N.N. Pisaruk

Optimization in Operations Management

Minsk 2012

N.N. Pisaruk. *Optimization in operations management*.

The goal of operations managers is to efficiently use raw materials, human resources, equipment, and facilities in supplying quality goods or services. Entry-level operations managers determine how best to design, supply, and run business processes. Senior operations managers are responsible for setting strategic decisions from an operations standpoint, deciding what technologies should be used, where facilities should be located, and managing the facilities making products or providing services.

The book discusses diverse applications of optimization in operations management. We do not restrict ourselves to formulating mathematical models. Anyone who has ever attempted to apply mathematical programming in practice knows that it is usually not a simple and straightforward exercise. To facilitate this task, optimization modeling languages were designed to easily implement optimization problems as computer programs. All the models considered in the book are converted into computer programs by means of **MIPshell —** an environment that facilitates modeling and solving mixed-integer programming problems with **MIPCL** (Mixed-Integer Programming Class Library). These simplified but practical computer programs can be used as templates when developing applications for supporting particular operations management decisions.

Aimed at undergraduates, postgraduates, research students and managers, this book shows how optimization is used in operations management.

Contents

Preface

Operations management may be defined as the design, operation, and improvement of the production systems that create the firm's primary products and services. Operations decisions are made in the context of the firm as a whole (see Figure 1). Starting from its marketplace (the firms's customers for its products and services), the firm determines its *corporate strategy*. This strategy is based on the corporate mission, and in essence, it reflects how the firm plans to use all its resources and functions (marketing, finance, and operations) to gain competitive advantage. The *operations strategy* specifies how the firm will employ its production capabilities to support its corporate strategy. The *marketing strategy* addresses how the firm will sell and distribute its goods and services, and the *finance strategy* identifies how best to utilize the firm's financial resources.

Operations management decisions can be divided into three broad categories:

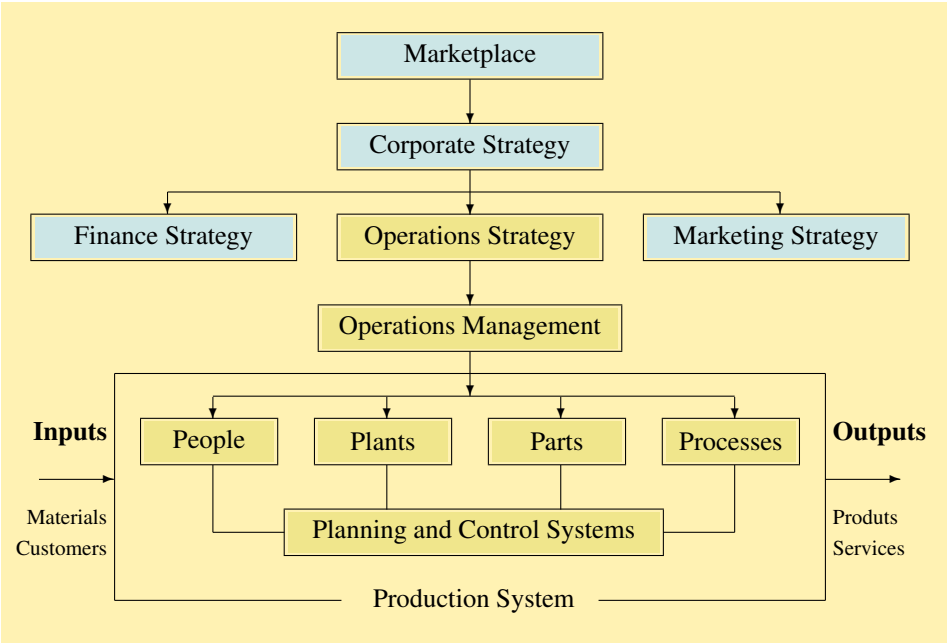

Figure 1: Firm's decision chart

- strategic (long-term) decisions;
- tactical (intermediate-term) decisions;
- operational planning and control (short-term) decisions.

