

运筹学



主讲人：

电子邮件：

办公地点：

答疑时间：

夫运筹帷幄之中，
决胜千里之外。

ILOG OPL

线性规划



主要内容

- 线性规划数学模型
- ILOG OPL 语法
- 线性规划问题的求解



线性规划数学模型

$$\max z = c_1x_1 + c_2x_2 + \dots + c_nx_n$$

$$s.t. \begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq (= \geq) b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq (= \geq) b_2 \\ \dots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq (= \geq) b_m \\ x_j \geq 0 (j = 1, 2, \dots, n) \end{cases}$$

紧缩形式

$$\max z = \sum_{i=1}^n c_i x_i$$

$$s.t. \sum_{j=1}^n a_{ij} x_j \leq (= \geq) b_i (i = 1, 2, \dots, m)$$

$$x_j \geq 0 (j = 1, 2, \dots, n)$$

矩阵形式

$$\max(\text{或}\min)z = CX$$

$$s.t. \begin{cases} AX \leq (=, \geq) b \\ X \geq 0 \end{cases}$$

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

称为决策变量向量

$$C = (c_1, c_2, \dots, c_n) \quad \text{称为价值系数向量或目标函数系数向量}$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

称为资源常数向量或
约束右端常数向量

称为技术系数或约束系数矩阵

ILOG OPL 语法介绍

- 标识符
- 数据类型
- 决策变量
- 目标函数
- 约束条件



标识符的定义

- 标识符由字母和下划线开头，包含字母、数字和下划线组成。
- 标识符大小写敏感



基本数据类型

- IBM ILOG OPL 基本数据类型包括：
- 整型 int
- 浮点型 float
- 字符串型 string
- 布尔型 boolean



整型

- OPL 整数的概念跟其它编程和建模语言的概念是相同的。OPL 预定义了一个整型符号常量 `maxint` 代表 OPL 中能够存储的最大正整数。因此，OPL 整型数的范围为 `-maxint` 到 `maxint`。
- 例 1：定义整型数 `i`，并把 `i` 初始化为 25。
- `int i = 25;`
- 例 2：定义整型数 `n`，并把 `n` 初始化为 3。然后，定义整型数 `size`，并把 `size` 初始化为 `n` 的平方。
- `int n = 3;`
- `int size = n*n;`

浮点型

- OPL 提供了双精度的浮点数数据类型（符合 IEEE 754 标准）。OPL 预定义了一个浮点型常量 `infinity` 代表无穷大。
- `float f = 3.2;`



字符串型

- OPL 提供了字符串数据类型。字符串常量中支持转义字符。
- 例 3：定义一个字符串 s，并初始化为 “hello world!”。
- `string s="hello world!"`；
- 字符串的作用是作为数组的下标，如下例所示，定义了一个字符串集合 Tasks，并进行了初始化，然后可以定义一个数组，数组的下标可以使用该字符串集合里的元素。
- `{string} Tasks = {"masonry", "carpentry", "plumbing", "ceiling", "roofing", "painting", "windows", "facade", "garden", "moving"};`

布尔型

OPL 提供了布尔型数据类型。定义布尔型数据类型使用关键字 `boolean`。布尔数据类型限定数据只能取“`true`”或“`false`”两个值。布尔型更多是用来定义布尔型决策变量，或者在条件约束中用于决定哪些约束条件有效。



区间（range）

- OPL 提供了区间（range）这个复杂的数据结构，主要用来做数组的下标、迭代区间和定义形参的取值范围。
- 通过给定下限值和上限值的方式定义能够表达区间的数据。
- 例：定义一个区间变量 Rows，代表整数区间 [1， 10]。
- `range Rows = 1..10;`

区间 (range)

- 区间的下限值和上限值可以使用表达式。
- 例：
- $\text{int } n = 8;$
- $\text{range Rows} = n+1..2*n+1;$



用途 1：用作数组下标

- 例：定义具有 100 个元素的整型数组 A，数组下标从 1 到 100。
- range R = 1..100;
- int A[R];
- // 数组 A 是一个具有 100 个元素的整型数组。



用途 2：整型区间用作形参的迭代范围

- 例：建立一个循环，形参的迭代范围从 1 到 100
- range R = 1..100;
- forall(i in R) { // 循环体 ... }



用途 3：定义决策变量的取值范围

- 例：定义一个整型决策变量 i ，取值在 $[1, 100]$ 之间。
- `range R = 1..100;`
- `dvar int i in R;`



用途 4：定义决策变量的取值范围

- 也可以定义浮点型区间，用于浮点型变量的取值范围的定义。
- `range float X=1.0..100.0;`
- `dvar float x in X;`



集合

- OPL 提供了集合（set）这个复杂的数据结构，主要用来做数组的下标。集合是非索引无重复元素的集合。OPL 允许定义任何类型的集合。



1、集合的定义

- 假设 T 是个数据类型，可以使用 $\{T\}$ 或者 $\text{setof}(T)$ 来定义集合数据。
- 【例 3-13】定义整型集合 setInt
- $\{\text{int}\} \text{setInt} = \dots;$
- 【例 3-14】定义结构体类型为 Precedence 的集合
- $\text{setof}(\text{Precedence}) \text{precedences} = \dots;$
- 默认情况下，集合元素是按照创建的顺序排列的。

2 、集合的初始化

- OPL 集合初始化可以分为 2 种，一种是内部初始化，是在模型文件 (.mod) 完成；另一种是外部初始化，在数据文件 (.dat) 完成。



1) 集合的内部初始化

- 在模型文件中，定义集合的同时使用 {} 对集合进行赋值。这种初始化方式导致数据在脚本代码中不能使用。
- 【例 3-15】初始化集合 Week 为 "Mon", "Tu", "Wed", "Thu", "Fri", "Sat", "Sun" 。
- {string} Week = {"Mon", "Tu", "Wed", "Thu", "Fri", "Sat", "Sun"};
- 使用 asSet 内置函数和区间数据初始化集合

- 【例 3-16】使用区间 $[1..10]$ 初始化集合 s
- $\{\text{int}\} s = \text{asSet}(1..10);$
- s 被初始化为 $\{1, 2, \dots, 10\}$ 。 asSet 功能是将 range 数据转换为集合数据。
- 使用表达式初始化集合
- 【例 3-17】初始化集合 s 为 $\{1, 4, 7, 10\}$
- $\{\text{int}\} s = \{i \mid i \text{ in } 10:i \bmod 3 == 1\};$

2) 集合的外部初始化

- 在独立的数据文件中使用 {} 给出集合元素的值。
- 【例 3-18】集合的外部初始化
- 在模型文件中声明整型集合 a。
- {int} a = ...;
- 在数据文件中进行集合的初始化。
- $a = \{10, 20, 30, 40, 50, 60\};$

【例 3-19】 使用数据库表初始化集合

公交车的班次信息存储在 Access 数据库 bus.accdb 的“发车时刻表”中。在 IBM ILOG OPL 模型中定义公交车班次时刻表结构体 shift。定义 shift 类型的 Shifts 结构体集合，数据表中公交车的班次信息将存储于 Shifts 中。

- tuple shift{
- string ID;
- int hour;
- int minute;
- }
- {shift} Shifts=...;
- 在数据文件中添加如下语句。
- DBConnection db("access","bus.accdb");
- Shifts from DBRead(db,"select ID,hour,minute from 发车时刻表 ");

- 【例 3-20】使用工作表初始化集合
- 城区的交通路口信息存储于名为 "cumcm2011B.xls" 的 Excel 文件的“全市交通路口的路线”工作表中。定义结构体 adj 代表邻接表的结构，定义 adj 类型的 connectionstmp 结构体集合，工作表中数据将存储于 connectionstmp 集合中。

- tuple adj
- {
- int o;
- int d;
- float dist ;
- }
- {adj}connectiontmp=...;
- 在数据文件中添加如下语句。
- SheetConnection sheet("cumcm2011B.xls");
- connectionstmp from SheetRead(sheet," 全市交通路口的路线 '!A2:B929");

数组

- IBM ILOG OPL 的数组定义必须给出数组的下标范围。下标范围可以是区间，可以是集合，可以是整数集合或字符串集合，也可以是结构体集合。



一维数组

- 一维数组是最简单的数组，定义一维数据需要说明数组元素的类型和下标范围。
- 例：定义具有 4 个元素的整型数组 a ，数据元素分别为 10，20，30 和 40，4 个元素的索引值分别为 1，2，3 和 4。
- $\text{int } a[1..4] = [10, 20, 30, 40];$
- 其中，1..4 是下标值域；10, 20, 30, 40 非别是通过下标值引用的数组元素的值。

一维数组

- 例：定义 4 个元素的浮点型数组 f，4 个元素分别为 1.2，2.3，3.4 和 4.5。4 个元素的索引分别为 5，6，7 和 8。
- `float f[5..8] = [1.2, 2.3, 3.4, 4.5];`
- 其中，5..8 是下标值域；1.2, 2.3, 3.4, 4.5 分别是通过下标值引用的数组元素的值。

一维数组

- 例：定义一个字符串数组 d ，其数组元素分别为 `d[1]="Monday"`； `d[2]="Wednesday"`；
- `string d[1..2] = ["Monday", "Wednesday"]`;



一维数组

- 例：定义一个整型数组 a ，数组元素的值为 $a["Mon"]=10, \dots, a["Sun"]=70$ 。
- 首先定义一个字符串集合 $week$ 。
- $\{string\} Week = \{ "Mon", "Tu", "Wed", "Thu", "Fri", "Sat", "Sun" \};$
- 以字符串集合 $week$ 中的元素做索引。
- $int a[Week] = [10, 20, 30, 40, 50, 60, 70];$
- 即元素索引可以是字符串，如 $a["Mon"], \dots, a["Sun"]$ 。

