
Angewandte Optimierung mit IBM ILOG CPLEX Optimization Studio

Modellierung von Planungs- und Entscheidungsproblemen
des Operations Research mit OPL

Lösungen zu den Aufgaben an den Kapitelenden

Stefan Nickel
Claudius Steinhardt
Hans Schlenker
Wolfgang Burkart
Melanie Reuter-Oppermann

Stand: Mai 2020

Aufgaben Kapitel 3 – Lösung

1. Die vier Hauptbestandteile einer Modelldatei sind die Deklaration der Modellparameter, die Deklaration der Entscheidungsvariablen, die Zielfunktion sowie die Nebenbedingungen des Modells.
2. Durch die Deklaration wird dem Parameter ein Datentyp zugewiesen. Anschließend kann der Parameter initialisiert werden, d. h. ihm wird ein konkreter Wert zugewiesen. Die Deklaration muss zwingend in der Modelldatei erfolgen, die Initialisierung von Parametern kann in einer separaten .dat-Datei erfolgen.
3. Ganzen Zahlen wird der elementare Datentyp int zugewiesen. Der Datentyp float wird für die Darstellung von reellen Zahlenwerten verwendet. Zeichenketten werden mit dem Datentyp string gekennzeichnet.
4. Ein Entscheidungsausdruck ist ein separater in der Modelldatei definierter Term. Er wird mit dem Schlüsselwort dexpr gekennzeichnet. Innerhalb der Modelldatei kann auf den Term mittels seines Namens zugegriffen werden. Insbesondere bei häufig wiederkehrenden, komplexen Bestandteilen einer Zielfunktion oder einer Nebenbedingung kann durch den Einsatz von Entscheidungsausdrücken die Fehleranfälligkeit reduziert und die Übersichtlichkeit verbessert werden.
5. Die Zielfunktion ist eine mathematische Funktion, die die Zielsetzung des Optimierungsproblems beschreibt. Die unabhängigen Variablen der Funktion sind die Entscheidungsvariablen. Nebenbedingungen sind mathematische Gleichungen bzw. Ungleichungen, welche mit denen die Restriktionen des Modells beschrieben werden. In ILOG können Nebenbedingungen auch logische Ausdrücke sein, welche im Vorfeld des Lösungsvorgangs im Hintergrund als lineare Ausdrücke ausgedrückt werden.
6. Die Fehler sind rot markiert:

```
//1. Deklaration der Modellparameter
float anlaufkosten1 = 3.0; //fehlende Deklaration
float anlaufkosten2 = 0.0; //fehlende Deklaration
float gewinn1 = 2.0; //falscher Datentyp (optional int gewinn1 = 2;)
float gewinn2 = 3.0; //falscher Datentyp (optional int gewinn2 = 3;)
float kapa1 = 0.2;
float kapa2 = 0.4;
int maxAnzahl1 = 3; //Zuweisung eines Wertes, keine Gleichung
int maxAnzahl2 = 2; //Zuweisung eines Wertes, keine Gleichung

//2. Deklaration der Entscheidungsvariablen
dvar int+ x1; //Anzahl Produkte für Kunde 1
dvar int+ x2; //Anzahl Produkte für Kunde 2
dvar boolean y1; //fehlende Kennzeichnung als Entscheidungsvariable
dvar boolean y2; //fehlende Kennzeichnung als Entscheidungsvariable

//3. Zielfunktion
maximize (gewinn1*x1 - anlaufkosten1*y1) + (gewinn2*x2 -
        anlaufkosten2*y2); //falsches Schlüsselwort

//4. Nebenbedingungen
subject to {
```

```

    kap1*x1 + kap2*x2 <= 1; /*unzulässiger Operator (mögliche Lösung:
                                von rechter Seite eine hinreichend kleine,
                                echt positive Zahl abziehen*/
    x1 <= maxAnzahl1*y1;
    x2 <= maxAnzahl2*y2;
}

```

7. Die Deklaration der Modellparameter und der Entscheidungsvariablen bleibt gegenüber Beispiel 3.16 unverändert. Um die in der Aufgabenstellung genannten Anforderungen an die Zielfunktion und die Nebenbedingungen zu erfüllen, sind vier Entscheidungsausdrücke zu definieren.

```

dexpr float gesamtdb = db1*menge1 + db2*menge2; //Gesamtdeckungsbeitrag
dexpr float zeitbedarf1 = bedarf11*menge1 +
    bedarf21*menge2; //Gesamtzeitbedarf auf Maschine M1
dexpr float zeitbedarf2 = bedarf12*menge1 +
    bedarf22*menge2; //Gesamtzeitbedarf auf Maschine M2
dexpr float zeitbedarf3 = bedarf13*menge1 +
    bedarf23*menge2; //Gesamtzeitbedarf auf Maschine M3

```

Die Entscheidungsausdrücke ersetzen die jeweiligen Terme an den entsprechenden Stellen in der Zielfunktion und in den Nebenbedingungen.

```

//3. Zielfunktion
maximize gesamtdb; //DB-Maximierung

//4. Nebenbedingungen
subject to{
    zeitbedarf1 <= kap1; //Maschinenzeitrestriktion M1
    zeitbedarf2 <= kap2; //Maschinenzeitrestriktion M2
    zeitbedarf3 <= kap3; //Maschinenzeitrestriktion M3
    menge1 <= nf1; //Nachfragerestriktion Fahrradtyp 1
    menge2 <= nf2; //Nachfragerestriktion Fahrradtyp 2
}

```

8. Der maximale Gewinn wird durch die Produktion (und den Verkauf) von 20 Einheiten von Fahrradtyp 1 und 200 Einheiten von Fahrradtyp 2 erzielt. Setzt man diese Werte in die Maschinenzeitrestriktionen (Nebenbedingungen 1 bis 3) ein, stellt man fest, dass lediglich die Kapazität von Maschine 3 komplett ausgeschöpft wird.

```

0.2*20 + 0.4*200 = 84 < 120 // verbleibende Kapazität auf M1: 36
0.3*20 + 0.2*200 = 46 < 140 // verbleibende Kapazität auf M2: 94
0.5*20 + 0.5*200 = 110 // verbleibende Kapazität auf M3: 0

```

Für die Erfüllung der gesamten Nachfrage nach Typ 3 in Höhe von 100 Einheiten sind $0.3 \cdot 100 = 30$ Kapazitätseinheiten auf Maschine 1 und $0.9 \cdot 100 = 90$ Kapazitätseinheiten auf Maschine 2 erforderlich. Da Maschine 3 nicht benötigt wird und die erforderlichen Maximalkapazitäten geringer als die Restkapazitäten der jeweiligen Maschinen sind, wird die gesamte Nachfrage nach Fahrradtyp 3 erfüllt.