Strategic issues usually address such questions as:

- How will we make the product?
- Where do we locate the facilities and what are their capacities?
- When should we add more capacity?

The time horizon for strategic decisions is typically very long. These decisions impact firm's long-range effectiveness in terms of how it can address its customer's needs. Thus, for the firm to succeed, strategic decisions must be consistent with firm's corporate strategy. These decisions become operating constraints under which firm's decisions are made in both the intermediate and short term.

At the next level in the decision-making process, tactical planning primarily addresses how to efficiently use materials and labor within the constraints of previously made strategic decisions. Issues to be managed on this level are:

- How many workers do we need, and when do we need them?
- Should we work overtime or put a second shift?
- When should we have material delivered?
- Should we have raw materials and finished products inventory?

Tactical decisions, in turn, become the operating constraints for making operational planning and control decisions. Issues at this level include:

- What jobs be done today or this week?
- Who is assigned to a particular task?
- What jobs are most urgent?

The hart of operations management is the management of production systems. A *production system* uses operations resources to transform inputs (raw materials, customer demands, or finished products from another production system) into outputs (finished products or services). *Operations resources* include people, plants, parts, processes, and planning and control systems (*five P's of operations management*).

People are the direct and indirect workforce. *Plants* include the factories or service units where production is carried out. *Parts* comprise the materials or supplies. *Processes* include equipment and technological processes. *Planning and control systems* perform procedures and information management to operate the production system.

Chapter 1

Aggregate Production Planning

Aggregate production planning is intermediate-range planning (usually covers a planning horizon from 6 to 18 months) that is concerned with determining production rates for some product groups. An aggregate plan is to find an optimal trade off of production rates, used resources, and inventory levels. In this chapter we consider aggregate production planning of different complexity. Usually, these models are integrated into supply chain systems that manage the flows of information, materials, and services from raw material supplies through factories and warehouses to the end customers.

1.1 Simple Inventory Models

Inventories are stocks of goods being held for future use or sale. Any company dealing with physical products (, wholesalers, and retailers) maintains inventories. Manufacturers need inventories of the materials required to produce their products. They also need inventories of the finished products awaiting shipment. Similarly, both wholesalers and retailers need to maintain inventories of goods to avoid shortages of these goods.

On the one hand, inventories are like money placed in a drawer, assets tied up in investments that are not producing any return and, in fact, incurring a borrowing cost (the cost of capital tied up in inventory). They also incur costs for the care of the stored material (storage cost, insurance, taxes) and are subject to spoilage and obsolescence. On the other hand, inventories provide a stable source of raw materials required for production. A large inventory requires fewer replenishments and may reduce ordering costs because of economies of scale.

1.1.1 Economic Order Quantity (EOQ)

Assumptions:

- one product;
- consumption is instantaneous;

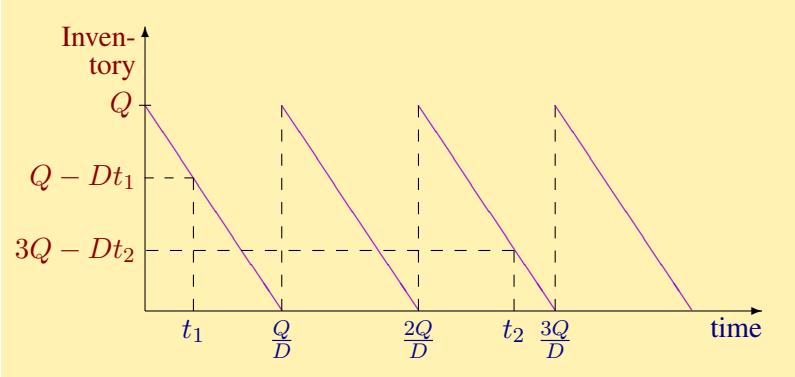

Figure 1.1: Inventory Level in EOQ

- zero lead time (the time period between the placement of order and receipt of goods);
- constant demand of D units of product per time unit;
- fixed buying cost, C, for each item;
- fixed ordering cost, F, for each batch;
- fixed holding cost, H, for each item.