一维数组

- 例：定义一个整型数组 a 表示一个带权图的边的权值。
- `tuple Edges { int orig; int dest; }`
- `{Edge} Edges = {<1,2>, <1,4>, <1,5>};`
- `int a[Edges] = [10,20,30];`
- 即下标也可以是结构体集合， $a[<1,2>]=10$ ， $a[<1,4>]=20$ ， $a[<1,5>]=30$ 。

多维数组

- 定义多维数组需要指明数组名和每一维的下标范围。
- 【例 3-26】 定义一个二维数组，数组一维和二维的下标分别为区间常量 [1..2] 和 [1..3] 。
- `int a[1..2][1..3] ;`
- 多维数组的每一维下标可以是不同类型的集合

- 【例 3-27】 定义一个二维数组，数组第一维下标区间是字符串集合 Week，第二维的下标区间是 [1..3]
- {string} Week = {"Mon", "Tu", "Wed", "Thu", "Fri", "Sat", "Sun"};
- int a[Week][1..3] = ...;

2、数组初始化

- OPL 数组初始化可以分为 2 种，一种是内部初始化，是在模型文件完成；另一种是外部初始化，在数据文件完成。



1) 数组内部初始化

- 在模型文件中，定义数组的同时直接给数组元素赋值。这种初始化方式导致数据在脚本代码中不能使用。（后期通过具体例子讲解）



2、数组外部初始化

- 在独立的数据文件中使用语句对数组进行初始化。语句可以给出数组元素的值，也可以从数据库中读取数据，也可以从数据表中读取数据。（后期通过具体例子讲解）



结构体

- 结构体类型可以构造复杂的数据结构。
- 1、结构体类型的定义
- 定义结构体使用关键字 `tuple` ，同时给出结构体的名称和数据成员的类型和名称。



- tuple 结构体名 {
- 类型 数据成员 1;
- 类型 数据成员 2;
- ...
- };



- 【例 3-36】 声明一个结构体类型 Point，该结构体类型具有两个整型的数据成员 x 和 y。
- tuple Point {
- int x;
- int y;
- };

- 结构体类型 T 定义完毕，就可以用这个结构体类型 T 定义类型为 T 的结构体数据、元素类型为类型 T 的结构体数组、元素类型为 T 的结构体集合、数据域类型为 T 的复杂结构体类型。



- 结构体数据的初始化
- 结构体数据可以使用内部初始化和外部初始化方式。（后期通过例子讲解）



数据一致性

- 对一个优化问题进行求解，模型和数据的正确性都很关键。模型给出错误解或者无解，可能是模型本身存在问题，也可能是数据本身存在问题。对于一个实际的应用，经过数据的定义、初始化和数据转换后，模型中的有些数据存在关联关系，因此通过验证模型输入数据的一致性可以避免因数据不一致导致的错误的结果和烦琐的调试过程。

- OPL 可以通过几种方式检查数据的一致性。
- 1、使用 with...in... 检查数据成员一致性
- with...in... 限定一个结构体集合的数据成员的值必须包含在另一个集合中。OPL 在模型运行后，初始化集合时对结构体集合中数据成员的值进行一致性检查。

- 【例 3-45】一个图 $G (v, e)$ 有若干个顶点和若干条边。顶点集合为整数集合 `vertex`，存储顶点的编号；`edge` 是结构体，数据成员为边的两个顶点的编号，`edges` 是 `edge` 类型的结构体集合。集合中每条边的两个顶点应该都包含在顶点集合 `vertex` 中。

- $\{\text{int}\} \text{ vertex} = \{1,2,3,4,5\};$
- tuple edge {
- int origin;
- int destination;
- }
- 语句 $\{\text{edge}\} \text{ edges with origin in vertex, destination in vertex} = \{<1,4>, <5,1>\};$ 不会产生任何问题，因为集合 edges 中的数据通过了一致性检查 with origin in vertex, destination in vertex。但是语句 $\{\text{edge}\} \text{ edges with origin in vertex, destination in vertex} = \{<1,4>, <5,6>\};$ 在程序运行时会出现错误，因为没有通过一致性检查。

- 2、通过断言表达式检查数据的一致性。
- 断言表达式也经常用来检查优化模型是否遵守了模型的有效性条件。断言表达式使用 `assert` 指令。断言是布尔表达式，结果必须为 `true`，否则将会引起一个运行时错误。

- 【例 3-46】在运输问题中，验证需求和供应数据是否具有关联性（总和相等）
- `int demand[Customers] = ...;`
- `int supply[Suppliers] = ...;`
- `assert sum(s in Suppliers) supply[s] == sum(c in Customers) demand[c];`
- 再如，如果是多产品的情况：
- `int demand[Customers] [Products] = ...;`
- `int supply[Suppliers] [Products] = ...;`
- `assert forall(p in Products)`
`sum(s in Suppliers) supply[s][p] == sum(c in Customers) demand[c]`
`[p];`

3.7 决策变量和决策表达式

- OPL 决策变量的声明使用关键字 `dvar` 。
- 声明决策变量时，声明决策变量的类型，也可以同时声明取值范围。
- 决策变量可以是整型 `int`、浮点型 `float` 和布尔型 `boolean` 简单类型变量，也可以是关于这些类型的一维数组或多维数组。

决策变量的定义

- 优化问题中待求解的未知量称为决策变量。OPL 决策变量的声明使用关键字 `dvar`。声明决策变量时，需要声明决策变量的类型，也可同时声明决策变量的取值范围。决策变量可以是整型 `int`、浮点型 `float` 和布尔型 `boolean` 的简单变量，也可以是关于这些类型的一维数组或多维数组。

决策变量的定义

- `dvar int x1;`
- `dvar int x2;`
- `dvar float x[Products];`



决策变量的定义

- 【例 3-47】 定义决策变量 transp 是一个二维的整型数组，这个数组的所有元素都是决策变量，每个决策变量的取值在 $[0, 100]$ 。
- {string} Orig=…;
- {string} Dest=…;
- dvar int transp[Orig][Dest] in 0..100;

- 【例 3-48】 定义决策变量 transp 是一个一维整型数组，这个数组以结构体集合 routes 中的成员为数组下标。这个数组的所有元素都是决策变量，每个决策变量的取值在 $[0, 100]$ 。
- tuple Route { string orig; string dest }
- {Route} routes = ...:
- dvar int transp[routes] in 0..100;

- 【例 3-49】 决策变量的取值范围可以是区间数据
- `range Capacity = 0..limitCapacity;`
- `dvar int transp[Orig][Dest] in Capacity;`
- 【例 3-50】 决策变量的取值范围下限和上限也可以通过其他数据的值来指定
- `dvar int averageDelay in 0..maxDelay;`
- 如果不同决策变量的范围不同，可以这样定义
- `int capacity[route] = ...;`
- `dvar int transp[r in routes] in 0..capacity[r];`
- 在决策变量类型名后使用 “+” 关键字来限制决策变量只能为非负值。

- 【例 3-51】 定义决策变量 x 的取值范围是非负的整数
- `dvar int+ x;`
- 等价于
- `dvar int x in 0..maxint;`
- 【例 3-52】 定义决策变量 y 的取值范围是非负数
- `dvar float+ y;`
- 等价于
- `dvar float y in 0..infinity;`
- 【例 3-53】 定义决策变量 z 是布尔型决策变量，取值范围是 0 或者 1
- `dvar boolean z;`
- 等价于
- `dvar int z in 0..1;`

决策表达式

- 表达式中包含决策变量，这样的表达式叫决策表达式。决策表达式可以通过引用表达式名称而在模型中重复引用。通常，如果一个表达式有特定的含义，把它定义成决策表达式将会使模型可读性更好。使用关键字 `dexpr` 定义决策表达式。
- `dexpr int TotalFixedCost = sum(w in Warehouses) Fixed[w] * Open[w];`
- `dexpr float TotalSupplyCost = sum(w in Warehouses, s in Stores) SupplyCost[s][w] * Supply[s][w];`
- 如果在模型中使用了决策表达式，决策表达式的值会在问题浏览器窗口显示，可以通过观察决策表达式的值来判断模型存在的问题。
- 同类型不同值的决策表达式可以定义成决策表达式数组。

- 【例 3-54】 定义决策表达式数组 slack
- $\text{dexpr int slack}[i \text{ in } r] = x[i] - y[i];$



运算符和表达式

- 在 OPL 模型中，可以使用各种运算符和操作数来构建各种表达式。表达式可以用来：
- 指定数据或集合中的数据的值；
- 在循环中做筛选条件；
- 表达约束条件。
- 在第 1 和第 2 种表达式中，不包含决策变量，第 3 种表达式包含决策变量。

- 1、整型运算符和整型表达式
- 整型表达式由整型符号常量（maxint）、整型数据、整型运算符和整型决策变量以及一些系统函数（例如，abs（））构成。整型运算符包括+，-，*，div（整除），^2(只能使用2次方)，mod(or%)等运算符。在整型表达式中可以使用系统函数abs()用于求表达式的绝对值。

- 2、浮点运算符和浮点表达式
- 浮点表达式由浮点型符号常量（infinity）、浮点型数据、浮点型运算符和浮点型决策变量以及一些浮点型系统函数（例如，abs（））构成。浮点运算符包括 +，-，/，*，^2（使用受限）等运算符。在浮点型表达式中可以使用大量的系统函数，比如：floor()，round() 等。

- 3、条件表达式
- $(condition)?thenExpr : elseExpr$
- OPL 语言中的条件表达式和 C 语言类似，condition 可以是布尔表达式，也可以是逻辑表达式。条件表达式首先计算 condition 的结果，如果为真，则 thenExpr 表达式的结果做整个表达式的结果；否则 elseExpr 表达式的结果做整个表达式的结果。

