Lehrstuhl für Informatik I
Algorithmen und Komplexität
Universität Würzburg

Würzburg, den 27. Mai 2025

Prof. Dr. Alexander Wolff
Samuel Wolf, M. Sc.

Einführung in IBM ILOG CPLEX Optimization Studio – Teil 2 (unter Verwendung der Kommandozeile)

Im zweiten Teil der CPLEX-Einführung lernen Sie, wie man Konstanten aus Daten-Dateien lädt und wie Sie Summen und Allquantoren in Ihren Modell-Dateien verwenden können.

Dazu bauen wir auf den Inhalten von Teil 1 auf: Sie sollten also bereits wissen, wie Sie sich mit einem CIP-Pool-Rechner verbinden, einfache Modell-Dateien im OPL-Format anlegen und wie Sie eine solche Modell-Datei von IBM ILOG CPLEX lösen lassen.

Diese Anleitung ist zum selbstständigen Durcharbeiten gedacht und wird nicht gesondert in den Übungen besprochen. Sie können allerdings natürlich Ihre Übungsleiter befragen, sollten Teile der Ausführungen hier unklar sein.

1 Anlegen einer Daten-Datei

Für große lineare Programme ist es besser, nicht die vollständige Formulierung in der Modell-Datei zu halten, sondern Daten und Modell sauber zu trennen. Das hat außerdem den Vorteil, dass dasselbe lineare Programm mit unterschiedlichen Daten bzw. Konstanten gelöst werden kann.

Erstellen Sie also im gleichen Ordner, in dem auch Ihre Modell-Datei liegen wird, eine Daten-Datei, welche auf `.dat` endet. Legen Sie also beispielsweise die Datei `meine-daten.dat` an:

```
touch meine-daten.dat
```

Betrachten wir das lineare Programm aus Teil 1, so erkennen wir, dass alle Constraints in der Form

$$c_1^i x_1 + c_2^i x_2 \leq c_3^i$$

dargestellt werden können, wobei i hier das i-te Constraint spezifiziert. Folgende Konstanten lassen sich aus den Constraints extrahieren:

$$\begin{array}{lll} c_1^1 = 4 & c_2^1 = 11 & c_3^1 = 880 \\ c_1^2 = 1 & c_2^2 = 1 & c_3^2 = 150 \\ c_1^3 = 0 & c_2^3 = 1 & c_3^3 = 60 \end{array}$$

Diese Konstanten wollen wir nun in unserer neu angelegten Daten-Datei speichern. Die einfachste Möglichkeit wäre, die Werte einzeln in die Datei zu schreiben – was dann wie folgt aussehen würde:

```
c11 = 4;
c12 = 11;
c13 = 880;
...
```

Zum einen würde die Datei so für größere Modelle schnell sehr lang und unübersichtlich, zum anderen hängen c_1^1, c_2^1 und c_3^1 logisch zusammen und sollten deshalb auch sinnvoll gruppiert gespeichert werden. Dazu verwenden wir hier einfache Arrays wie folgt:

```
c1 = [4, 11, 880];
c2 = [1, 1, 150];
c3 = [0, 1, 60];
```

Nun haben wir drei Arrays, die jeweils alle Konstanten für eines der Constraint beinhalten. Diese Arrays sind nun erneut semantisch verwandt, sodass wir sie gerne unter einem gemeinsamen Eintrag zusammenfassen würden. In CPLEX lassen sich Arrays in der erwarteten Weise schachteln, sodass folgender Code in der Daten-Datei möglich ist:

```
c = [ [4, 11, 880], [1, 1, 150], [0, 1, 60] ];
```

Wie in anderen Sprachen auch, haben Arrays in CPLEX eine feste Länge. Beim späteren Einlesen der Daten in das Modell müssen wir diese Länge kennen. Um uns den Einlesevorgang später zu erleichtern, erweitern wir unsere Daten-Datei also um zwei einfache Konstanten – die Anzahl der gespeicherten Constraints und die Anzahl der Konstanten je Constraint. Außerdem fügen wir noch ein Array und dessen Länge für die Konstanten der Zielfunktion hinzu. Die vollständige Daten-Datei sieht dann so aus:

```
varAnzahl = 2;
conAnzahl = 3;
konstAnzahl = 3;

v = [30, 50];
c = [ [4, 11, 880], [1, 1, 150], [0, 1, 60] ];
```

2 Laden einer Daten-Datei

Die Syntax, um einfache einzelne Variablen aus einer Daten-Datei zu laden, lautet:

```
<typ> <name> = ...;
```

In den Daten wurde für unsere Variablen kein Typ spezifiziert, dies geschieht im Modell. Sie müssen also selbstständig darauf achten, dass die Typen der Werte, die Sie laden wollen auch tatsächlich den vorliegenden Werten der Datei entsprechen. Um

den Variablen ihre richtigen Daten zuzuweisen, werden die Namen der Variablen in beiden Dateien verglichen. Unsere drei Hilfsvariablen laden wir also mit dem Code:

```
int varAnzahl = ...;
int conAnzahl = ...;
int konstAnzahl = ...;
```