Typically, holding (or inventory carrying) cost includes:

- insurance cost: 2%;
- maintenance cost: 6%;
- opportunity cost of alternative investment: 7-10%

If the size of a lot is Q, then the total cost per *consumption cycle* (time interval of length $T = Q/D$) is

$$F + CQ + \int_0^{Q/D} h(Q - Dt)dt = F + CQ + \frac{HQ^2}{2D}.$$

The average cost per unit of time is

$$c(Q) \stackrel{\text{def}}{=} \frac{FD}{Q} + \frac{H}{2}Q + CD.$$

A graphical representation of EOQ model is give in Figure 1.1. We see that the stock is replenished at the beginning of each cycle time interval.

Minimizing $c(Q)$, we find the optimal size, Q^*, of a batch (lot), which is the positive solution of the equation $c'(Q) = -\frac{FD}{Q^2} + \frac{H}{2} = 0$. Hence,

$$Q^* = \sqrt{\frac{2DF}{H}} \tag{1.1}$$

and the optimal cycle time is

$$T^* = Q^*/D = \sqrt{\frac{2F}{HD}}. \tag{1.2}$$

Let us note that the optimal size of a batch does not depend on the item cost C.

Example 1.1 *The deman for electical components is* 2500 *units per month. For the store, to make an order it costs* \$300. *One item costs* \$3. *The annual inventory holding cost rate is* 20%. *What are optimal lot size and cycle time?*

Solution. The input for this example is: $D = 2500$, $F = 300$, $C = 3$, and $H = (3 \times 0.2)/12 = 0.05$. Therefore, by (1.1) and (1.2),

$$Q^* = \sqrt{\frac{2 \cdot 2500 \cdot 300}{0.05}} \approx 5477.226, \quad T^* = Q^*/D \approx 5477.226/2500 \approx 2.19.$$

Since the product is indivisible, the lot size must be integral. We could first round down and then round up Q^* to choose from two values, $Q_1 = 5477$ and $Q_2 = 5455$, the one with the smallest value of $c(Q)$:

$$c(Q_1) = (300 \cdot 2500)/5574 + (0.05 \cdot 5574)/2 + 3 \cdot 2500 \approx 7773.903,$$
$$c(Q_2) = (300 \cdot 2500)/5575 + (0.05 \cdot 5575)/2 + 3 \cdot 2500 \approx 7773.904.$$

Thus, we conclude that the best lot size is $Q_2 = 5575$ with cycle time of $T_2 = Q_2/D = 5575/2500 = 2.23$ months. □

1.1.2 Economic Production Quantity (EPQ)

Assumptions:

- production is instantaneous;
- constant production rate of P units of product per time unit;
- constant demand of D units of product per time unit;
- fixed setup cost, F, for each batch;
- fixed unit production cost, C, for each item;
- fixed holding cost, H, for each item.

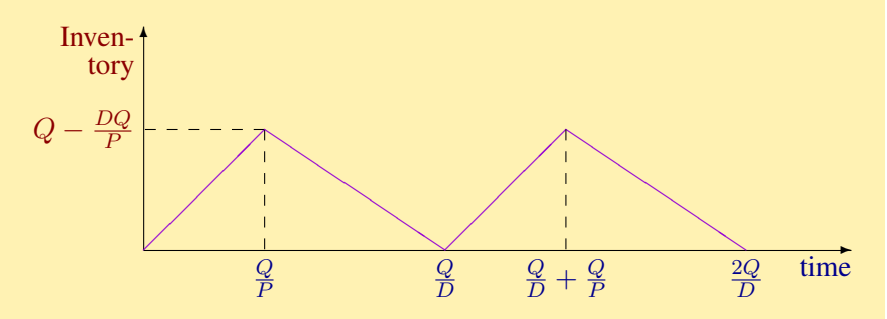

Figure 1.2: Inventory Level in EPQ

It is assumed that $P > D$. If the size of a batch (lot) is Q, then the production time is $T_p = Q/P$. It is also assumed that production starts at the beginning of each *production cycle* of $T_c = Q/D$ units of time.