- 【例 3-55】定义条件表达式，signValue 的值由数据 value 的值决定：value 大于 0，signvalue 等于 1；value 等于 0，signvalue 等于 0；value 小于 0，signvalue 等于 -1
- `int value = ...;`
- `int signValue = (value>0) ? 1 : (value<0) ? -1 : 0;`
- 定义条件表达式，absValue 的值是数据 value 绝对值
- `int absValue = (value>=0) ? value : -value;`
- 在表达式中，条件表达式 condition 只能由已知数据组成，不能包含决策变量。

布尔表达式和逻辑运算符

- 布尔表达式可以通过各种方式来构建。布尔表达式可以作为独立的约束条件，也可以用于条件约束，或者过滤条件。
- 整型表达式和传统的关系运算符 `==`, `!=` (not equal), `>=`, and `<=`



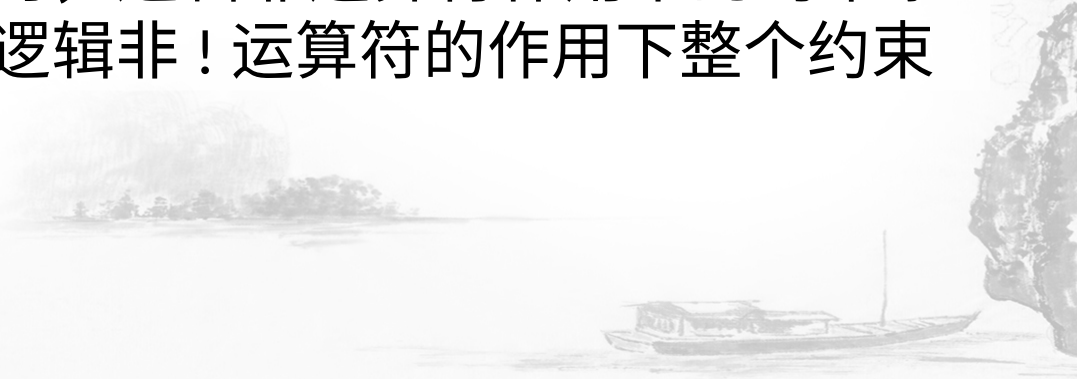
- 【例 3-56】 最短路径问题使用邻接矩阵存储数据结构时起点的约束条件中， $i \neq 1$ 是过滤条件， $adj[1][i] * x[1][i] == 1$ 是约束条件。
- $sum(i \text{ in } v: i \neq 1) adj[1][i] * x[1][i] == 1;$
- 浮点表达式和传统的关系运算符 $==$, $!=$ (not equal), $>=$, and $<=$
- 【例 3-57】 生产计划问题中的资源消耗总量小于库存量
- $forall(c \text{ in } Components)$
- $sum(p \text{ in } Products) Demand[p][c] * Production[p] <= Stock[c];$
- 字符串表达式和传统的关系运算符 $==$, $!=$ (not equal), $>=$, and $<=$
- 【例 3-58】 字符串表达式作为过滤条件
- $forall(i \text{ in } s : i \neq "a") x[i] >= 5;$

- 布尔表达式的结果是一个 boolean 值 true 或者 false。布尔表达式可以通过逻辑运算符组合成更复杂的表达式。逻辑运算符包括逻辑与 “&&”，逻辑或 “||”，逻辑非 “!”，逻辑不等于 “!=” 和逻辑等于 “==”，用于约束条件时还包括逻辑蕴含 “=>”。逻辑运算符的结果为 boolean 值 true 或 false。
- 逻辑运算符可以用于约束条件。逻辑与 && 用于约束条件时，所有的约束条件默认的逻辑关系是逻辑与关系，也就是所有约束表达式的结果必须是 true。

- 【例 3-59】 最短路径问题的起点和终点的约束条件分别为
- $\text{sum}(i \text{ in } v: i \neq 1) \text{adj}[1][i] * x[1][i] == 1;$
- $\text{sum}(i \text{ in } v: i \neq n) \text{adj}[n][i] * x[n][i] == 1;$
- 等价于
- $\text{sum}(i \text{ in } v: i \neq 1) \text{adj}[1][i] * x[1][i] == 1 \&\& \text{sum}(i \text{ in } v: i \neq n) \text{adj}[n][i] * x[n][i] == 1;$
- 逻辑或 || 用于约束条件时，逻辑或用来表达运算符两端的两个约束表达式至少有一个结果是 true 。从而保证整个约束条件结果为真。



- 【例 3-60】每种产品要么不生产，如果生产，则产量至少 30 个单位。
- forall(p in Products)
- $x[p] \geq 30 \mid \mid x[p] = 0;$
- 逻辑非！用于约束条件时，逻辑非运算符作用下的约束条件必须不成立，从而在逻辑非！运算符的作用下整个约束条件成立。



- 【例 3-61】 决策变量不能等于 0
- $\text{dvar int+ } a;$
- $\text{dvar int+ } b;$
- $\text{minimize } a;$
- $\text{subject to}\{$
- $!(a==0);$
- $\}$
- a 不能等于 0，才会使 $!(a==0)$ 结果为真。因此 a 最小值等于 1。
- 逻辑不等于！ = 用于约束条件时，指的是运算符两端的约束条件一个成立另一个不成立。

- 【例 3-62】 a 和 b 中有且只能有一个等于 0， a 取最小值
- $\text{dvar int+ } a;$
- $\text{dvar int+ } b;$
- $\text{minimize } a;$
- $\text{subject to}\{$
- $(a==0) \neq (b==0);$
- $\}$
- 当 $a=0$ ， $b=1$ 时，在逻辑不等于 \neq 的作用下，整个约束条件成立， a 取得最小值 0。

- 【例 3-63】 a 和 b 中有且只能有一个等于 0， b 取最小值
- $\text{dvar int+ } a;$
- $\text{dvar int+ } b;$
- $\text{minimize } b;$
- $\text{subject to}\{$
- $(a==0) \neq (b==0);$
- $\}$
- 当 $a=1$ ， $b=0$ 时，在逻辑不等于 \neq 的作用下，整个约束条件成立， b 取得最小值 0。
- 逻辑等于 $==$ 用于约束条件时，指的的运算符两端的约束条件必须同时成立或同时不成立。

- 逻辑蕴含 \Rightarrow : 逻辑蕴含表达约束条件或逻辑表达式的结果之间如果一个结果为真, 另一个必须为真; 而一个结果为假, 另一个结果可真可假的逻辑关系。
- 【例 3-66】 如果某个月的产品中混合有 v1 蔬菜油, 则必须混合 o3 蔬菜油。
- forall(m in Months) {(Use[m]["v1"] \geq 20) \Rightarrow Use[m]["o3"] \geq 20;}



5、聚合运算符和聚合表达式

- 可以利用聚合运算符计算一组相关表达式的累加 (sum)，连乘 (prod)，最小 (min)，最大 (max) 值等。聚合运算符包括 sum，prod，min 和 max。



- 【例 3-67】使用聚合运算符 min 来计算数组 capacity 中的最小值。
- `int capacity[Routes] = ...;`
- `int minCap = min(r in Routes) capacity[r];`



6、集合运算符和集合表达式：

- 集合的值通过集合表达式来赋值。可以利用 union , inter, , diff 和 symdiff 集合运算符来构建集合表达式
- `{int} s1 = {1,2,3};`
- `{int} s2 = {1,4,5};`
- `{int} i = s1 inter s2;`
- `{int} u = s1 union s2;`
- `{int} d = s1 diff s2;`
- `{int} sd = s1 symdiff s2;`
- 集合 i 被初始化为 {1} ; 集合 u 被初始化为 {1,2,3,4,5} ; 集合 d 被初始化为 {2,3} ; 集合 sd 被初始化为 {2,3,4,5} 。除此之外，集合表达式也可以通过区间来构建。

- 【例 3-68】把集合 s 初始化为有限集合 $\{1, 2, \dots, 10\}$ 。
- `{int} s = asSet(1..10);`
- 可以使用集合函数处理集合元素。
- 现有 S 是一个集合 $\{3, 6, 7, 9\}$ ， $item$ 是 S 的一个集合成员， n 是一个整型数，表 3-2 给出了集合函数的使用说明。

约束条件

- 关于决策变量的布尔表达式称为约束。为了让求解算法识别哪些布尔表达式是约束，必须使用优化指令 `subject to` 或者 `constraints`。约束必须放在目标函数之后。



- 1、条件约束
- 默认情况下，决策变量的取值必须使约束指令范围内的所有约束成立。可以通过使用 if...else... 来定义条件约束。通过此种方式定义的约束在程序执行过程中会根据设定的条件是否满足选择哪些约束必须满足，哪些条件不用考虑。

- 【例 3-69】 根据 d 的值决定两组约束条件中那个起作用
- `if ($d > 1$) {`
- `abs(freq[f] - freq[g]) \geq d;`
- `} else {`
- `freq[f] == freq[g];`
- `}`
- 如果 d 的值大 1，则决策变量的取值必须满足约束条件
- `abs(freq[f] - freq[g]) \geq d;`
- 否则，决策变量的取值必须满足约束条件
- `freq[f] == freq[g];`
- 使用条件约束时，`if...else...` 中的条件不能包含决策变量，也不能含有 `forall` 等关键字。当条件里必须包含决策变量，则可以使用条件约束 `=>`。
- 布尔表达式除了用于表达约束之外，还可以用来做形参的过滤条件。

- 2、浮点型约束
- 浮点型约束是约束里包含浮点型决策变量的约束表达式。浮点型约束不能使用关系运算符！ = ， > 和 < 。
- 3、离散型约束
- 离散型约束是由离散型数据、离散型决策变量、一些系统函数和算术运算符构成的布尔表达式。