Um Arrays zu laden, ergänzt sich die obige Syntax um ein Paar eckige Klammern je Dimension des Arrays. Anders als in den meisten anderen Programmiersprachen beginnen Arrays in CPLEX bei Index 1. Um die Indexbereiche von Arrays zu definieren benutzt CPLEX *Ranges* (Intervalle) der Form `1..n` – in unserem Fall also `1..conAnzahl` und `1..konstAnzahl`. In unserem Beispiel für die beiden Arrays `v` und `c` sieht das also wie folgt aus:

```
int v[1..varAnzahl] = ...;
int c[1..conAnzahl][1..konstAnzahl] = ...;
```

Damit lässt sich nun das alte Modell unter Verwendung dieser Variablen wie folgt umschreiben. Legen Sie hierfür wieder eine Modell-Datei, z. B. `mein-modell.mod`, an:

```
touch mein-modell.mod
```

Speichern Sie den folgenden OPL-Code:

```
int varAnzahl = ...;
int conAnzahl = ...;
int konstAnzahl = ...;

int v[1..varAnzahl] = ...;
int c[1..conAnzahl][1..konstAnzahl] = ...;

dvar float+ x1;
dvar float+ x2;

maximize v[1]*x1 + v[2]*x2;

subject to {
      c[1][1]*x1 + c[1][2]*x2 <= c[1][3];
      c[2][1]*x1 + c[2][2]*x2 <= c[2][3];
      c[3][1]*x1 + c[3][2]*x2 <= c[3][3];
}

execute {
      writeln("x1: ", x1);
      writeln("x2: ", x2);
}
```

Achten Sie darauf, dass in ihren Daten keine Variablen definiert sind, die nicht auch vom Modell gelesen werden wollen. CPLEX geht davon aus, dass alle Daten relevant

sind und wird eine Fehlermeldung ausgeben, wenn Teile der Daten nicht auf Variablen des Modells zugewiesen werden können.

3 Ausführen des Programms

Analog zum Ausführen eines Modells ohne Daten-Datei können Sie mittels

```
oplrun mein-modell.mod meine-daten.dat
```

Ihr Modell mit spezifischen Daten ausführen. Durch das Anlegen mehrerer Daten-Dateien lassen sich so einfach zahlreiche Läufe des gleichen Modells, jedoch mit unterschiedlichen Eingaben bewerkstelligen.

4 Schleifen in CPLEX

4.1 Summen

Die bekannte Schreibweise für Summen $\sum_{i=1}^{n} a_i$ lässt sich mit Hilfe von Ranges oder Mengen (mehr zu Mengen unten) sehr einfach in CPLEX darstellen:

```
sum (i in 1..n) a[i]
```

4.2 Allquantoren

Wie sich das Speichern der Konstanten zu logischen Gruppen zusammenfassen lies, so lassen sich diese auch wieder im Code des Modells abrufen. Zunächst betrachten wir, wie sich die Constraints des obigen linearen Programms durch Quantifizieren zusammenfassen lassen.

$$\bigvee_{i\in\{1,2,3\}} c_1^i x_1 + c_2^i x_2 \leq c_3^i$$

Die Syntax um Quantifizierung in CPLEX zu erreichen lautet wie folgt:

```
forall(<varname> in <Range>) {
      erzeuge Constraints hier
}
```

Die Syntax erinnert an *foreach*-Schleifen, wie sie aus anderen Programmiersprachen wie etwa Java bekannt sind. Um die Laufweite des forall zu definieren bedient sich CPLEX wieder der *Ranges*, wie sie bereits oben beim Indizieren von Arrays benutzt wurden. Alle drei Constraints lassen sich also elegant mittels eines einzigen forall aus den gespeicherten Daten erzeugen:

```
forall(i in 1..conAnzahl){
      c[i][1]*x1 + c[i][2]*x2 <= c[i][3];
}
```

5 Andere Datentypen und -strukturen in CPLEX

5.1 Vergleichsoperatoren und logische Operatoren

Die typischen arithmetischen, vergleichenden und logischen Operatoren sind identisch mit denen in anderen Sprachen wie Java, also z. B.:

x gleich y:	`x == y`
x ungleich y:	`x != y`
x kleiner y:	`x < y`
x größergleich y:	`x >= y`
nicht x:	`!x`
x oder y:	`x || y`
x und y:	`x && y`
wenn x, dann y, sonst z:	`x ? y : z`

5.2 Primitive Datentypen

Neben `int` und `float` für Ganz- und Gleitkommazahlen bietet CPLEX noch `string` an. Diese verwendet man einfach wie die anderen beiden Typen:

```
string vorlesung = "Algorithmische Graphentheorie";
```