A graphical representation of EPQ model is given in Figure 1.2. We see that during the production time the stock is being replenished to the maximum value $Q - \frac{DQ}{P}$, and when pro at the beginning of each cycle time interval.

The total cost per production cycle is

$$F + H \int_0^{Q/P} (P - D)t\, dt + h \int_{Q/P}^{Q/D} (Q - Dt)dt = F + + \frac{HQ^2(P - D)}{2PD}.$$

The average cost per unit of time is

$$c(Q) \stackrel{\text{def}}{=} \frac{FD}{Q} + \frac{HQ(P - D)}{2PD}.$$

Minimizing $c(Q)$, we find the optimal size, Q^*, of a batch (lot), which is the positive solution of the equation

$$c'(Q) = -\frac{FD}{Q^2} + \frac{H(P - D)}{2P} = 0.$$

Hence,

$$Q^* = \sqrt{\frac{2PDF}{H(P - D)}} \tag{1.3}$$

and the optimal production time and production cycle are

$$T_p^* = Q^*/P, \quad T_c^* = Q^*/D. \tag{1.4}$$

Example 1.2 (Manufacturing Speakers for TV Sets) *A television manufacturing company produces its own speakers, which are used in the production of its television sets. The television sets are assembled on a continuous production line at*

a rate of $10\,000$ *per month, with one speaker needed per set. The production rate of speakers is* $40\,000$ *per month, i.e., two times higher than the production rate of TV sets. Therefore, there is no need to produce speakers on a continuous production line, and the speakers are produced in batches and then the speakers are placed into inventory until they are needed for assembly into television sets on the production line.*

Each time a batch is produced, a setup cost of $\$5\,000$ *is incurred. The unit production cost of a single speaker (excluding the setup cost) is* $\$10$*. The estimated holding cost of keeping a speaker in stock is* $\$0.30$ *per month.*

The company is interested in determining when to produce a batch of speakers and how many speakers to produce in each batch.

Solution. In this example $P = 40\,000$, $D = 10\,000$, $F = 12\,000$, $C = 10$, $H = 0.3$. By (1.3),

$$Q^* = \sqrt{\frac{2 \cdot 40000 \cdot 10000 \cdot 5000}{0.3 \cdot (40000 - 10000)}} \approx 21081.85,$$

and since

$$c(\lfloor Q^* \rfloor) = \frac{12000 \cdot 10000}{21081} + \frac{0.3 \cdot 21081(40000 - 10000)}{2 \cdot 40000 \cdot 10000} = 5692.57,$$
$$c(\lceil Q^* \rceil) = \frac{12000 \cdot 10000}{21082} + \frac{0.3 \cdot 21082(40000 - 10000)}{2 \cdot 40000 \cdot 10000} = 5692.30,$$

the optimal batch size is $\bar{Q} = 21082$, and the production must start at the beginning of each production cycle of

$$\bar{T}_c = \bar{Q}/D = 21082/10000 \approx 2.11 \text{ months}.$$

□

1.2 Single Product Lot-Sizing

The assumptions made in sections 1.1.1 and 1.1.2 are too ideal, and, therefore, may be too restrictive in practice. The demand for a product is rarely constant because of season fluctuations and many other reasons. The cost of product also depends on the season. In this section we consider a more realistic model. The problem is to decide on a production plan for a single product within a T-period horizon. For each period t the following parameters are known:

f_t: fixed production setup cost;

p_t: unit production cost;

h_t: unit inventory cost;

d_t: product demand;

u_t: production (or transportation) capacity.