4、字符串约束

- 字符串约束有字符串常量和关系运算符组成的约束。字符串约束不能用在决策变量上，因为决策变量没有字符串型。字符串约束可以作为形参的过滤条件。
- 【例 3-71】字符串常量和关系运算符组成的约束
- `{string} s = {"a", "b"};`
- `dvar int x[s] in 0..10;`
- `minimize sum(i in s) x[i];`
- `subject to {`
- `forall(i in s : i != "a")`
- `x[i] >= 5;`
- `}`

逻辑约束

- 用逻辑结构把线性约束组合在一起，或者使用 if...else... 把线性约束组合在一起来表达约束条件之间的复杂关系。OPL 包含的逻辑约束运算符如表 3-4 所示。



表 3-4 逻辑约束运算符

运算符	含义
&&	逻辑与
	逻辑或
!	逻辑非
=>	蕴含
!=	不同
==	等价

形参

- 形参在 IBM ILOG OPL 中起着非常重要的作用，在聚合运算符（sum、prod、min、max）、forall 语句、集合和数组中广泛使用。
- 形参有多种形式和用途：



- 1、 $p \in S$
- 其中， p 是形参， S 是 p 的迭代范围，可以是整数区间，也可以是整数集合、字符串集合或者结构体集合。 \in 是 OPL 关键字，用于表达成员关系。
- 【例 3-72】 求 $1^2+2^2+3^2+4^2+5^2+6^2$ ，形参从一个整数区间 $[1, 6]$ 中依次取值，这个区间实际上是一个整数集合
- `int n=6;`
- `int s == sum(i in 1..n) i*i;`



- 【例 3-73】 形参从一个字符串集合中取值
- `{string} Products = {"car", "truck"};`
- `float cost[Products] = [12000, 10000];`
- `float maxCost = max(p in Products) cost[p];`
- 【例 3-74】 形参在一个结构体集合中取值
- `{string} Cities = { "Paris", "London", "Berlin" };`
- `tuple Connection { string orig; string dest; }`
- `{Connection} connections = { <"Paris", "Berlin">, <"Paris", "London"> };`
- `float cost[connections] = [1000, 2000];`
- `float maxCost = max(r in connections) cost[r];`
- 不加过滤条件，意味着形参遍历集合，取得集合中所有集合元素的值。如果仅保留满足条件的形参的值，就必须使用过滤条件。

- 2、 $p \in S$: 过滤条件表达式
- 【例 3-75】 将过滤出满足 $1 \leq i < j \leq 8$ 这个条件的 i 和 j
- `int n=8;`
- `dvar int a[1..n][1..n];`
- `subject to {`
- `forall(i in 1..8)`
- `forall(j in 1..8: i < j)`
- `a[i][j] >= 0; }`
- 因此该约束条件只要求数组 a 的一维下标小于二维下标的数组元素的值大于等于 0。比如 $a[1][2]$ 必须大于等于 0；而 $a[2][1]$ 不受这个限制。

- 【例 3-76】把 i 小于 j 的 $i*j$ 项累加起来
- $\text{int } s = \text{sum}(i, j \text{ in } 1..n : i < j) i*j;$
- 形参 i 和 j 都从 1 变到 n ，那么有 $n*n$ 项，只把 i 小于 j 的 $i*j$ 项累加起来。下面的表达式和上面的表达式等价。
- $\text{int } s = \text{sum}(i \text{ in } 1..n) \text{sum}(j \text{ in } 1..n : i < j) i*j;$
- 在写形参表达式的时候，下列形式是等价的。
- $\text{forall}(i, j \text{ in } 1..n : i < j) a[i][j] \geq 0;$
- $\text{forall}(i \text{ in } 1..n, j \text{ in } 1..n : i < j) a[i][j] \geq 0;$
- $\text{forall}(\text{ordered } i, j \text{ in } 1..n) a[i][j] \geq 0;$
- 过滤条件中不支持聚合运算符。

案例 1

- 某公司生产氨气和氯化铵两种产品。公司的日处理能力为 50 单位的氮，180 单位的氢，40 单位的氯。
- 氨气的利润是 40 欧元每单位
- 氯化铵的利润是 50 欧元每单位
- 如何确定氨气和氯化铵的产量，使利润最大。

数据

	氮	氢	氯	利润 (欧元)
氨气	1	3	0	40
氯化铵	1	4	1	50
资源总量	50	180	40	

数学模型

$$\max z = 40x_1 + 50x_2$$

$$s.t. \begin{cases} x_1 + x_2 \leq 50 \\ 3x_1 + 4x_2 \leq 180 \\ x_2 \leq 40 \\ x_1, x_2 \geq 0 \end{cases}$$



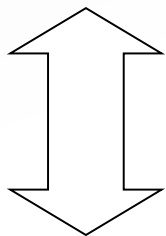
OPL 模型

- 定义决策变量
- `dvar float+ x1;`
- `dvar float+ x2;`



OPL 模型

数学模型中的目标函数： \max
 $z=40*x1+50*x2$



OPL 模型中的目标函数： maximize
 $40*x1+50*x2;$

OPL 模型

数学模型中的约束条件: $x_1 + x_2 \leq 50$

$$\begin{aligned} 3x_1 + 4x_2 &\leq 180 \\ x_2 &\leq 40 \end{aligned}$$

OPL 模型中的约束条件:

subject to {

$$x_1 + x_2 \leq 50;$$

$$3x_1 + 4x_2 \leq 180;$$

$$x_2 \leq 40;$$

}

ILOG OPL 模型

- dvar float+ x1;
- dvar float+ x2;
- maximize 40*x1+50*x2;
- subject to{
- x1+x2<=50;
- 3*x1+4*x2<=180;
- x2<=40;
- }
- 解： x1=20;x2=30;z=2300

ILOG OPL 模型

- dvar float+ gas;
- dvar float+ chloride;
- maximize 40*gas+50*chloride;
- subject to{
- gas+chloride<=50;
- 3*gas+4*chloride<=180;
- chloride<=40;
- }
- 解： gas=20;chloride=30;z=2300

模型说明

- **dvar** : (decision variable)
- **OPL 关键字**，表示此定义的变量是决策变量。
- **基本格式**：**dvar 数据类型 变量名**；例如：**dvar float gas**;
- **+**：放在“决策变量”中的“基本数据类型”之后，表示所定义的决策变量的非负约束。基本格式是：**数据类型 + 变量名**；例如：**dvar float+ gas**;
- **注**：“+”只能在决策变量中使用。

模型说明

- **maximize**（或 **minimize**）：
- **OPL 关键字**，放在表达式之前，表示求此表达式的最大值（或最小值）。
- **基本格式**：**maximize** 表达式；例如：**maximize 40 * gas + 50 * chloride;**



模型说明

- **subject to :**
- **OPL 的关键字，放在一组约束之前，是用于约束的另一种形式。**
- **基本格式： subject to { 一组约束 };**
- **例如： subject to {**
- **gas + chloride <= 50;**
- **3 * gas + 4 * chloride <= 180;**
- **chloride <= 40;}**

模型优化

使用数组使得模型可读性好，而且容易扩展。

```
{int} index = {1,2};
```

```
dvar float x[index];
```

```
maximize
```

```
    40 * x[1] + 50 * x[2];
```

```
subject to {
```

```
    x[1] + x[2] <= 50;
```

```
    3 * x[1] + 4 * x[2] <= 180;
```

```
    x[2] <= 40;
```

```
}
```

对比 C 的数组下标



模型优化

使用数组使得模型可读性好，而且容易扩展。

```
{string} Products = {"gas", "chloride"};
```

```
dvar float production[Products];
```

```
maximize
```

```
    40 * production["gas"] + 50 * production["chloride"];
```

```
subject to {
```

```
    production["gas"] + production["chloride"] <= 50;
```

```
    3 * production["gas"] + 4 * production["chloride"] <= 180;
```

```
    production["chloride"] <= 40;
```

```
}
```

对比 C 的数组下标

$$\max z = \sum_{i=1}^n c_i x_i$$

$$s.t. \sum_{j=1}^n a_{ij} x_j \leq (=, \geq) b_i (i = 1, 2, \dots, m)$$

$$x_j \geq 0 (j = 1, 2, \dots, n)$$



矩阵形式

$$\max(\text{或}\min)z = CX$$

$$s.t. \begin{cases} AX \leq (=, \geq) b \\ X \geq 0 \end{cases}$$

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

称为决策变量向量

$$C = (c_1, c_2, \dots, c_n) \quad \text{称为价值系数向量或目标函数系数向量}$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

称为资源常数向量或
约束右端常数向量

称为技术系数或约束系数矩阵

进一步优化模型

$$\sum_{i=1}^n c_i * x_i$$

sum (i in 1..n) c[i] * x[i]

$$\max z = \sum_{i=1}^n c_i x_i$$

maximize sum (i in 1..n) c[i] * x[i]



进一步优化约束条件

$$\sum_{i=1}^n a_{ij} * x_i \leq b_j (j = 1, \dots, m)$$

forall(j in 1..m)

sum(i in 1..n) a[i][j] * x[i] <= b[j];



ILOG OPL 模型

- dvar float+ x1;
- dvar float+ x2;
- maximize $40 * x1 + 50 * x2$;
- subject to{
 - $x1 + x2 \leq 50$;
 - $3 * x1 + 4 * x2 \leq 180$;
 - $x2 \leq 40$;
 - }



$$c = (40 \quad 50)$$

$$a = \begin{pmatrix} 1 & 1 \\ 3 & 4 \\ 0 & 1 \end{pmatrix}$$

$$b = (50 \quad 180 \quad 40)$$



优化模型

- `float a[1..3][1..2] = [[1,1]`
- `[3,4],`
- `[1,0]];`
- `float c[1..2] = [40,50];`
- `float b[1..3] = [50,180, 40];`
- `dvar float x[1..2];`
- `maximize sum(i in 1..2) c[i] * x[i];`
- `subject to {`
- `forall(i in 1..3)`
- `sum(j in 1..2)a[i][j] * x[j] <= b[i];`
- `}`