9. Die Erweiterungen sind rot markiert:

```

//1. Deklaration der Modellparameter
int nf1 = 300; //Nachfrage nach Fahrradtyp 1
int nf2 = 200; //Nachfrage nach Fahrradtyp 2
int nf3 = 100; //Nachfrage nach Fahrradtyp 3
int kap1 = 120; //Gesamtmaschinenzeit von M1 im Planungszeitraum
int kap2 = 140; //Gesamtmaschinenzeit von M2 im Planungszeitraum

```

```

int kap3 = 110;           //Gesamtmaschinenzeit von M3 im Planungszeitraum

float bedarf11 = 0.2; //Bedarf von Fahrradtyp 1 an Maschinenzeit auf M1
float bedarf12 = 0.3; //Bedarf von Fahrradtyp 1 an Maschinenzeit auf M2
float bedarf13 = 0.5; //Bedarf von Fahrradtyp 1 an Maschinenzeit auf M3
float bedarf21 = 0.4; //Bedarf von Fahrradtyp 2 an Maschinenzeit auf M1
float bedarf22 = 0.2; //Bedarf von Fahrradtyp 2 an Maschinenzeit auf M2
float bedarf23 = 0.5; //Bedarf von Fahrradtyp 2 an Maschinenzeit auf M3
float bedarf31 = 0.3; //Bedarf von Fahrradtyp 3 an Maschinenzeit auf M1
float bedarf32 = 0.9; //Bedarf von Fahrradtyp 3 an Maschinenzeit auf M2
float bedarf33 = 0.0; //Bedarf von Fahrradtyp 3 an Maschinenzeit auf M3
float db1 = 110;         //Deckungsbeitrag je Einheit von Fahrradtyp 1
float db2 = 140;         //Deckungsbeitrag je Einheit von Fahrradtyp 2
float db3 = 90;          //Deckungsbeitrag je Einheit von Fahrradtyp 3

//2. Deklaration der Entscheidungsvariablen
dvar int+ menge1; //Geplante produzierte Mengeneinheiten von Fahrradtyp 1
dvar int+ menge2; //Geplante produzierte Mengeneinheiten von Fahrradtyp 2
dvar int+ menge3; //Geplante produzierte Mengeneinheiten von Fahrradtyp 3

//3. Zielfunktion
maximize db1*menge1 + db2*menge2 + db3*menge3; //DB-Maximierung

//4. Nebenbedingungen
subject to{
    //Zeitrestriktion M1
    bedarf11*menge1 + bedarf21*menge2 + bedarf31*menge3 <= kap1;
    //Zeitrestriktion M2
    bedarf12*menge1 + bedarf22*menge2 + bedarf32*menge3 <= kap2;
    //Zeitrestriktion M3
    bedarf13*menge1 + bedarf23*menge2 + bedarf33*menge3 <= kap3;
    menge1 <= nf1; //Nachfragerestriktion Fahrradtyp 1
    menge2 <= nf2; //Nachfragerestriktion Fahrradtyp 2
    menge3 <= nf3; //Nachfragerestriktion Fahrradtyp 3
}

```

Mit einer optimalen Lösung $\text{menge1} = 20$, $\text{menge2} = 200$ und $\text{menge3} = 100$ ergibt sich der optimale Zielfunktionswert 39 200 [EUR].

10. Der Aufwand für die Erweiterung des Modells ist sehr hoch. Fünf neue Modellparameter sind dem Modell hinzuzufügen, eine weitere Entscheidungsvariable ist zu definieren. Zudem sind alle Kapazitätsrestriktionen zu erweitern und eine weitere Nachfragerestriktion ist zu ergänzen. Eine weitaus einfachere Möglichkeit der Erweiterung bietet die Verwendung von Feldern, wie sie in Kapitel 4.2 eingeführt werden.

Aufgaben Kapitel 4 – Lösung

1. Mengen und Felder sind Sammlungen von Elementen, die sich in ihren Datentypen gleichen. Im Gegensatz zu Feldern kann in einer Menge jedes Element maximal höchstens einmal auftreten; Duplikate sind nicht zulässig. Feldelemente können durch Indizes direkt referenziert werden, bei Mengenelementen ist dies nicht möglich. Lediglich Felder können als Entscheidungsvariablen in einem OPL-Modell genutzt werden.
2. Zusammengesetzte Datentypen reduzieren den Aufwand bei der Erstellung der Modelldatei. Durch die Zusammenfassung ähnlich aufgebauter Gleichungen und Terme wird zudem die Lesbarkeit des OPL-Modells erhöht. Die Nutzung von zusammengesetzten Datentypen ermöglicht die Erstellung von generischen Modelldateien, in denen ein Teil der Parameter zum Zeitpunkt der Modellerstellung noch nicht bekannt ist. Sobald diese bekannt sind, können sie an einigen Stellen in das Modell eingebaut werden. Auf diese Weise können ohne größeren Aufwand zusätzlich Anpassungen der Modellparameter vorgenommen werden.
3. Wird das Schlüsselwort `ordered` bei der Definition einer Menge verwendet, werden die innerhalb der geschweiften Klammern aufgezählten Elemente genau der Reihenfolge abgespeichert, wie sie innerhalb dieser Klammern gelistet werden. Im Gegensatz dazu wird eine Menge, welche mit dem Schlüsselwort `sorted` gekennzeichnet wird, anhand ihrer natürlichen Ordnung abgebildet.
4. Bei der Initialisierung von expliziten Feldern werden die einzelnen Feldelemente explizit angegeben. Sie sind unabhängig vom Laufindex, mithilfe dessen sie referenziert werden können. Generische Felder nutzen die Laufindizes des Feldes für die Initialisierung. Die Laufindizes sind bei der Initialisierung explizit zu definieren, sodass diese anschließend auf der rechten Seite als Platzhalter für mathematische bzw. logische Ausdrücke genutzt werden können.
5.

```
int feldA [i in 1..11] = 3*(i - 1) - 1;
int feldB [I in 1..11] = i*i + 3;
int feldC [1..11] = [3*i - 1: 2*i + 1 | i in 1..4];
int feldD [I in 1..11] = 5*(i mod 4) - 3;
int feldE [1..11] = [2*(i mod 5) - 1: i*i | i in 1..4];
```
6. a) `mengeA = (menge1 union menge2) inter menge3;`
b) `mengeB = (menge2 symdiff menge3) inter menge1;`
7. a) `zahlA = 5;`
b) `arrayB = [20, 17, 15, 13, 8];`
c) `mengeC = {0, 1, 2, 3, 5, 6}`
8. Modell-Datei:

```
//1. Deklaration der Modellparameter
int nMedien = 3;
range medien = 1..nMedien;
int reichweite[medien] = [20000, 120000, 9000];
int kosten[medien] = [15000, 6000, 4000];
int kapazitaetMedium[medien] = [4, 10, 7];
int kapazitaetAgentur = 15;
int budget = 100000;
```

```

//2. Deklaration der Entscheidungsvariablen
dvar int+ nSchaltung[medien];

//3. Zielfunktion
maximize sum(i in medien) reichweite[i]*nSchaltung[i];

//4. Nebenbedingungen
subject to {
    sum(i in medien) kosten[i]*nSchaltung[i] <= budget;
    forall(i in medien){
        nSchaltung[i] <= kapazitaetMedium[i];
    }
    sum(i in medien) nSchaltung[i] <= kapazitaetAgentur;
}

```

Um die maximal mögliche Anzahl an Menschen zu erreichen, sollte die Werbeagentur zwei Radiospots senden und auf zwei Bahnen werben. Zudem sollten auf zehn Webseiten Anzeigen geschaltet werden. Auf diesem Weg erreicht die Kampagne 1 258 000 Personen.

9. Modell-Datei:

```

//1. Deklaration der Modellparameter
int nAnfragen = 3;
range Anfragen = 1..nAnfragen;
int kosten[Anfragen] = [300, 200, 0];
int gewinn[Anfragen] = [200, 300, 80];
float kapazitaetsverbrauch[Anfragen] = [0.02, 0.04, 0.02];
int maxAnzahl[Anfragen] = [30, 20, 50];

//2. Deklaration der Entscheidungsvariablen
dvar int+ anzahlRaeder[Anfragen];
dvar boolean wirdAngenommen[Anfragen];

//3. Zielfunktion
maximize sum(i in Anfragen) (gewinn[i]*anzahlRaeder[i] -
    kosten[i]*wirdAngenommen[i]);

//4. Nebenbedingungen
subject to {
    sum(i in Anfragen) anzahlRaeder[i]*kapazitaetsverbrauch[i] <= 1;
    forall(i in Anfragen) anzahlRaeder[i] <= maxAnzahl[i]*
        wirdAngenommen[i];
}

```

RideEasy sollte die Kundenanfragen 1 und 2 annehmen und die Anfrage des Kunden 3 ablehnen. Gefertigt werden 30 Räder für Kunde 1 und 10 Räder für Kunde 2. Der Gesamtgewinn abzüglich der Entwicklungskosten beträgt 8 500 [EUR].