The problem is to determine the amounts of the product to be produced in each of T periods so that the demands in all periods are fully satisfied, the production (transportation) capacities are not violated, and the total cost of producing and storing the product is minimum.

To formulate this problem as a *mixed-integer program* (*MIP*), we introduce the following variables:

x_t: amount of product produced in period t;

s_t: stock at the end of period t;

y_t: with $y_t = 1$ if production occurs in period t, and $y_t = 0$ otherwise.

With these variables the problem is formulated as follows:

$$\sum_{t=1}^{T} p_t x_t + \sum_{t=1}^{T} h_t y_t + \sum_{t=1}^{T} f_t y_t \to \min \tag{1.5a}$$

$$s_{t-1} + x_t = d_t + s_t, \quad t = 1, \ldots, T, \tag{1.5b}$$

$$x_t \le u_t y_t, \quad t = 1, \ldots, T, \tag{1.5c}$$

$$s_t, x_t \ge 0, \; y_t \in \{0, 1\}, \quad t = 1, \ldots, T. \tag{1.5d}$$

Let us note that the initial stock, s_0, is not a variable but an input parameter.

The balancing constraints (1.5b) ensure a proper transition from one period to the next, e.g., the inventory from period $t-1$ plus the production in period t equals the demand (sales) in period t plus the inventory to the next period.

The capacity constraints (1.5c) (such constraints are also called *variable upper bounds*) say that in any period t the product is produced only if the production occurs in this period ($y_t = 1$), and the amount produced, x_t, does not exceed the production capacity for this period.

The objective is to minimize the total production cost.

1.2.1 MIPshell Implementation

We developed a C++ class **Clotsize** that solves the Single Product Lot-Size Problem instances written in text files. **Clotsize** has the following members:

- *m_iT*: number of time periods;
- *m_ipPrice*: integer array of size *m_iT*, where *m_ipPrice*[t] is unit production cost in period t;
- *m_ipFixedCost*: integer array of size *m_iT*, where *m_ipFixedCost*[t] is fixed cost of producing in period t;

- *m_ipHoldCost*: integer array of size *m_iT*, where *m_ipHoldCost*[t] is unit storage cost in period t;
- *m_ipDemand*: integer array of size *m_iT*, where *m_ipDemand*[t] is demand in period t;
- *m_ipCapacity*: integer arrays of size *m_iT*, where *m_ipCapacity*[t] is production capacity in period t.

The default constructor of **Clotsize**

Clotsize(**const char** **name*)

gets as its input parameter a name of a text file (without extention **.txt**) describing an instance of the Single Product Lot-Size Problem, and then calls the function

void ***readData***(**const char** **fileName*)

to read the file.

MIPshell implementation of IP (1.5) is given in Listing 1.1.

Listing 1.1: **MIPshell** model for single product lot-size problem

```
#define p(t) m_ipPrice[t]
#define h(t) m_ipHoldCost[t]
#define f(t) m_ipFixedCost[t]
#define d(t) m_ipDemand[t]
#define c(t) m_ipCapacity[t]

int Clotsize::model()
{
  int t, T=m_iT, s0=m_iInitStock;
  VAR_VECTOR x("x",REAL_GE,T), s("s",REAL_GE,T), y("y",BIN,T);

  minimize(sum(t in [0,T)) (p(t)*x(t) + h(t)*s(t) + f(t)*y(t)));
  s0 + x(0) == d(0) + s(0);
  x(0) <= c(0)*y(0);
  forall(t in [1,T)) {
    s(t-1) + x(t) == d(t) + s(t);
    x(t) <= c(t)*y(t);
  }

  optimize();
  printSol(x,s);
  return 0;
} // end of Clotsize::model
```

You can find a complete Single Product Lot-Size **MIPshell** application in the following folder

$MIPDIR/examples/mipshell/lotsize.

1.3 Lot-Sizing With Backordering and Overtime Production

When production capacities are not sufficient to meet all demands in required terms, sometimes it is allowed that a product demanded in period τ can be delivered in later period t ($\tau < t$) with a discount of $b_{t\tau}$ that includes cost of expediting, loss of customer goodwill (usually hard to measure), and a possible discount. This is called *backordering* or *backlogging*.