优化模型

- `{string} Products ={"gas","chloride" };`
- `{string} Resources ={"N","H","Cl" };`
- `float a[Resources][Products] =[[1, 1], [3, 4],[1,0]];`
- `float c[Products] = [40,50];`
- `float b[Resources] =[50, 180, 40];`
- `dvar float x[Products];`
- `maximize sum(p in Products) c[p] * x[p];`
- `subject to {`
- `forall(r in Resources)`
- `sum(p in Products)a[r][p] * x[p] <= b[r];`
- `}`



数据调整

- $c=(40,50);$
- $x=("gas", "chloride");$
- $a=[[1,3,0],$
- $[1,4,1]];$
- $b=(40$
- $50);$



模型调整

- {string} Products ={"gas","chloride" };
- {string} Resources ={"N","H","Cl" };
- float a[Products][Resources] =[[1, 3, 0], [1, 4, 1]];
- float c[Products] = [40,50];
- float b[Resources] =[50, 180, 40];
- dvar float x[Products];
- maximize sum(p in Products) c[p] * x[p];
- subject to {
- forall(r in Resources)
- sum(p in Products)a[p][r] * x[p] <= b[r];
- }
-

- {string} Products =...;
- {string} Resources =...;
- float a[Products][Resources] =...;
- float c[Products] = ...;
- float b[Resources] =...;
- dvar float x[Products];
- maximize sum(p in Products) c[p] * x[p];
- subject to {
- forall(r in Resources)
- sum(p in Products) a[p][r] * x[p] <= b[r];}

模型与数据分离

- $\text{Products} = \{ \text{"gas"}, \text{"chloride"} \};$
- $\text{Resources} = \{ \text{"N"}, \text{"H"}, \text{"Cl"} \};$
- $a = [[1, 3, 0], [1, 4, 1]];$
- $c = [40, 50];$
- $b = [50, 180, 40];$
- ** 不能出现下标

数据初始化方式

- `Products = {"gas","chloride" };`
- `Resources = {"N","H","Cl" };`
- `a=#["gas":#["N":1,"H":3,"Cl":0]#," chloride ":
#["N":1,"H":4,"Cl":1]#]#;`
- `c=#["gas":40," chloride ":50]#;`
- `b=#["N":50,"H":180,"Cl":40]#;`
- ** 注意 #[与]# 的匹配

显示结果

- 使用脚本
- `execute {`
- `writeln("p1=",x["gas"]);`
- `writeln("p2=",x["chloride"]);`
- `}`
- 则会在脚本日志窗口中按照输出格式显示结果

案例 2

- 某公司根据市场需求计划加工生产三种产品，各产品的原材料消耗定额、工时定额、单位利润和最高资源限制如表 1。现要求安排三种产品的产量 X_1 ， X_2 ， X_3 ，满足在资源限制条件下使得总利润为最大。

数据

项目	甲产品 X1	乙产品 X2	丙产品 X3	资源限制
材料 1 (kg/ 件)	3	2	4	600kg
材料 2 (kg/ 件)	5	8	6	1030kg
材料 3 (kg/ 件)	2	5	3	688kg
加工能力 (工时 / 件)	2	3	2	495 工时
单位利润 (元 /	80	109	105	

数学模型

$$\begin{aligned} \max Z &= 80x_1 + 109x_2 + 105x_3 \\ s.t. \quad &\begin{cases} 3x_1 + 2x_2 + 4x_3 \leq 600 \\ 5x_1 + 8x_2 + 6x_3 \leq 1030 \\ 2x_1 + 5x_2 + 3x_3 \leq 688 \\ 2x_1 + 3x_2 + 2x_3 \leq 495 \\ x_j \geq 0 \quad (j = 1, 2, 3) \end{cases} \end{aligned}$$

- 利润系数向量 (80,109,105);
- 产品名 (" 甲 ", " 乙 ", " 丙 ");
- 资源消耗矩阵 $\begin{bmatrix} 3 & 5 & 5 & 2 \\ 2 & 8 & 5 & 3 \\ 4 & 6 & 3 & 2 \end{bmatrix}$
- 资源名称 (" 材料 1" , " 材料 2" , " 材料 3" , " 加工能力")
- 资源上限 (600,1030,688,495)

模型文件

```
{string} Products =...;
{string} Resources =...;
float a[Products][Resources] =...;
float c[Products] =...;
float b[Resources] =...;
dvar float+ x[Products];
maximize sum( p in Products ) c[p] * x[p];
subject to {
forall( r in Resources )
sum( p in Products ) a[p][r] * x[p] <= b[r];
}
execute {
  for(var p in Products)
    writeln("p1=",x[p]);
}
```



数据文件

- $\text{Products} = \{ \text{" 甲 "}, \text{" 乙 "}, \text{" 丙 "} \};$
- $\text{Resources} = \{ \text{" 材料 1"}, \text{" 材料 2"}, \text{" 材料 3"}, \text{" 加工能力 "} \};$
- $a = [[3, 5, 5, 2], [2, 8, 5, 3], [4, 6, 3, 2]];$
- $c = [80, 109, 105];$
- $b = [600, 1030, 688, 495];$

数据结构

整数做数组下标方法

```
int a[1..10];
```

```
int a[2..20];
```

```
range rows=1..10;
```

定义了一个整数范围 1 到 10

```
int a[rows];
```

定义了一个 10 个元素的数组，下标范围从 1 到 10

决策表达式 dexpr

如果一个表达式有明确的含义，可以把该表达式写成 决策表达式
决策表达式可以重复利用
模型的可读性更好



定义和使用决策表达式

maximize sum(p in Products) c[p] * x[p];

调整成

dexpr float profit=sum(p in Products) c[p] * x
[p];

maximize profit;

模型可读性更好



定义和使用决策表达式数组

定义

dexpr float resourcesusage

[r in Resources]=

sum(p in Products)a[p][r] * x[p];

模型调整为


forall(r in Resources)

resourcesusage[r] <= b[r];



优化后的模型

```
{string} Products =...;
{string} Resources =...;
float a[Products][Resources] =...;
float c[Products] =...;
float b[Resources] =...;
dvar float+ x[Products];
dexpr float resourcesusage[r in Resources]=sum( p in Products )a[p][r] * x[p];
dexpr float profit=sum( p in Products ) c[p] * x[p];
maximize profit;
subject to {
forall( r in Resources )
resourcesusage[r] <= b[r];
}
execute {
  for(var p in Products)
    writeln("p1=",x[p]);
}
```



案例 3

一个食用油加工企业通过混合 5 种食用油（2 种蔬菜油，3 种非蔬菜油）的方式生产一种产品。现制定 6 个月生产计划。每个月每种原油最多可以存储 1000 吨供以后使用。计划开始时，每种原油库存 500 吨。计划结束时也要保证每种原油库存 500 吨的量。每吨原油的存储价格是 5 个货币单位。产品不能存储。为了满足生产需求，原油需要购买。每种原油的价格随月份不同而有所变化。

案例 3

每个月所能加工的原油量有上限。

蔬菜油不超过 200 吨

非蔬菜油不超过 250 吨

在油的种类上也有限制

产品中最多由 3 种油混合成

如果产品中混有某种原材料，则至少混合 20 吨



案例 3

如果产品混有疏菜油，第 3 种非疏菜油必须混合在产品中

产品利润每吨 150 个货币单位

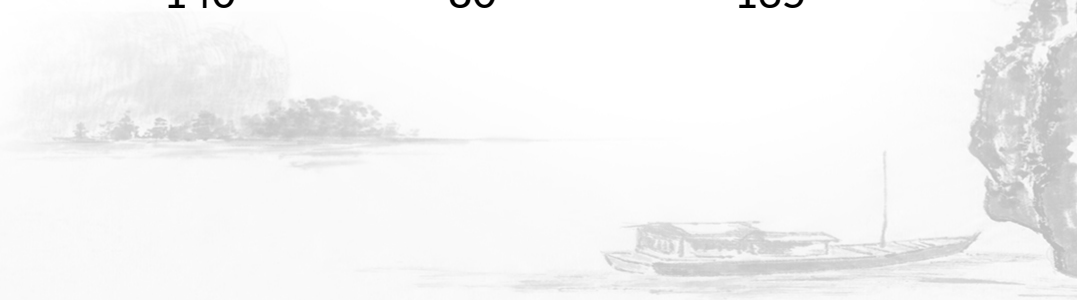
每种原材料的硬度如下表

原材料	V1	V2	o3	o4	o5
硬度	8.8	6.1	2	4.2	5

产品的硬度必须介于 3 和 6 之间

未来 6 个月原材料价格预测

价格	V1	V2	o1	o2	o3
M1	110	120	130	110	115
M2	130	130	110	90	115
M3	110	140	130	100	95
M4	120	110	120	120	125
M5	110	120	150	110	105
M6	90	100	140	80	135



思考

已知条件？

未知条件？

约束条件？

目标是什么？



已知条件

计划时间
购买成本
存储成本



未知条件

每个月买多少？

每个月用多少？

每个月存多少？

每个月生产多少产品？



foodmanufact1.mod

```
{string} Products = ...;
```

```
int NbMonths = ...;
```

```
range Months = 1..NbMonths;
```

```
float Cost[Months][Products] = ...;
```

```
dvar float+ Produce[Months];
```

```
dvar float+ Use[Months][Products];
```

```
dvar float+ Buy[Months][Products];
```

```
dvar float Store[Months][Products] in 0..1000;
```