10. Modell-Datei:

```

//1. Deklaration der Modellparameter
int nFabriken = 3;
int nKunden = 3;
range Fabriken = 1..nFabriken;
range Kunden = 1..nKunden;
int transportkosten[Fabriken][Kunden] = [[10, 15, 12], [17, 14, 20],
    [15, 10, 11]];

int kapazitaet[Fabriken] = [1800, 1400, 1300];
int nachfrage[Kunden] = [1200, 800, 800];
int eroeffnungskosten[Fabriken] = [8000, 10000, 9000];

```

```

//2. Deklaration der Entscheidungsvariablen
dvar int+ transportmenge[Fabriken][Kunden];
dvar boolean fabrikOffen[Fabriken];

//3. Zielfunktion
minimize sum(i in Fabriken, j in Kunden) (transportkosten[i][j]*
    transportmenge[i][j]) + sum(i in Fabriken) (eroeffnungskosten[i]*
    fabrikOffen[i]);

//4. Nebenbedingungen
subject to {
    forall(j in Kunden)
        sum(i in Fabriken) transportmenge[i][j] == nachfrage[j];
    forall(i in Fabriken)
        sum(j in Kunden) transportmenge[i][j] <= kapazitaet[i] *
        fabrikOffen[i];
}

```

Die Standorte 1 und 3 werden eröffnet, die Gesamtkosten belaufen sich auf 46 100 [EUR].

Aufgaben Kapitel 5 – Lösung

1. OPL-Datenelemente können im OPL-Modell sowie in ILOG Script verwendet werden. In ILOG Script besteht die Möglichkeit des Zugriffs auf die Datenelemente des umgebenden OPL-Modells unmittelbar über ihre OPL-Bezeichnung. Zusätzlich zu den OPL-Datenelementen können in einem ILOG-Script-Block auch neue Datenelemente erzeugt werden, diese heißen ILOG-Script-Variablen. Dabei handelt es sich aber nicht um OPL-Entscheidungsvariablen, sondern lediglich um lokale Datenelemente. Auf sie kann aus OPL heraus nicht zugegriffen werden.

2. In OPL: `card(set)`
In ILOG Script: `set.size` oder `opl.card(set)`

3. Die Fehler sind rot markiert:

```
{int} produkte = {1, 2, 3, 4, 5};
int anzahl = card(produkte);

int menge[produkte] = [3 ,7, 6, 5, 4]; //Falscher Laufbereich

execute Vorabberechnungen{
    var zaehler = 0;
    for (i in produkte) //Falscher Laufbereich
        zaehler += menge[i];

    if (anzahl < 25)
        writeln("Nur wenige Produkte sind zu beachten.");
    else if (anzahl = 25) //Anführungszeichen fehlt*/
        writeln("Es sind genau 25 Produkte zu beachten.");
    else
        writeln("Es sind viele Produkte zu beachten.");

    writeln("Die durchschnittliche Menge pro Produkt beträgt: "
        + (zaehler/anzahl));
        /*Anführungszeichen an falscher Stelle,
        Konkatenation fehlt*/
    writeln("Teile die Anzahl der Produkte auf 5 neue Produkte"
        + " zu gleichen Teilen auf.");
    produkte.clear();

    var abbruch = false; //Zuweisung eines Wertes, keine Gleichung
    var neu = 0;
    while (abbruch != true){ //Falsche Schreibweise der Ungleichheit
        neu++;
        produkte.add(neu); //Falsche Anwendung der Funktion „add“ um ein
            //neues Element hinzuzufügen*/
        if (produkte.size == anzahl){ //Befehl für OPL, nicht ILOG Script
            abbruch = true;
        }
    }
    writeln();
    writeln("Produkt    Anzahl");
    writeln("_____");
    for (var i=1; i<=anzahl; i++){
        menge[i] = zaehler/anzahl;
        writeln("    ",i,"    ",menge[i]); /*hier soll die Anzahl ausgegeben
```

```

    }
}

```

werden, deshalb muss das Feld „menge“ verwendet werden*/

Ausgabe:

Nur wenige Produkte sind zu beachten.
 Die durchschnittliche Menge pro Produkt beträgt: 5
 Teile die Anzahl der Produkte auf 5 neue Produkte zu gleichen Teilen auf.

Produkt	Anzahl
1	5
2	5
3	5
4	5
5	5

4. Funktionen aus OPL in ILOG Script lassen sich durch vorangestelltes „opl.“ nutzen. Funktionen aus ILOG Script lassen sich in OPL nicht nutzen.
5. Folgende Möglichkeiten wurden beschrieben:
 1. `for(s in set) {...}` (Für jedes Element s in der Menge set wird der Block {...} ausgeführt)
 2. `for(var i=n; i<k; i++) {...}` (Für jedes Element i von i=n bis k (ohne k) führe {...} aus)
 3. `while(bedingung) {...}` (Solange bedingung wahr ist führe {...} aus)

Aufgaben Kapitel 6 – Lösung

1. Als Tupeltyp werden die im Modell definierten strukturierten Datentypen verstanden, während die konkreten Instanzen dieses Tupeltyps als Tupel bezeichnet werden.
2. Sowohl elementare Datentypen (z.B. float, int, string) als auch komplexe Datentypen können in Tupeln verwendet werden. Ebenso ist eine Verwendung von anderen Tupeln möglich, was zu einer Verschachtelung dieser führt. Entscheidungsvariablen können nicht in Tupel-Typen verwendet werden. Stattdessen empfiehlt es sich für die Entscheidungsvariable ein Feld zu definieren, welches über die entsprechende Tupel-Menge läuft.
3. In OPL deklarierte, aber noch nicht initialisierte Tupel werden zunächst mit den Default-Werten belegt. Während in OPL die Initialisierung mit der „< >“-Notation in einer Zeile erfolgt, ist in ILOG Script jedes einzelne Attribut unter Verwendung des Punkt-Operators zu initialisieren.
4. Tupel in OPL unterliegen der Referenzsemantik. Durch die Verwendung des „=-“-Operators zwischen zwei deklarierten Tupeln wird lediglich dem links stehenden Tupel die Referenz des rechten Tupels zugewiesen. Beide Platzhalter zeigen auf dasselbe Tupel und Änderungen eines Elements werden somit in beiden Tupeln sichtbar.
5. Eine explizite Funktion für diese Problematik existiert in OPL nicht. Allerdings kann mit der Funktion first() stets auf das erste Element einer Menge zugegriffen werden. Existiert nur ein Element in der gefilterten Menge ist das Ergebnis dieser Funktion eindeutig.
6. Slicing bezeichnet das verschachtelte Durchlaufen einer Tupel-Menge mit definierten Filterbedingungen. Hierbei werden Tupel oder Tupel-Attribute eines inneren Laufbereichs in Beziehung zu einem entsprechenden äußeren Laufbereich der Menge gesetzt. Besteht eine Gleichheitsbeziehung ermöglicht Slicing eine effiziente und übersichtliche Modellerstellung. Slicing kommt u. a. bei der Initialisierung generische Felder oder der Bildung neuer Mengen anhand von Filterkriterien zum Einsatz.
7. Ausgabe:

```
<"Marmelade" <"Glas" 300> 500>
300
<"Plastikbeutel" 20>
Plastikflasche
300
700
500
```

