq x_i$
UP	obere Schranke	$x_i \leq b_i$
FX	fester Wert	$x_i = b_i$
FR	freie Variable	$-\infty < x_i < \infty$
MI	untere Schranke $-\infty$	$-\infty < x_i$
PL	obere Schranke ∞	$x_i < \infty$

   - Feld 2: Label für die Schranke
   - Feld 3: Name der Variablen
   - Feld 4: Wert b_i der Schranke
   - Feld 5 und 6: leer

   Da wir uns auf binäre Probleme beschränken, ist für jede Variable als obere Schranke 1 angegeben (die untere Schranke 0 ist der Standard-Wert).

7. ENDATA
   Durch das Wort ENDATA in den Spalten 1-6 wird die MPS-Datei abgeschlossen.

6.1.1 Beispiel

Ein kleines Beispiel soll die Ausführungen des vorigen Abschnitts verdeutlichen:

```
NAME              test
ROWS
 L  c1
```

```
 M  g
COLUMNS
    x1      c1      -2        g       -3
    x2      c1       1        g        2
    x3      c1      -5        g        2
    x4      c1       6
    x5      c1       1        g       -1
    x6      c1      -1        g       -2
RHS
    rhs     c1      -4
BOUNDS
 UP one     x1       1
 UP one     x2       1
 UP one     x3       1
 UP one     x4       1
 UP one     x5       1
 UP one     x6       1
ENDATA
```

Hierdurch wird ein binäres lineares Optimierungsproblem mit einem Constraint (`c1`) und einer Zielfunktion (`g`) dargestellt mit

$$\begin{aligned} c_1: &\quad -2x_1 + x_2 - 5x_3 + 6x_4 + x_5 - x_6 \leq -4 \\ g: &\quad g(x) = -3x_1 + 2x_2 + 2x_3 - x_5 - x_6 \end{aligned}$$

Das MPS-Format ist sehr allgemein. Für binäre Probleme braucht man diese Allgemeinheit nicht. Außerdem ist das Parsen relativ aufwendig. Deshalb habe ich mich entschlossen, ein einfacheres Eingabeformat zu verwenden und einen Konverter für MPS-Dateien in Perl zu schreiben. Durch die Möglichkeit, reguläre Ausdrücke in Perl zu verwenden, war das Parsen bedeutend einfacher als in C++.

6.2 Das vereinfachte Format

Das vereinfachte Format ist ebenfalls zeilenorientiert. In der ersten Zeile stehen, durch Leerzeichen getrennt, zuerst die Anzahl der Variablen und dann die Anzahl der Constraints.
Darauf folgt eine Zeile mit den Namen der Variablen, ebenfalls durch Leerzeichen getrennt.
Es wird vorausgesetzt, daß alle Constraints die Form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n + b_i \leq 0$$

haben. Für jeden Constraint folgt eine Zeile mit den Koeffizienten a_{ij} der Variablen in der Reihenfolge, wie ihre Namen in der zweiten Zeile angegeben sind. Die letzte Ziffer in der Zeile ist der Wert der Konstanten b_i.
Den Schluß macht eine Zeile mit den Koeffizienten der Goal-Funktion im selben Format wie bei den Constraints.

6.2.1 Beispiel

Dasselbe Beispiel wie beim MPS-Format lautet vereinfacht:

```
6 1
x1 x2 x3 x4 x5 x6
-2  1 -5  6  1 -1  4
-3  2  2  0 -1 -2  0
```

6.3 Kommandozeilenparameter

Der Aufruf des Ilp-Solvers hat folgende Syntax:

```
ilp [Optionen] dateiname
```

Als Optionen sind zugelassen (in beliebiger Reihenfolge):

- `-n=<Wert>`
  Hierdurch wird der Wert von `n_supp` auf <Wert> gesetzt. Nur wenn die K*BMDs weniger als `n_supp` Knoten haben, werden sie in die charakteristischen Funktionen umgewandelt. Siehe Seite 14.

- `-c=<Wert>`
  Setzt den Wert des Parameter `c_size` auf <Wert>. Dieser wurde auf Seite 14 beschrieben und dient dazu, die Größe der mit AND zu verknüpfenden charateristischen Funktionen zu begrenzen.

- `-m=size|support|natural|satcount`
  Mit `-m` läßt sich die Heuristik auswählen, mit der der nächste Constraint für die AND-Verknüpfung ausgewählt wird. Dabei bedeuten:

  - `size` SizeConjunctionOrder
  - `support` SupportConjunctionOrder
  - `natural` NaturalConjunctionOrder
  - `satcount` SATConjunctionOrder

  Zu ConjunctionOrder siehe Kapitel 4.

7 Experimentelle Ergebnisse

Ich will in diesem Abschnitt einen Vergleich anstellen zwischen meinem ILP-Solver und dem, der in [5] angegeben ist. Alle Laufzeitmessungen wurden auf einem Athlon-Rechner mit 1 GHz Taktfrequenz und 256 MB RAM durchgeführt. Die Problemstellungen stammen aus MILPLIB, einer Sammlung von ILP Problemen. Da vor allem der Parameter `c_size` großen Einfluß auf die Laufzeit hat, ist sein Wert in der nachfolgenden Tabelle mit angegeben.

Problem	Variablen	Constraints	Lösung	`c_size`	mein ILP-Solver (sec)	FGILP (sec)
bm23	27	20	34	10000	431.44	1509.07
p0033	33	16	3089	10000	0.63	2.91
p0040	40	23	62027	15000	11.12	0.98
stein9	9	13	5	10000	0.03	0.13
stein15	15	36	9	10000	0.21	1.44
stein27	27	118	18	20000	24.92	51.24

Die absoluten Laufzeiten meines ILP-Solvers sind zwar bedeutend kleiner als die in [5] angegeben. Allerdings muß man beachten, daß dort die Messungen auf einem langsameren Rechner durchgeführt wurden. Außerdem gibt es einige Probleme (wie z. B. stein45 und p0201), bei denen mein ILP-Solver gescheitert ist, die aber von Lai, Pedram et. al gelöst werden konnten.

Literatur

[1] Barth, P. A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. Tech. Rep. MPI-I-95-2-003, Max-Planck-Insitut für Informatik, Saarbrücken, 1995.

[2] Becker, B., and Drechsler, R. *Graphenbasierte Funktionsdarstellung.* B. G. Teubner Stuttgart, 1998.

[3] Mehlhorn, K. *Data Structures and Algorithms 2 (Graph Algorithms and NP-Completeness).* Springer Verlag, 1984, p. 198f.

[4] Murtagh, B. A. *Advanced Linear Programming: Computation and Practice.* ?

[5] Y.-T. Lai, M. Pedram, and S. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation and function decomposition. In *IEEE Trans. on CAD, Vol. 13, No. 8* (Aug. 1994), pp. 959–975.