数据

Products = { "v1", "v2", "o1", "o2", "o3" };

NbMonths = 6;

Cost = [[110.0, 120.0, 130.0, 110.0, 115.0],
[130.0, 130.0, 110.0, 90.0, 115.0],
[110.0, 140.0, 130.0, 100.0, 95.0],
[120.0, 110.0, 120.0, 120.0, 125.0],
[100.0, 120.0, 150.0, 110.0, 105.0],
[90.0, 100.0, 140.0, 80.0, 135.0]];

目标函数

maximize

$$\begin{aligned} & \text{sum}(m \text{ in Months}) \quad (150 * \text{Produce}[m] \\ & - \text{sum}(p \text{ in Products}) \text{Cost}[m][p] * \text{Buy}[m][p] \\ & - 5 * \text{sum}(p \text{ in Products}) \text{Store}[m][p]); \end{aligned}$$



约束条件

最后一个月每种原材料的库存是 500 吨

forall(p in Products)

Store[NbMonths][p] == 500;



约束条件

每个月所能加工的原油有上限的。

蔬菜油不超过 200 吨

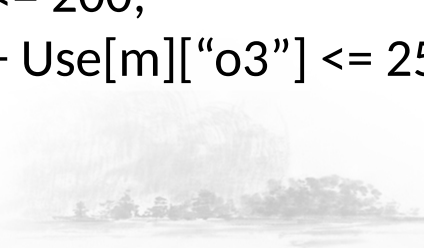
非蔬菜油不超过 250 吨

forall(m in Months) {

 Use[m]["v1"] + Use[m]["v2"] <= 200;

 Use[m]["o1"] + Use[m]["o2"] + Use[m]["o3"] <= 250;

}



约束条件

上月的存储量 + 购买量 = 当月使用量 + 当月存储量
(第一个月除外)

```
forall( m in Months )  
  forall( p in Products ) {  
    if (m == 1) {  
      500 + Buy[m][p] == Use[m][p] + Store[m][p];  
    }  
    else {  
      Store[m-1][p] + Buy[m][p] == Use[m][p] + Store[m][p];  
    }  
  }  
}
```

约束条件

硬度要求

```
forall( m in Months ) {  
  3 * Produce[m] <=  
    8.8 * Use[m]["v1"] + 6.1 * Use[m]["v2"] +  
    2 * Use[m]["o1"] + 4.2 * Use[m]["o2"] + 5 * Use[m]["o  
3"];  
  8.8 * Use[m]["v1"] + 6.1 * Use[m]["v2"] +  
  2 * Use[m]["o1"] + 4.2 * Use[m]["o2"] + 5 * Use[m]["o  
3"]  
  <= 6 * Produce[m];  
}
```

约束条件

当月产品产量 = 当月所有使用的原材料的量

forall(m in Months) {

Produce[m] == sum(p in Products) Use[m][p];

}



约束条件

每个月每种原材料至少混合 20 吨

```
forall( m in Months ) {
```

```
forall( p in Products )
```

```
    (Use[m][p] == 0) || (Use[m][p] >= 20);
```

```
}
```



约束条件

如果产品混有疏菜油，第 3 种非疏菜油必须混合在产品中

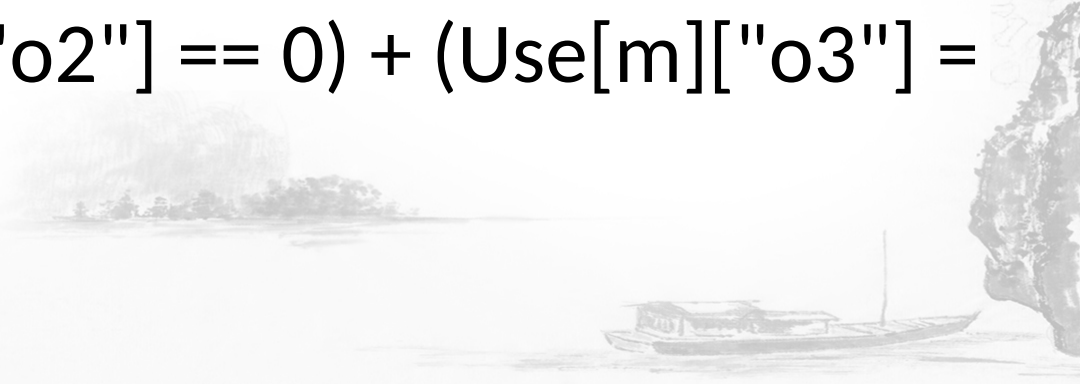
```
forall( m in Months ) {  
  (Use[m]["v1"] >= 20) || (Use[m]["v2"] >= 20) =>  
  Use[m]["o3"] >= 20;  
}
```



约束条件

产品中最多由 3 种油混合成

```
forall( m in Months ) {  
    (Use[m]["v1"] == 0) + (Use[m]["v2"] == 0) + (U  
se[m]["o1"] == 0) +  
        (Use[m]["o2"] == 0) + (Use[m]["o3"] =  
= 0) >= 2;
```



每个月原材料存储量最多 1000 吨
dvar float Store[Months][Products] in 0..1000;



数据结构 - 元组

下面举例说明 OPL 结构体 (Tuples) 的用法：

例，一个工厂有 3 种产品（面条，面包，蛋糕），各产品的市场需求量为 (100,200,300)。工厂可以自己生产产品，也外包生产。如果自己生产，每个产品消耗一定的资源（面粉和鸡蛋），资源总量为 (20,40)。如何确定每种产品自己生产和外包的产量，使得总费用最小。

		面条	面包	蛋糕
资源消耗	面粉	0.5	0.4	0.3
	鸡蛋	0.2	0.4	0.6
费用情况	自己生产	0.6	0.8	0.3
	外包	0.8	0.9	0.4

模型文件（part1）

容易得到这个线性规划模型的 OPL 程序：

```
{string} Products = ...;
```

```
{string} Resources = ...;
```

```
float consumption[Products][Resources] = ...;
```

```
float capacity[Resources] = ...;
```

```
float demand[Products] = ...;
```

```
float insideCost[Products] = ...;
```

```
float outsideCost[Products] = ...;
```

模型文件 (part2)

minimize

**sum(p in Products) (insideCost[p]*inside[p] +
outsideCost[p]*outside[p]);**

subject to {

forall(r in Resources)

sum(p in Products) consumption[p][r] * inside[p] <= capacity[r];

forall(p in Products)

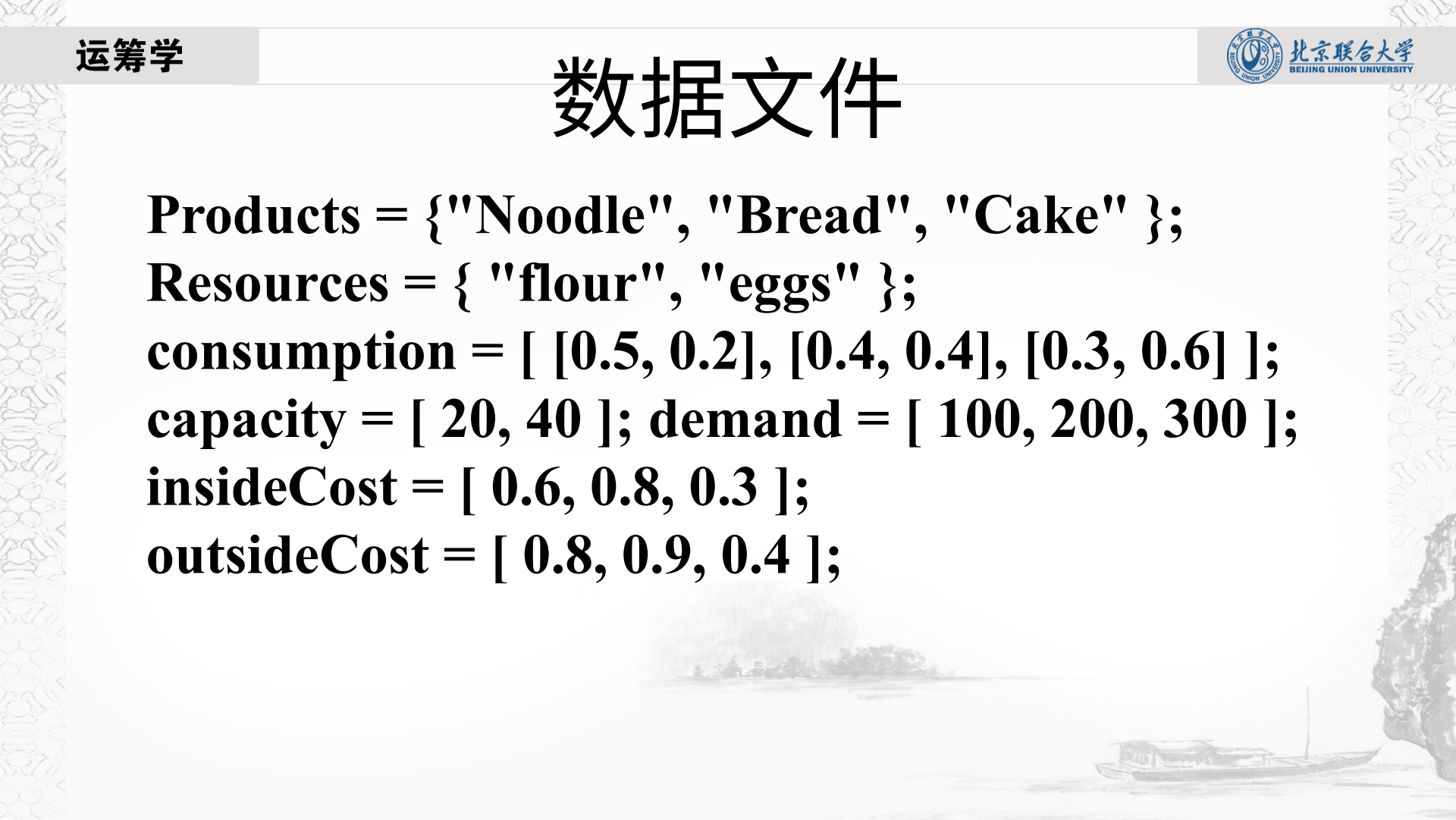
inside[p] + outside[p] >= demand[p];