8. Ausgabe:

```
20
30
{<"Wasser" <"Plastikflasche" 30> 1500 1> <"Wasser" <"Glas" 300> 700 1>}
Marmelade in Glas
Ei in Karton
Apfel in Plastikbeutel
Wasser in Glas
<"Plastikflasche" 30>
```

9. Es bestehen keine Tupel-Referenzen. Die Beziehungen zwischen den Tupel-Typen werden lediglich über die Schlüsselattribute nachgebaut. Folgende Modellierung wäre an dieser Stelle effizienter:

```
tuple Produktionsprozess {  
    key Produkt;  
    key Maschine;  
}  
  
{Produktionsprozess} produktionsProzesse  
    with produkt in produkte, maschine in maschinen = {  
        <<"X-1", 5>, <"M-1", 4>>, <<"X-1", 5>, <"M-3", 8>>, <<"Y-2", 10>,  
        <"M-3", 8>>, <<"Y-2", 10>, <"M-4", 13>>, <<"Z-3", 25>, <"M-2", 7>>  
    };
```

10. Modellierung mit implizitem Slicing:

```
// Implizites Slicing  
int durchlaufZeitImplizit[p in produkte] =  
    sum(<p, m> in produktionsProzesse) (m.produktionszeit * p.loggroesse);
```

Aufgaben Kapitel 7 – Lösung

1. Daten können in OPL intern oder extern initialisiert werden. Bei der internen Initialisierung werden alle Daten-Elemente in der Modell-Datei deklariert und initialisiert. Es existiert nur diese eine Datei. Die Initialisierung in einer oder mehreren separaten Dateien bezeichnet man als externe Initialisierung. Die Deklaration erfolgt weiterhin in der Modell-Datei.
2. Ja, die Initialisierung der Daten-Elemente kann in verschiedenen Daten-Dateien erfolgen. Allerdings ist sicherzustellen, dass jedes Element nur einmal initialisiert wird, d. h. dass die Daten-Dateien disjunkt sind.
3.
 1. Externe Initialisierungen
 2. Ausführung aller execute-Blöcke im Modell (entsprechend ihrer Reihenfolge im Modell)
 3. Lazy Initialization

Unter Lazy Initialization versteht man das Initialisieren von internen Daten-Elementen zum spätestmöglichen Zeitpunkt, d. h. beim Zugriff in einem execute-Block oder einer nachfolgenden internen Initialisierung.
4. In der Daten-Datei kann ein ganzzahliges Daten-Element initialisiert werden, welches in der Modell-Datei deklariert wird. Dieses Daten-Element kann nun als Grenze bei der Definition des Zahlenbereichs in der Modell-Datei verwendet werden.
5. Das Schreiben in eine Excel-Datei ist nur möglich, wenn diese nicht zur Laufzeit geöffnet ist. Zudem muss die Datei auf dem ausführenden Rechner gespeichert sein.

6. Ausgabe:

```
12 5 0
0 29
116
15 29
```

7. Modell-Datei:

```
int nLager = ...;
int nKunden = ...;
int nProdukte = ...;

range Lager = 1..nLager;
range Kunden = 1..nKunden;
range Produkte = 1..nProdukte;

float preis[Produkte] = ...;
int transportkostenFix[Lager][Kunden] = ...;
int nachfrage[Kunden][Produkte] = ...;

tuple Kunde {
    key int kundenID;
    string ort;
}

{Kunde} kunden = ...;
```

8. Daten-Datei:

```
SheetConnection xls("Variablen.xlsx");

nProdukte from SheetRead(xls, "'Szenario1'!G9");
nLager from SheetRead(xls, "'Szenario1'!G10");
nKunden from SheetRead(xls, "'Szenario1'!G11");

kunden from SheetRead(xls, "'Szenario1'!A3:B5");
preis from SheetRead(xls, "'Szenario1'!Preis");
nachfrage from SheetRead(xls, "'Szenario1'!Nachfrage");
transportkostenFix from SheetRead(xls, "'Szenario1'!B9:D11");

kumulierteTransportmenge to SheetWrite(xls, "'Szenario1'!B15:D17");
```

Aufgaben Kapitel 8 – Lösung

1. Aussagen, die über logische Operatoren dargestellt sind, nehmen entweder den Wert „wahr“ oder „falsch“ an. Das Prinzip des ausgeschlossenen Dritten besagt, dass es außer den beiden Wahrheitswerten keinen weiteren Zustand für eine Aussage gibt. Ferner kann eine Aussage nicht gleichzeitig beide Wahrheitswerte annehmen (Prinzip des ausgeschlossenen Widerspruchs).
2. a) Mögliche Formulierung:
$$(A == 1) \Rightarrow ((B == 1) \ \&\& \ (C == 1))$$

Alternative Formulierung: Die allgemeine Implikation „wenn X, dann Y“, kann auch folgendermaßen formuliert werden: „wenn nicht Y, dann auch nicht X“: $(!Y \Rightarrow !X)$

$$!((B == 1) \ \&\& \ (C == 1)) \Rightarrow !(A == 1)$$

Alternative Formulierung (Interpretation):

$$((B == 0) \ || \ (C == 0)) \Rightarrow (A == 0)$$

b) Mögliche Formulierung:
$$((A == 1) \ \&\& \ (B == 0) \ \&\& \ (C == 0)) \ || \ ((A == 0) \ \&\& \ (B == 1) \ \&\& \ (C == 1))$$

Alternative Formulierung:

$$((A == 1) \ \&\& \ (B == 0) \ \&\& \ (C == 0)) \ != \ ((A == 0) \ \&\& \ (B == 1) \ \&\& \ (C == 1))$$

Alternative Formulierung:

$$(A != B) \ \&\& \ (A != C) \text{ bzw. } ((A == 1) != (B == 1)) \ \&\& \ ((A == 1) != (C == 1))$$

c) Mögliche Formulierung:
$$!((A == 1) \ \&\& \ (B == 1) \ \&\& \ (C == 1))$$

Alternative Formulierung:

$$(A == 0) \ || \ (B == 0) \ || \ (C == 0)$$

Alternative Formulierung:

$$(A != B) \ || \ (A != C) \ || \ (B != C) \text{ bzw. } ((A == 1) != (B == 1)) \ || \ ((A == 1) != (C == 1)) \ || \ ((B == 1) != (C == 1))$$
3. Ist der Operator „==“ als relationaler Vergleichsoperator eingesetzt, vergleicht dieser Zahlenwerte (keine Aussagen) auf beiden Seiten des Operators. Diese Zahlenwerte müssen gleich sein. Der Einsatz von „==“ als logischer Operator fordert auf beiden Seiten des Operators denselben Wahrheitswert (jeweils wahr bzw. jeweils falsch). Nachdem wahre Aussagen mit dem Wert 1 und falsche Aussagen mit dem Wert 0 ausgewertet werden, lassen sich logische Aussagen programmintern auf relationale Vergleiche zurückführen.
4. Für ein effizienteres Lösungsverfahren des Solvers werden in einem OPL-Modell logische Ausdrücke intern durch CPLEX in lineare Ausdrücke umgewandelt. Dabei entstehen programmintern zusätzliche Variablen und Nebenbedingungen. Wird das Problem also bereits in linearisierter Form implementiert kann sich dies aufgrund der geringeren Anzahl an Nebenbedingungen und Entscheidungsvariablen positiv auf die Laufzeit des Modells auswirken. Ein weiterer Nachteil von Nebenbedingungen mit logischen Ausdrücken ist, dass Fehler bei diesen Nebenbedingungen in

der Unlösbarkeits-Analyse von CPLEX im Konflikte-Fenster gar nicht angezeigt werden. Im Relaxationen-Fenster können diese Nebenbedingungen zwar angezeigt werden, jedoch keine Relaxations-Vorschläge.