In some cases it is also convenient to divide time periods into a number of production periods. For example, we may produce in regular time and in overtime. Overtime production is, of course, more expensive. Therefore, let us assume that there are T_p production periods, and f_τ, p_τ, u_τ are, respectively, fixed production cost, unit production cost, and production capacity in production period $\tau = 1, \ldots, T_p$. As before, h_t and d_t denote unit storage cost and demand in period $t = 1, \ldots, T$.

To model backordering, we use the following variables:

$x_{\tau t}$: amount of product produced in production period τ to satisfy demand of period t;

$y_\tau = 1$ if production occurs in production period τ, and $y_\tau = 0$ otherwise.

With these variables the model (1.5) is extended as follows:

$$\sum_{\tau=1}^{T_p} \sum_{t=1}^{T} c_{\tau t} x_{\tau t} + \sum_{\tau=1}^{T_p} f_\tau y_\tau \to \min \tag{1.6a}$$

$$\sum_{\tau=1}^{T_p} x_{\tau t} = d_t, \quad t = 1, \ldots, T, \tag{1.6b}$$

$$\sum_{t=1}^{T} x_{\tau t} \le u_\tau y_\tau, \quad \tau = 1, \ldots, T_p, \tag{1.6c}$$

$$x_{\tau t} \ge 0, \quad \tau = 1, \ldots, T_p, \ t = 1, \ldots, T, \tag{1.6d}$$

$$y_\tau \in \{0, 1\}, \quad \tau = 1, \ldots, T_p. \tag{1.6e}$$

Here the costs $c_{\tau t}$ are defibed by the rule:

$$c_{\tau t} \stackrel{\text{def}}{=} \begin{cases} p_\tau, & t = \mathcal{T}(\tau), \\ p_\tau + b_{\tau t}, & t < \mathcal{T}(\tau), \\ p_\tau + \sum_{\bar{t}=t}^{\tau-1} h_{\bar{t}}, & t > \mathcal{T}(\tau), \end{cases}$$

where $\mathcal{T}(\tau)$ denote the time period that contains production period τ.

The formulation (1.6) does not take into account initial stock. But we can correct this by introducing in the first time period a production period, say indexed with 0, with production capacity u_0 equal to the initial stock, and production cost $p_0 = 0$.

Let us note in the conclusion that the famous transportation problem is a special case of problem (1.6) when all fixed costs f_τ are zeroes (what implies that all y_τ may be fixed to 1).

1.3.1 MIPshell Implementation

As usual, to solve Lot-Sizing Problem With Backlogging, we developed a $C++$ class named **Cbklotsize** which definition is given in Listing 1.2.

Listing 1.2: **MIPshell** Class **Cbklotsize**: definition

```
#include <mipshell.h>

class Cbklotsize: public CProblem
{
  int m_iT, // number of time periods
      m_iTp, // number of production periods
      *m_ipFxCost, *m_ipCapacity, *m_ipDemand, *m_ipCost;
public:
  Cbklotsize(const char* name);
#ifdef __THREADS_
  Cbklotsize(const Cbklotsize &other, int thread);
  CMIP* clone(const CMIP *pMip, int thread);
#endif
  virtual ~Cbklotsize();

  int model();
// implementation
  void readData(const char* fileName);
  void printSol(VAR_VECTOR &x, VAR_VECTOR &y);
};
```

Cbklotsize has the following members:

- *m_iT*: number of demand periods;
- *m_iTp*: number of production periods;
- *m_ipDemand*: integer array of size *m_iT*, where *m_ipDemand*[t] is demand in period t;

- *m_ipCapacity*: integer arrays of size *m_iTp*, where *m_ipCapacity*[tau] is production capacity in production period tau.
- *m_ipFxCost*: integer array of size *m_iTp*, where *m_ipFxCost*[tau] is fixed cost of producing in production period tau;
- *m_ipCost*: integer array of size *m_iTp*×*m_iT*, where *m_ipCost*[tau×*m_iT*+t] is fixed cost of producing in production period tau for demand period t.