}



数据文件

Products = {"Noodle", "Bread", "Cake" };
Resources = { "flour", "eggs" };
consumption = [[0.5, 0.2], [0.4, 0.4], [0.3, 0.6]];
capacity = [20, 40]; demand = [100, 200, 300];
insideCost = [0.6, 0.8, 0.3];
outsideCost = [0.8, 0.9, 0.4];



模型优化

但是，从数据分离的角度来说，上述模型仍然有问题。

demand, insideCost, outsideCost, consumption 都是关于 Products 的相关信息，但是被定义成独立的数组，这样模型可读性差，不宜于维护且容易修改出错。

利用 OPL 的 Tuples 是一个解决问题的办法。原来的程序可以修改为：

```
{string} Products = ...;  
{string} Resources = ...;
```

```
tuple ProductData {  
    float demand;  
    float insideCost;  
    float outsideCost;  
    float consumption[Resources];
```

相对于声明一个结构体类型
注意：末尾无分号！！

相对于定义一个结构体数组

模型优化

```
dvar float+ inside[Products];  
dvar float+ outside[Products];
```

相对于使用结构体变量中的成员
(也用一个点取其成员)

```
minimize  
    sum(p in Products) (product[p].insideCost*inside[p] +  
        product[p].outsideCost*outside[p]);  
subject to {  
    forall(r in Resources)  
        sum(p in Products) product[p].consumption[r] * inside[p] <=  
capacity[r];  
    forall(p in Products)  
        inside[p] + outside[p] >= product[p].demand;  
}
```

数据文件

数据文件修改为：

```
Products = { "Noodle", "Bread", "Cake" };  
Resources = { "flour", "eggs" };
```

```
product = #[
```

```
    Noodle : < 100, 0.6, 0.8, [ 0.5, 0.2 ] > ,
```

```
    Bread : < 200, 0.8, 0.9, [ 0.4, 0
```

```
    Cake : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
```

```
    ]#;
```

```
capacity = [ 20, 40 ];
```

Product 的初始化也可以简化写为

```
product = [
```

相对于初始化结构体数组

注意：每个结构体变量初始化使用 < 和 >

后续处理

ILOG 提供脚本（Script）可以帮助显示程序运行的结果。

在前面的 `.mod` 文件的末尾加入以下代码：

函数：执行脚本

```
execute{  
writeln("Show Results: ");  
writeln("inside =: ",inside);  
writeln("outside =: ",outside);
```

输出一行信息

```
for(p in Products)  
    writeln("inside["+p+"] reducedCost = ", inside[p].reducedCost);  
}
```

比较 C 语言的 `sprintf` 函数

后续处理

ILOG 提供脚本（Script）可以设置一些 CPLEX 参数

在前面的 .mod 文件的末尾加入以下代码：

```
execute PARAMS {  
  cplex.tilim = 100;  
}
```

最大求解时间 time limit=100 秒

CPLEX 和 OPL 有很多可以设置的参数，具体可以参见帮助文档中的“CPLEX Parameters and OPL Parameters”

线性规划的典型应用

数据包络分析
混合配料问题



数据包络分析

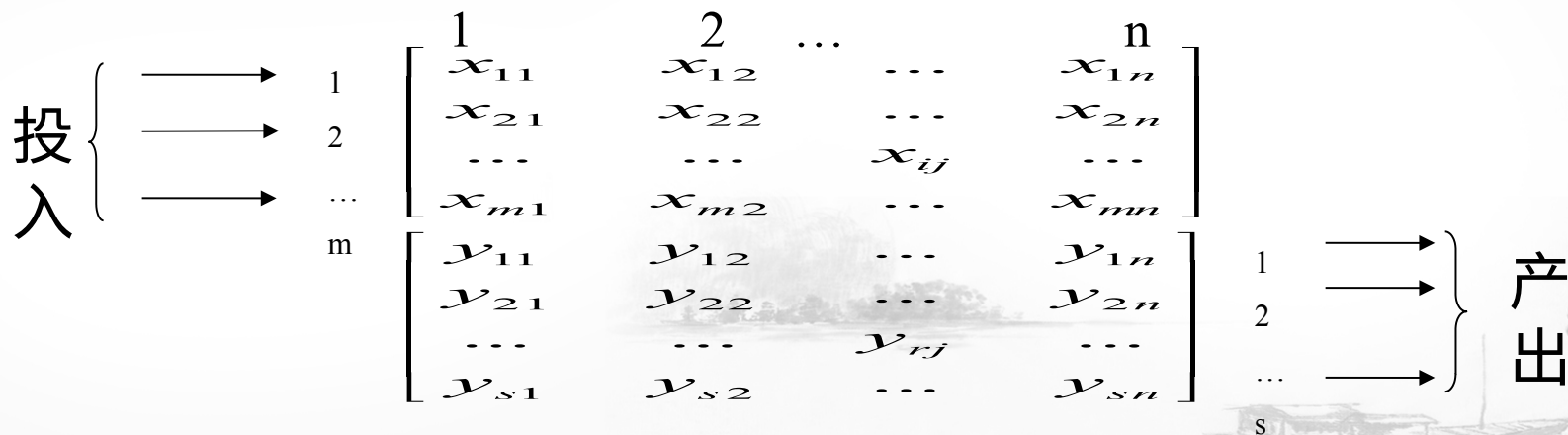
根据多项投入指标和多项产出指标，利用线性规划的方法，对具有可比性的同类型单位进行相对有效性评价的一种数量分析方法。

生产前沿面
DEA 有效



数据包络分析

在 DEA 中通常称被衡量绩效的组织为决策单元 (Decision Making Unit, DMU)。决策单元的个数指的是有多少并列的机构。第一个下标指的是投入指标号; 第二个下标指的是机构号。N 代表机构号。



数据包络分析

数学模型

$$\min E$$

$$s.t. \begin{cases} \sum_{j=1}^n \lambda_j y_{rj} \geq y_{rj_0} \quad (r = 1, \dots, s) \\ \sum_{j=1}^n \lambda_j x_{ij} \leq E x_{ij_0} \quad (i = 1, \dots, m) \\ \sum_{j=1}^n \lambda_j = 1, \lambda_j \geq 0 \quad (j = 1, \dots, n) \end{cases}$$

当求解结果有 $E < 1$ 时，则 j_0 决策单元非 DEA 有效，否则 j_0 决策单元 DEA 有效。

数据包络分析

例 8 振华银行的 4 个分理处的投入产出情况如表 1-16 所示。要求分别确定各分理处的运行是否有效。

产出单位：处理笔数 / 月

分理处	投入		产出		
指标	职员数	营业 面积 (m ²)	储蓄存 取	贷款	中间业务
分理处 1	15	140	1800	200	1600
分理处 2	20	130	1000	350	1000
分理处 3	21	120	800	450	1300
分理处 4	20	135	900	420	1500

数据包络分析

解 先确定分理处 1 的运行是否 DEA 有效。

线性规划模型如下：

$$\begin{aligned} & \min E \\ & s.t. \begin{cases} 1800\lambda_1 + 1000\lambda_2 + 800\lambda_3 + 900\lambda_4 \geq 1800 \\ 200\lambda_1 + 350\lambda_2 + 450\lambda_3 + 420\lambda_4 \geq 200 \\ 1600\lambda_1 + 1000\lambda_2 + 1300\lambda_3 + 1500\lambda_4 \geq 1600 \\ 15\lambda_1 + 20\lambda_2 + 21\lambda_3 + 20\lambda_4 \leq 15E \\ 140\lambda_1 + 130\lambda_2 + 120\lambda_3 + 135\lambda_4 \leq 140E \\ \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1 \\ \lambda_j \geq 0 (j = 1, \dots, 4) \end{cases} \end{aligned}$$

求解结果为 $E=1$ ，说明分理处 1 的运行行为 DEA 有效。

数据包络分析

同理计算其他三个分理处的 E 值，结果为分理处 3 和分理处 4， $E=1$ ；

对于分理处 2，有 $E=0.966$ ；

此时 $\lambda_1=0.28, \lambda_3=0.72, \lambda_2=\lambda_4=0$ ，即分理处 2 运行非 DEA 有效。理由为：若将 28% 的分理处 1 同 72% 分理处 3 组合，其各项产出不低于分理处 2 的各项产出，但其投入只有分理处 2 的

ILOG 模型数据

- nbInputs=2;
- nbOutputs=3;
- nameOfDMUs={"1","2","3","4"};
- Inputs={<"1" 15 140>,
<"2" 20 130>,
<"3" 21 120>,
<"4" 20 135>;
- Outputs={<"1" 1800 200 1600>,
<"2" 1000 350 1000>,
<"3" 800 450 1300>,

DEA 模型 (part1)

```
int nbInputs=...;
int nbOutputs=...;
{string} nameOfDMUs=...;
tuple input{
    string DMU;
    int nbemployees;
    float area;
}
tuple output{
    string DMU;
    float deposit;
    float loan;
    float midbusiness;
}
{input} Inputs=...;
{output} Outputs=...;
tuple infoOfDMU
{
    key string name;
    int nbemployees;
    float area;
    float deposit;
    float loan;
    float midbusiness;
}
```