5.

```
2*produktion <= kunde[1] + kunde[2];
produktion >= kunde[1] + kunde[2]-1;
produktionsmenge >= 100*produktion;
produktionsmenge <= M*produktion;
```

6. Die Fehler sind rot markiert:

```
//1. Deklaration und Initialisierung der Modellparameter
int nProdukte = 4;
range produkte = 1..nProdukte;
float kosten[produkte] = [1, 1.5, 2, 2.55];

//2. Deklaration der Gewinnfunktion
int n = 5;
int intervallobergrenze [1..n] = [10, 15, 22, 26, 30];
float steigungSprung [1..2][1..n] = [ [0, 1.5, 0, -2, 0],
[3, 0, 0, 0, -4]]; //falsche Intervalle
float letzteSteigung = -1;

pwlFunction gewinn = piecewise(i in 1..n, k in 1..2) {/* falscher Befehl für
die Initialisierung der stückweise linearen Funktion */

steigungSprung[k][i] -> intervallobergrenze[i]; letzteSteigung
}; //Semikolon an falscher Stelle

//3. Deklaration der Entscheidungsvariablen
dvar int+ produktionsmenge[produkte];
dvar boolean lieferung[produkte]; //1, falls Produkte ausgeliefert
// werden sollen, 0 sonst

//4. Zielfunktion
maximize sum (p in produkte) gewinn(produktionsmenge[p]);
//Funktionswert über runde Klammer abrufen

//5. Nebenbedingungen
subject to {
forall (p in produkte){
(lieferung[p] == 0) => (produktionsmenge[p] == 0); /*inhaltlicher Fehler: alle
Produkte würden nicht geliefert werden*/
}
sum (p in produkte) kosten[p]*lieferung[p] <= 6;
(lieferung[1] == 1) => (lieferung[3] == 1) => (lieferung[4] == 0); /*Äquivalenz
und keine Zuweisung eines Wertes und Klammern vergessen*/

(lieferung[4] != 1) => ((produktionsmenge[2] >= 20 ) && (20 >=
produktionsmenge[1])); //nicht erlaubt -> Einführen einer Konjunktion

sum(p in produkte) (lieferung[p] >= 1) >= 3; //,,>“-Bedingung nicht zulässig
}
```

Optimale Lösung:

```
lieferung[produkte] = [1, 1, 0, 1];
produktionsmenge[produkte] = [15, 20, 0 15];
```

Optimaler Zielfunktionswert: 31,5

7. Mit Hilfe der Ausdrücke `pwlFunction` und `stepFunction` werden in CPLEX Abschnittsweise lineare Funktionen deklariert. Während `pwlFunction` zur Deklaration von allgemeinen stückweisen linearen Funktionen verwendet wird, wird zur Deklaration des Spezialfalls einer Treppenfunktion der Ausdruck `stepFunction` verwendet. Die Initialisierung der Funktionen des Typs `pwlFunction` erfolgt über den Befehl `piecewise{}`. Funktionen des Typs `stepFunction` benötigen den Befehl `stepwise{}`.
8. Programmcode Grenzsteuersatz:

```
pwlFunction grenzsteuersatz = piecewise {
    0 -> 9169; 14 -> 9169;
    0.0019661817 -> 14255;
    0.0004315925 -> 55961;
    0 -> 265327; 3 -> 265327; 0
};

{float} einkommen = {0, 5000, 9408, 9169, 10000, 14250, 14255, 14260, 55900,
    55961, 265326.999, 265327, 1000000};
float steuersatz [einkommen];

execute{
    writeln ("Steuersätze (in Prozent): ")
    for (var e in einkommen){
        steuersatz[e] = grenzsteuersatz(e);
        writeln (steuersatz [e]);
    }
}
```

Alternativ:

```
int n = 4;
int intervallobergrenze [1..n] = [9169, 14255, 55961, 265327];
float steigung_sprung [1..2][1..n] = [[0, 0.0019661817, 0.0004315925, 0],
    [14, 0, 0, 3]];
float letzteSteigung = 0;

pwlFunction grenzsteuersatz = piecewise(i in 1..n, k in 1..2) {
    steigung_sprung[k][i] -> intervallobergrenze[i]; letzteSteigung
};

{float} einkommen = {0, 5000, 9168, 9169, 10000, 14250, 14255, 14260, 55900,
    55961, 265326.999, 265327, 1000000};
float steuersatz [einkommen];

execute{
    writeln ("Steuersätze (in Prozent): ")
    for (var e in einkommen){
        steuersatz[e] = grenzsteuersatz(e);
        writeln (steuersatz [e]);
    }
}
```

Ausgabe der Steuersätze (für beide Varianten):

Steuersätze (in Prozent):

0
0
0
14
15.633896993
23.990169218
24.000000126
24.002158089
41.973669789
41.999996931
41.999996931
44.999996931
44.999996931

9. a) Die Treppenfunktion hat im Intervall von $(-\infty; 12)$ einen Funktionswert von -2. An der Stelle mit Abszissen-Wert 12 erfolgt ein Sprung zu einem Funktionswert von 4. Dieser Funktionswert ist im Intervall $[12; 20)$ konstant. Nach dem nächsten Sprung nimmt die Funktion im Intervall $[20; 28)$ den Wert -8 an. Der vorletzte Sprung erfolgt bei dem Abszissen-Wert 28. Anschließend besitzt das Intervall $[28; 36)$ einen zugehörigen Funktionswert von 16. Im fünften und letzten Funktionsabschnitt $[36; \infty)$ liegt der Funktionswert konstant bei 5.
- b) Im Intervall von $(-\infty; -2)$ hat die Funktion eine Steigung von 2. Ferner ist bekannt, dass die Funktion im ersten Intervall durch den Punkt $(-3; -2)$ verläuft. Bei dem Abszissen-Wert -2 findet ein Sprung um 3 LE statt. Dieser Sprung erfolgt von dem Funktionswert 0 auf den Wert 3. Der zweite Funktionsabschnitt in $[-2; 6)$ verläuft mit einer Steigung von 1 weniger steil als das erste Intervall. Am Ende dieses Intervalls im Punkt $(6, 11)$ liegt eine Knickstelle ohne Sprung vor. Ab dem Abszissen-Wert von 6 bleibt der Funktionswert im Funktionsabschnitt in $[6; 14)$ konstant bei 11. Ein erneuter Sprung um 5 Einheiten erfolgt am Ende des Abschnitts bei dem Abszissen-Wert 14 und ein Funktionswert von 16 stellt sich ein. Im vorletzten Intervall $[14; 22)$ sinkt der Funktionswert stetig mit einer Steigung von -1 bis am Ende des Intervalls der Funktionswert 8 angestrebt wird. Am Abszissen-Wert 22 erfolgt der letzte Sprung und zwar reduziert sich der Funktionswert um 2 Einheiten auf den Wert 6. Ab diesem Punkt sinkt die Funktion in ihrem letzten Abschnitt $[22; \infty)$ stetig mit einer negativen Steigung von 2.
10. a) Falls ein Modell nicht lösbar ist, so unterstützt CPLEX den Nutzer mit einer Fehleranalyse. Dabei werden Nebenbedingungen ermittelt, die im Konflikt miteinander stehen. Das Konflikte-Fenster zeigt eine minimale Teilmenge an sich widersprechenden Nebenbedingungen an. Konflikte mit dem Definitionsbereich von Entscheidungsvariablen werden nicht angezeigt.
- b) Damit im Konflikte-Fenster konfligierenden Nebenbedingungen angezeigt werden, müssen die Nebenbedingungen benannt werden. Sind die Nebenbedingungen nicht benannt, so bleibt das Konflikte-Fenster leer. Außerdem werden logische Nebenbedingungen im Konflikte-Fenster ebenfalls nicht berücksichtigt.
- c) Es werden nacheinander Nebenbedingungen eliminiert und überprüft, ob das Modell weiterhin unlösbar bleibt. Dieses Vorgehen wird so lange wiederholt bis das Streichen einer weiteren Nebenbedingung zur Lösbarkeit des Modells führen würde.
11. a) Alle drei Nebenbedingungen gehören zur minimalen Teilmenge an konfligierenden Nebenbedingungen. Die NB1 beschränkt die Entscheidungsvariable $x[1]$ von unten auf den Wert 3 und