5.3 Entscheidungsvariablen

Entscheidungsvariablen in CPLEX sind ihrer Natur nach stets einfache numerische Werte. Boolsche Werte stehen nur als Entscheidungsvariablen (`dvar`) zur Verfügung. Diese werden mit dem Schlüsselwort `boolean` initialisiert und sind äquivalent zu den Ganzzahlen $\{0, 1\}$ – entsprechend lassen sich auch Rechenoperationen mit `booleans` durchführen. Wie erwartet entspricht `0` dem Wert `false` und `1` dem Wert `true`.

```
dvar boolean wahrheitswert;
```

Oftmals hängt die Anzahl der benötigten Entscheidungsvariablen von der Größe der Eingabe ab. Um diese Fälle in CPLEX darstellen zu können, lassen sich Entscheidungsvariablen in (mehrdimensionalen) Arrays anlegen.

```
dvar <Typ> <name>[<Range>];
```

Für unser einfaches Beispiel lässt sich folgender Code verwenden:

```
dvar float+ x[1..varAnzahl];
```

5.4 Tupel

Tupel sind definierbare Strukturen, um logisch zusammenhängende Daten verschiedenen Typs zu speichern – ähnlich wie structs in C. Sie lassen sich benennen und können beliebige andere Datentypen und -strukturen beinhalten. Definiert werden Tupel im Modell, speichern lassen sich ihre Werte in Daten-Dateien. Unser einfaches lineares Programm lässt sich unter Verwendung eines Tupels für die Constraint-Konstanten wie folgt umschreiben:

Modell:

```
dvar float+ x1;
dvar float+ x2;

int varAnzahl = ...;
int v[1..varAnzahl] = ...;

tuple ConstraintWerte {
      int x1Wert;
      int x2Wert;
      int groesserWert;
}

ConstraintWerte erstes = ...;
ConstraintWerte zweites = ...;
ConstraintWerte drittes = ...;

maximize v[1]*x1 + v[2]*x2;

subject to {
      erstes.x1Wert*x1 + erstes.x2Wert*x2 <= erstes.groesserWert;
      zweites.x1Wert*x1 + zweites.x2Wert*x2 <= zweites.groesserWert;
      drittes.x1Wert*x1 + drittes.x2Wert*x2 <= drittes.groesserWert;
}

execute {
      writeln("x1: ", x1);
      writeln("x2: ", x2);
}
```

Daten:

```
varAnzahl = 2;
v = [30, 50];

erstes = <4, 11, 880>;
zweites = <1, 1, 150>;
drittes = <0, 1, 60>;
```

Beachten Sie, wie mit dem . -Operator auf die Inhalte der Tupel zugegriffen wird und dass sich Tupel durch Spitzklammern < , > initialisieren lassen. Beim Initialisieren werden die angebenen Werte der Reihe nach auf die Elemente des Tupels zugewiesen, achten Sie also auf Reihenfolge und Typen.

5.5 Mengen (*Sets*)

In CPLEX ist eine Menge eine nicht-indizierte Sammlung von Elementen ohne Duplikate. Standardmäßig sind in CPLEX Mengen geordnet nach Reihenfolge des Hinzufügens und alle Operationen, die CPLEX zum Zugriff auf Mengen bereitstellt, werden diese Ordnung nicht ändern. Um eine Menge zu erzeugen werden geschweifte Klammern { , } verwendet.

```
{int} primzahlen = {2,3,5,7,11};
```

Es ist nicht möglich eine Menge aus anderen Datenstrukturen zu bilden, etwa Arrays oder weiteren Mengen. Was jedoch möglich ist, ist eine Menge aus Tupeln. Analog zu verschachtelten Arrays lässt sich der obige Code für Tupel unter Verwendung von Quantifizierung umschreiben. Im folgenden Code verwenden wir zusätzlich eine variable Anzahl an Entscheidungsvariablen und eine Summation für die Zielfunktion.

Modell:

```
int varAnzahl = ...;
dvar float+ x[1..varAnzahl];
int v[1..varAnzahl] = ...;

tuple ConstraintWerte {
      int x1Wert;
      int x2Wert;
      int groesserWert;
}

{ConstraintWerte} c = ...;

maximize sum (i in 1..varAnzahl) v[i]*x[i];

subject to {
      forall (werte in c) {
```

Daten:

```
varAnzahl = 2;
v = [30, 50];

c = { <4, 11, 880>, <1, 1, 150>, <0, 1, 60> };
```

Mengen haben eine variable Größe, sodass weder beim Einlesen noch beim Durchlaufen einer Menge deren Länge bekannt sein muss (anders als bei Arrays). Beachten Sie, dass Sie einfach mittels `foreach (werte in c)` durch alle Tupel iterieren können. Um auf das i-te Element einer Menge zugreifen zu können, können sie auch die `item(<Menge>,<index>)` -Funktion verwenden. Im obigen Beispiel liefert der Aufruf `item(c,2).groesserWert` die Zahl 60, da Mengen anders als Arrays ab 0 gezählt werden.

Summen über Mengen lassen sich wie folgt einfacher in CPLEX umsetzen:

```
sum (element in set) element
```