DEA 模型 (part2)

```
minimize E;
subject to
{
    sum(input in Inputs) input.nbemployees*lamda[input.DMU]<=item(allinfo,0).nbemployees*E;
    sum(input in Inputs)input.area*lamda[input.DMU]<=item(allinfo,0).area*E;
    sum(output in Outputs) output.deposit*lamda[output.DMU]>=item(allinfo,0).deposit;
    sum(output in Outputs) output.loan*lamda[output.DMU]>=item(allinfo,0).loan;
    sum(output in Outputs) output.midbusiness*lamda[output.DMU]>=item(allinfo,0).midbusiness;
    sum(name in nameOfDMUs)lamda[name]==1;
}
```

混合配料问题

某糖果厂用原料 A , B , C 加工成三种不同牌号的糖果甲、乙、丙。已知各种牌号糖果中 A、B、C 含量、原料成本、各种原料的每月限制用量, 三种牌号糖果的单位加工费及售价, 如表 1-17 所示。问该厂每月生产这三种牌号糖果各多少 kg , 才能使其获利最大。试建立这个问题

的线性规划的数学模型。

原料	甲	乙	丙	原料成本 (元 / kg)	每月限制 用量 (kg)
A	$\geq 60\%$	$\geq 30\%$		2.00	2000
B				1.50	2500
C	$\leq 20\%$	$\leq 50\%$	$\leq 60\%$	1.00	1200
加工费 (元 / kg)	0.50	0.40	0.30		
售价 (元 / kg)	3.40	2.85	2.25		

混合配料问题

解 用 $i=1,2,3$ 分别代表原料 A, B, C, 用 $j=1,2,3$ 分别代表甲、乙、丙三种糖果, x_{ij} 为生产第 j 种糖果耗用的第 i 种原料的 kg 数量。该厂的获利为三种牌号糖果的售价减去相应的加工费和原料成本, 三种糖果的生产量分别为:

$$\begin{aligned}
 \max z &= (3.40 - 0.50)(x_{11} + x_{21} + x_{31}) + (2.85 - 0.40)(x_{12} + x_{22} + x_{32}) \\
 &+ (2.25 - 0.30)(x_{13} + x_{23} + x_{33}) - 2.0(x_{11} + x_{12} + x_{13}) \\
 &- 1.50(x_{21} + x_{22} + x_{23}) - 1.0(x_{31} + x_{32} + x_{33}) \\
 &= 0.9x_{11} + 1.4x_{21} + 1.9x_{31} + 0.45x_{12} + 0.95x_{22} + 1.45x_{32} \\
 &- 0.05x_{13} + 0.45x_{23} + 0.95x_{33}
 \end{aligned}$$

混合配料问题



$$\text{s.t.} \left\{ \begin{array}{l} (x_{11} + x_{121} + x_{13}) \leq 2000 \\ x_{21} + x_{22} + x_{23} \leq 2500 \\ x_{31} + x_{32} + x_{33} \leq 1200 \\ x_{11} \geq 0.6(x_{11} + x_{21} + x_{31}) \\ x_{31} \leq 0.2(x_{11} + x_{21} + x_{31}) \\ x_{12} \geq 0.3(x_{12} + x_{22} + x_{32}) \\ x_{32} \leq 0.5(x_{12} + x_{22} + x_{32}) \\ x_{33} \leq 0.6(x_{13} + x_{23} + x_{33}) \\ x_j \geq 0 (i = 1, 2, 3; j = 1, 2, 3) \end{array} \right.$$

原料月供应量限制

含量成分的限制



混合配料模型

- `{string} products={" 甲 "," 乙 "," 丙 "};`
- `{string} components={"A","B","C"};`
- `float costofcomponents[components]=[2,1.5,1];`
- `float capacitiesofcomponents[components]=[2000,2500,1200];`
- `float costofproducts[products]=[0.5,0.4,0.3];`
- `float profitofproducts[products]=[3.4,2.85,2.25];`
- `dvar float+ x[products][components];`
- `dexpr float totalprofit=sum(p in products,c in components)(x[p][c]*profitofproducts[p]-x[p][c]*costofcom
ponents[c]-x[p][c]*costofproducts[p]);`
- `maximize totalprofit;`
- `subject to`
- `{`
- `forall(c in components)`
- `sum(p in products)x[p][c]<=capacitiesofcomponents[c];`
- `x[" 甲 "]["A"]>=0.6*sum(c in components,p in products:p==" 甲 ")x[p][c];`
- `x[" 甲 "]["C"]<=0.2*sum(c in components,p in products:p==" 甲 ")x[p][c];`
- `x[" 乙 "]["A"]>=0.3*sum(c in components,p in products:p==" 乙 ")x[p][c];`
- `x[" 乙 "]["C"]<=0.5*sum(c in components,p in products:p==" 乙 ")x[p][c];`

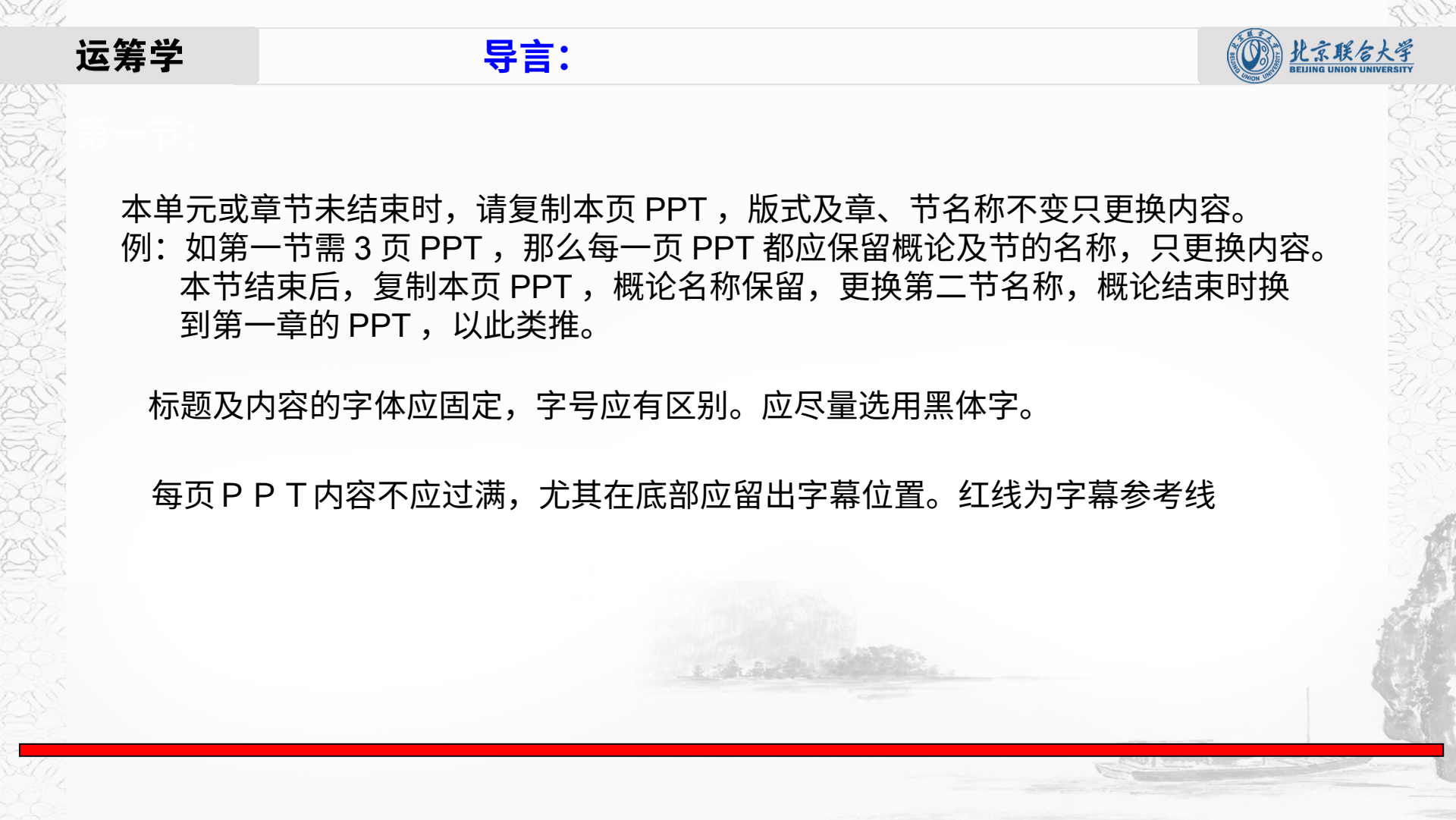
结束



本单元或章节未结束时，请复制本页 PPT，版式及章、节名称不变只更换内容。
例：如第一节需 3 页 PPT，那么每一页 PPT 都应保留概论及节的名称，只更换内容。
本节结束后，复制本页 PPT，概论名称保留，更换第二节名称，概论结束时换到第一章的 PPT，以此类推。

标题及内容的字体应固定，字号应有区别。应尽量选用黑体字。

每页 P P T 内容不应过满，尤其在底部应留出字幕位置。红线为字幕参考线



第一节：

第二节：

第三节：

第四节：

第五节：

第六节：



第一节：



第一节：

第二节：

第三节：

第四节：

第五节：

第六节：



第一节：



第一节：

第二节：

第三节：

第四节：

第五节：

第六节：



第一节：



第一节：

第二节：

第三节：

第四节：

第五节：

第六节：



第一节：



第一节：

第二节：

第三节：

第四节：

第五节：

第六节：



第一节：