NB2 stellt die untere Schranke von $x[2]$ auf den Wert 4. Setzt man die Werte der unteren Schranken der beiden Entscheidungsvariablen in die NB3 ein erhält man folgenden Widerspruch: $2*3 + 3*4 = 18 \geq 15$. Wird eine der Nebenbedingungen gestrichen, so ist das Modell lösbar.

- b) CPLEX möchte in NB2 eine Relaxation vornehmen. Der Wert der rechten Seite soll von 4 auf 3 gesenkt werden. Wird die untere Schranke von $x[2]$ auf 3 gesenkt, so ist durch genaues Hinsehen zu erkennen, dass das Modell für $(x[1] = x[2] = 3)$ lösbar ist und die Nebenbedingungen nicht mehr miteinander im Konflikt stehen. Der Hintergrund dieser Anpassung ist, dass CPLEX bei der Relaxation versucht die Veränderung der Parameterwerte zu minimieren. In diesem Fall wurde der Parameter auf der rechten Seite nur um den Wert 1 verändert. Dieser Wert ist auch im Lösungen-Fenster „solution (feasible relaxed sum of infeasibilities) with objective 1“ abzulesen. Hätte CPLEX mit den ursprünglichen unteren Schranken von 3 für $x[1]$ und 4 für $x[2]$ nur die NB3 relaxieren wollen, müsste der Wert der rechten Seite von 15 auf 18 erhöht werden, damit die Nebenbedingung erfüllt werden kann. Dies entspricht einer Veränderung um den Wert 3, wodurch diese Relaxation weniger erstrebenswert für CPLEX ist. Ferner wäre es CPLEX noch möglich gewesen die untere Schranke von $x[1]$ in NB1 anzupassen. Um das Modell lösbar zu machen hätte der Wert der rechten Seite von 3 auf 1 gesenkt werden müssen, wodurch eine Änderung um den Wert 2 resultiert. Da CPLEX die Veränderung der Parameterwerte in der Relaxation minimieren möchte, wird im Relaxationen-Fenster eine Relaxation in NB2 angezeigt.
- c) Die Anzeigen im Konflikte- und Relaxationen-Fenster ändern sich nicht. Lediglich die Anzeige im Lösungen-Fenster ändert sich: „solution (optimal relaxed sum of infeasibilities) with objective 21“. Während bei dem voreingestellten Befehl „cplex.feasoptmode = 0“ die Veränderung der Parameterwerte als Lösungswert ausgegeben wird, wird durch den Befehl „cplex.feasoptmode = 1“ hingegen der Zielfunktionswert der relaxierten Lösung ausgegeben. Durch die Relaxation der rechten Seite der NB2 auf den Wert 3 nehmen beide Entscheidungsvariablen den Wert 3 an ($x[1] = x[2] = 3$). Es folgt der relaxierte Zielfunktionswert: $2*3 + 5*3 = 21$.

Aufgaben Kapitel 9 – Lösung

1. Ein- und Ausgabe von Daten, Pre- und Post-Processing, Festlegung von CPLEX Einstellungen und Zeitmessung.
2. Die Fehler sind rot markiert:

```
execute{
    var objekt = new IloOplOutputFile ("ausgabe.txt", true);
                                     // keine Eingabe, sondern Ausgabe
                                     // falscher Dateiname

    objekt.writeln ("Ausgabe");
    objekt.close();                  // Ansprache der Datei vergessen
}
```

3. Neben Excel-Dateien können auch Text-Dateien und insbesondere CSV-Dateien als Datenquelle für OPL-Modelle verwendet werden, um aus diesen Inhalte einzulesen und in OPL Datenelemente zu schreiben. CSV-Dateien sind einfach strukturierte, universell und häufig eingesetzte Datendateien, die von allen gängigen Tabellenkalkulationsprogrammen verarbeitet werden können. CSV steht dabei für *Comma Separated Values* (kommaseparierte Werte), weil die einzelnen Werte innerhalb des textbasierten Dateiformats durch Kommata voneinander abgetrennt werden.
4. Zunächst muss ein Datenelement erzeugt werden, in dem der Name der zu lesenden CSV-Datei angegeben wird. Anschließend wird im Script Block zum Lesen der Datei ein Dateiojekt erzeugt. Dann kann geprüft werden, ob die Datei tatsächlich existiert. Sollte dies nicht der Fall sein, kann eine entsprechende Fehlermeldung ausgegeben werden. Existiert die Datei, ist es hilfreich – zur Kontrolle als Debugging-Ausgabe – zunächst den Namen der nun zu lesenden Datei im Scriptingprotokoll des Studios ausgeben zu lassen. Nun können die OPL Datenelemente beispielsweise durch Schleifen befüllt werden, bis das Ende der zu lesenden Datei erreicht ist. Nach dem Durchlaufen der Schleife wird abschließend die Datei geschlossen.
5. Die Fehler sind rot markiert:

```
execute funktion {
    function Fehlersuche(dateiname,variable) {
        var datei = new IloOplInputFile(dateiname);
        if (datei.exists) {
            /*Dateiobjekt „datei“, nicht der Name der
            einzulesenden CSV-Datei*/
            while (!datei.eof) { /*Ohne Negation: datei.eof=false, da
            Datei nicht vollends ausgelesen ist
            -> while-Schleife wird nicht durchlaufen*/
                var zeile = datei.readline();
                if (zeile.length>0 && zeile.indexOf("/")!=0) {
                    var array = zeile.split(",");
                    // CSV-Datei wird an Kommata getrennt
                    variable(array);
                }
            }
            datei.close();
        } else {
            writeln("Fehler: Die Datei '"+ dateiname + "' existiert nicht!");
        }
    }
}
```

```

{string} produkte = {};

execute produktmenge {
    function produktinitialisierung(array) {    // Parameter fehlt
        produkte.add(array[0]);
    }
    Fehlersuche("Fehlersuche.csv", produktinitialisierung);
                                                // Falscher Aufruf der Funktion
}

float preis[produkte];                        //Menge der Produkte vor dem Durchlaufen
float kosten[produkte];                      //des execute-Blocks "produktmenge" noch
float wichtigkeit[produkte];                 //nicht initialisiert und deshalb sind die
string standort[produkte];                   //OPL-Felder noch nicht deklarierbar

execute daten {
    function dateninitialisierung(array) {
        preis[array[0]] = (array[1]);
        kosten[array[0]] = (array[2]);
        wichtigkeit[array[0]] = (array[3]);
        standort[array[0]] = (array[4]);    // Initialisierung von standort fehlt
    }

    Fehlersuche("Fehlersuche.csv", dateninitialisierung);
    /*Parameterübergabe in die Funktion Fehlersuche falsch. In Zeile 9
    würde sonst stehen: dateninitialisierung(array)(array) und die
    Funktion kann nicht richtig aufgerufen werden*/
}

```

6. Pre-Processing: Unterstützung der Vorverarbeitung von Daten und Initialisierung von Datenstrukturen. Entsprechende ILOG Script Blöcke stehen in der Modelldatei vor dem eigentlichen Modell.

Post-Processing: Weiterverarbeitung von Ergebnisdaten aus der Optimierung, Aufbereitung von Ergebnisdaten für die Ausgabe. Entsprechende ILOG Script Blöcke stehen dann in der Modelldatei nach dem eigentlichen Modell. Im Post-Processing steht eine Reihe von Status-Funktionen bzw. Eigenschaften des cplex-Objekts von Entscheidungsvariablen und Nebenbedingungen zur Nutzung in ILOG Script zur Verfügung.

7. Mögliche Lösung:

```
//Deklarieren von zweidimensionalen Feldern
```

```
int produkt1 [preis][tag];
int produkt2 [preis][tag];
```

execute-Block:

```

execute{
    /*Umwandeln der Entscheidungsvariablen x [produkt][preis][tag] in ein zweidimensionales Feld je Produkt, sodass dieses in Excel ausgegeben werden kann*/

    //Produkt 1
    for (var p in preis){
        for (var t in tag){
            produkt1 [p][t]=x["Produkt1"][p][t];
        }
    }
}

```

```

//Produkt 2
for (var p in preis){
    for (var t in tag){
        produkt2 [p][t]=x["Produkt2"][p][t];
    }
}
}

```

8. Möglicher execute-Block:

```

execute{
    writeln("Die Anzahl zu schaltender Werbung auf Strassenbahnen");
    writeln("\t liegt im Intervall zwischen 0 und "
        + kapazitaetMedium[1] + ".");
    writeln("Die optimale Anzahl für dieses Medium beträgt " + nSchaltung[1]);
    writeln();
    writeln("Die Anzahl zu schaltender Online-Werbung");
    writeln("\t liegt im Intervall zwischen 0 und "
        + kapazitaetMedium[2] + ".");
    writeln("Die optimale Anzahl für dieses Medium beträgt " + nSchaltung[2]);
    writeln();
    writeln("Die Anzahl zu schaltender Radiospots");
    writeln("\t liegt im Intervall zwischen 0 und "
        + kapazitaetMedium[3] + ".");
    writeln("Die optimale Anzahl für dieses Medium beträgt " + nSchaltung[3]);
    writeln();
    writeln("Damit ergibt sich eine Gesamtreichweite von " +
        cplex.getObjValue() + " Personen.");
}

```

9. a) Modell-Datei:

```

//Inputparameter
int nMedien;
range medien = 1..nMedien;
int reichweite[medien];
int kosten[medien];
int kapazitaetMedium[medien];
int kapazitaetAgentur;
int budget;

//Name der Textdatei
string txtdatei="RideEasy.txt";

//Name der CSV-Datei
string csvdatei = "RideEasy.csv";

execute {

    //Initialisierung durch Textdatei
    var dateiobjekttxt = new IloOplInputFile(txtdatei);
    nMedien= parseFloat(dateiobjekttxt.readline());
    kapazitaetAgentur = parseFloat(dateiobjekttxt.readline());
    budget = parseFloat(dateiobjekttxt.readline());

    //Initialisierung der Mengen durch CSV-Datei
    var dateiobjektcsv = new IloOplInputFile(csvdatei);
    if (dateiobjektcsv.exists) {
        writeln("Lese " + csvdatei);
        while (!dateiobjektcsv.eof) {

```

```

        var zeile = dateiobjektcsv.readline();
        writeln(zeile);
        if (zeile.length>0 && zeile.indexOf("//")!=0) {
            var feld = zeile.split(",");
            var medien = feld[0];
            reichweite[medien] = feld[1];
            kosten[medien] = feld[2];
            kapazitaetMedium[medien] = feld[3];
        }
    }
    dateiobjektcsv.close();
} else {
    writeln("Fehler: Die Datei '"+ csvdatei +"' existiert nicht!");
}
}

dvar int+ nSchaltung[medien];

maximize sum(i in medien) reichweite[i] * nSchaltung[i];

subject to {
    sum(i in medien) kosten[i] * nSchaltung[i] <= budget;
    forall(i in medien)
        nSchaltung[i] <= kapazitaetMedium[i];
    sum(i in medien) nSchaltung[i] <= kapazitaetAgentur;
}

```

b) Modell-Datei:

```

//Erstellen von Hilfsvariable im eigentlichen Modell
float dateninitialisierung;
float gesamtlaufzeit;

execute Laufzeit_Dateninitialisierung{
    dateninitialisierung = cplex.getCplexTime();
    writeln("Die Laufzeit der Dateninitialisierung beträgt " +
        dateninitialisierung + " Sekunden.");
}

```

Dieser ILOG Script Block muss direkt vor dem Einführen der Entscheidungsvariablen stehen.

```

execute Laufzeit_Solver{
    gesamtlaufzeit = cplex.getCplexTime();
    writeln("Die Laufzeit des Gesamtmodells beträgt "+ gesamtlaufzeit +
        " Sekunden.");
    writeln("Damit beträgt die reine Solverlaufzeit "+
        (gesamtlaufzeit-dateninitialisierung) + " Sekunden.");
}

```

Dieser ILOG Script Block steht am Ende des Modells nach dem Nebenbedingungs-Block.

c) Modell-Datei:

```

execute Timelimit_Solver{
    cplex.tilim = 0.5;
}

```

Dieser ILOG Script Block muss vor dem Nebenbedingungs-Block stehen (allerdings nicht direkt zwischen Zielfunktion und Nebenbedingungs-Block), damit die Solverlaufzeit beschränkt werden kann.

10. Die Funktion `cplex.getDetTime()` liefert eine sogenannte deterministische Zeit. Dies ist ein Versuch, von der aktuellen und zufälligen Belastung des ausführenden Computers zu abstrahieren und auf demselben Computer oder ähnlichen Computern für dasselbe Modell und denselben Datensatz (bei gleicher Lösung) immer dieselbe „deterministische Zeit“ zu erhalten und damit auch reproduzieren zu können. Eine Einschränkung besteht allerdings darin, dass damit nur die Zeit gemessen werden kann, die der CPLEX Solver selbst benötigt, nicht aber die Zeit für Vor- bzw. Nachverarbeitungen in OPL. Für viele Zwecke ist diese Funktion somit nur bedingt geeignet, da damit nicht der Laufzeitbedarf der eigentlichen Modellbildung gemessen werden kann.

Aufgaben Kapitel 10 – Lösung

1. Mögliche Einsatzfelder:
 - Ein Modell mehrmals mit variierenden Input-Daten lösen.
 - Sensitivitätsanalysen durch Veränderung eines Input-Parameters unter sonst gleichen Bedingungen.
 - Integration von Optimierungsmodellen als Teilprobleme eines Lösungsprozesses.
 - Implementierung von Heuristiken/Matheuristics durch Verwendung von Output-Daten eines Optimierungsproblems als Inputdaten für die nächste Verfahrensiteration.
 - Kombination mehrerer Optimierungsprobleme.
2. Bei der programminternen Verarbeitung von OPL-Optimierungsmodellen wird der Programmcode einer Modellinstanz einmalig und Schritt für Schritt abgearbeitet. Im Gegensatz dazu ist es im Rahmen eines main-Blocks möglich, auf ein oder mehrere Optimierungsmodelle zuzugreifen, d. h., verschiedene Modelle bzw. Modellinstanzen in einer festgelegten Reihenfolge nacheinander zu durchlaufen. Durch die sequentielle Abarbeitung des Programmcodes innerhalb eines main-Blocks ist dort die Umsetzung rein sequentiell abzuarbeitender Lösungsverfahren möglich (z. B. von Heuristiken).
3. Wenn sich das OPL-Optimierungsmodell und der main-Block in derselben Modell-Datei befinden, existiert per Default ein Objekt `thisOplModel` (vom Typ `IloOplModel`), das eine Instanz des Optimierungsmodells repräsentiert. Zudem existiert per Default das Objekt `cplex` (vom Typ `IloCplex`) als Instanz des Solvers. Das OPL-Optimierungsmodell kann auch durch einen `include`-Befehl in die Modell-Datei eingebunden werden, das komplett ausgeführte Modell kann also auch in einer anderen Modell-Datei stehen. Auf dieses wird durch den `include`-Befehl vor dem main-Block verwiesen.
Existiert vor einem main-Block kein OPL-Optimierungsmodell, so funktioniert der Zugriff nicht mehr über die Default-Modellinstanz. In diesem Fall muss eine individuelle Modellinstanz verwendet werden, deren Bestandteile zuvor deklariert, initialisiert und zusammengefügt werden müssen.
4. Schritt 1: Erzeugen eines Objekts für die Modell-Quelle (`IloOplModelSource("...")`).
Schritt 2: Erzeugen eines Objektes für eine neue CPLEX-Instanz (`IloCplex()`).
Schritt 3: Erzeugen eines Objektes für eine Instanz des Optimierungsmodells (`IloOplModelDefinition(...)`).
Schritt 4: Erzeugen eines Objektes, bei dem die Instanz des Optimierungsmodells mit der CPLEX-Instanz verknüpft wird (`IloOplModel(..., ...)`).

5. a) main-Block:

```
include "beispiel_3.16.mod";

main{
    thisOplModel.generate();
    cplex.solve();
    writeln("Optimaler Zielfunktionswert: " + cplex.getObjValue());
    writeln("Benötigte Zeit: " + cplex.getCplexTime());
}
```

```
}
```

b) main-Block:

```
main{
    var source = new IloOplModelSource("beispiel_3.16.mod");
    var cplex = new IloCplex();
    var def = new IloOplModelDefinition(source);
    var exerciseInstance = new IloOplModel(def,cplex);
    exerciseInstance.generate();
    cplex.solve();
    writeln("Optimaler Zielfunktionswert: " + cplex.getObjValue());
    writeln("Benötigte Zeit: " + cplex.getCplexTime());
}
```

6. a) neue Modell-Datei:

```
//1. Deklaration der Modellparameter
int nf1 = ...; //Nachfrage nach Fahrradtyp 1
int nf2 = ...; //Nachfrage nach Fahrradtyp 2
int kap1 = ...; //Gesamtmaschinenzeit von M1 im Planungszeitraum
int kap2 = ...; //Gesamtmaschinenzeit von M2 im Planungszeitraum
int kap3 = ...; //Gesamtmaschinenzeit von M3 im Planungszeitraum

float bedarf11 = ...; //Bedarf von Fahrradtyp 1 an Maschinenzeit auf M1
float bedarf12 = ...; //Bedarf von Fahrradtyp 1 an Maschinenzeit auf M2
float bedarf13 = ...; //Bedarf von Fahrradtyp 1 an Maschinenzeit auf M3
float bedarf21 = ...; //Bedarf von Fahrradtyp 2 an Maschinenzeit auf M1
float bedarf22 = ...; //Bedarf von Fahrradtyp 2 an Maschinenzeit auf M2
float bedarf23 = ...; //Bedarf von Fahrradtyp 2 an Maschinenzeit auf M3
float db1 = ...; //Deckungsbeitrag je Einheit von Fahrradtyp 1
float db2 = ...; //Deckungsbeitrag je Einheit von Fahrradtyp 2

//2. Deklaration der Entscheidungsvariablen
dvar int+ menge1; //Geplante produzierte Mengeneinheiten von Fahrradtyp1
dvar int+ menge2; //Geplante produzierte Mengeneinheiten von Fahrradtyp2

//3. Zielfunktion
maximize db1*menge1 + db2*menge2; //DB-Maximierung

//4. Nebenbedingungen
subject to {
    bedarf11*menge1 + bedarf21*menge2 <= kap1; //Zeitrestriktion M1
    bedarf12*menge1 + bedarf22*menge2 <= kap2; //Zeitrestriktion M2
    bedarf13*menge1 + bedarf23*menge2 <= kap3; //Zeitrestriktion M3
    menge1 <= nf1; //Nachfragerestriktion Fahrradtyp1
    menge2 <= nf2; //Nachfragerestriktion Fahrradtyp2
}
```

main-Block (in separater Modell-Datei):

```
include "beispiel_3.16_ohneDaten.mod";

main{
    thisOplModel.generate();
    cplex.solve();
    writeln("Optimaler Zielfunktionswert: " + cplex.getObjValue());
    writeln("Benötigte Zeit: " + cplex.getCplexTime());
}
```

Daten-Datei (Wird der Ausführungskonfiguration hinzugefügt):

```
nf1 = 300;
nf2 = 200;
kap1 = 120;
kap2 = 140;
kap3 = 110;

bedarf11 = 0.2;
bedarf12 = 0.3;
bedarf13 = 0.5;
bedarf21 = 0.4;
bedarf22 = 0.2;
bedarf23 = 0.5;
db1 = 110;
db2 = 140;
```

b) Modell- und Daten-Datei wie in Aufgabenteil a), nur Modell-Datei in der der main-Block steht wird der Ausführungskonfiguration hinzugefügt.

main-Block:

```
main {
    var source = new IloOplModelSource("beispiel_3.16_ohneDaten.mod");
    var cplex = new IloCplex();
    var def = new IloOplModelDefinition(source);
    var exerciseInstance = new IloOplModel(def, cplex);
    var data = new IloOplDataSource("aufgabe_10.6.dat");
    exerciseInstance.addDataSource(data);
    exerciseInstance.generate();
    cplex.solve();
    writeln("Optimaler Zielfunktionswert: " + cplex.getObjValue());
    writeln("Benötigte Zeit: " + cplex.getCplexTime());
}
```

7. (1) ein, (2) Programmablauf, (3) cplex.solve(), (4) getObjValue(), (5) derselben, (6) Dateipfad, (7) absolut, (8) relativ, (9) IloOplDataSource("...").
9. a) Das Programm würde endlos laufen (es würde stets dieselbe Ursprungs-Instanz mit prognose = 0.8 gelöst werden), da das Abbruchkriterium nie erreicht wird.
b) Es würde fünf Mal dieselbe Ursprungs-Instanz mit prognose = 0.8 gelöst werden (mit einem Zielfunktionswert von 349 800 [EUR].
c) Siehe a).

Angewandte Optimierung mit IBM ILOG CPLEX

Optimization Studio

Modellierung von Planungs- und

Entscheidungsproblemen des Operations Research mit
OPL

Nickel, S.; Steinhardt, C.; Schlenker, H.; Burkart, W.;

Reuter-Oppermann, M.

2021, XIII, 293 S. 57 Abb., 32 Abb. in Farbe., Softcover

ISBN: 978-3-662-62184-4