The constructor

```
Cbklotsize::Cbklotsize(const char* name): CProblem(name) {
  m_ipDemand=0;
  char fileName[128];
  strcpy(fileName,name);
  strcat(fileName,".txt");
  readData(fileName);
}
```

gets as its input parameter a name of a text file (without extension **.txt**) describing an instance of the Lot-Sizing Problem With Backlogging, and then calls the function

```
void readData(const char *fileName)
```

to read the file which format will be explained in the next section.

MIPshell implementation of IP (1.6) in Listing 1.3 is almost straightforward. Here we adopt the convention that if some $c(\tau, t) < 0$, then the demand in period t cannot be satisfied by the production in period τ; therefore, we assign zero values to all assignment variables $x(\tau, t)$ with negative costs.

Listing 1.3: **MIPshell** model for lot-sizing with backlogging

```
#define d(t) m_ipDemand[t]
#define u(t) m_ipCapacity[t]
#define f(t) m_ipFxCost[t]
#define c(tau,t) m_ipCost[tau*T+t]

int Cbklotsize::model()
{
  int t, tau, T=m_iT, Tp=m_iTp;
  VAR_VECTOR x("x",REAL_GE,Tp,T), y("y",BIN,Tp);

  minimize(
      sum(tau in [0,Tp), t in [0,T): c(tau,t) >= 0) c(tau,t)*x(tau,t) +
      sum(tau in [0,Tp)) f(tau)*y(tau)
```

```
        );

    forall(t in [0,T))
        sum(tau in [0,Tp)) x(tau,t) == d(t);

    forall(tau in [0,Tp))
        sum(t in [0,T)) x(tau,t) <= u(tau)*y(tau);

    forall(tau in [0,Tp), t in [0,T): c(tau,t) < 0)
        x(tau,t) == 0;

    optimize();
    printSol(x,y);
    return 0;
} // end of Cbklotsize::model
```

1.3.2 Aggregate Planning Example

A plant producing vehicles is to elaborate an aggregate plan for a month divided into four weeks with demand forecast as 1000 units in each. The capacity available is 4400 units per month:

- weeks 1 and 4: 600 in regular time and 200 in overtime;
- weeks 2 and 3: 1000 in regular time and 400 in overtime.

At the beginning of the month there are 100 cars in inventory. Therefore an excess capacity is 500 ($100 + 2 \cdot 1400 + 2 \cdot 800 - 4 \cdot 1000$). However, it is highly desired to have 200 units in inventory at the end of the month. The inventory cost (holding cost + insurance + lost revenue) of one unit is \$50 for each week. The average workforce cost per car is \$1500 in regular time, and \$2000 in overtime. Overtime is, of course, more expensive to start with; therefore there is an additional fixed cost of \$10000 to organize overtime production in each week.

Input data for model (1.6) is presented in Table 1.1. To solve this example problem with our program described in the previous section, we have to transfer data from Table 1.1 into a text file which contents is given in Listing 1.4. Two numbers in the first line are, respectively, the number of demand periods and the number of production periods. the next three lines describe demands, capacities, and fixed costs. In the rest nonempty lines describe the costs $c(\tau, t)$. Let us remember that we agreed that a negative cost $c(\tau, t)$, say $c(\tau, t) - 1$, indicates that the demand in period t cannot be satisfied by production in period τ.

Listing 1.4: Input file for aggregate planning example

Table 1.1: Aggregate planning data

Production periods		Sales periods 1	2	3	4	Stock	Fixed costs	Total capacity
Stock		0	50	100	150	200	0	100
1	Reg. time	1500	1550	1600	1650	1700	0	600
	Overtime	2000	2050	2100	2150	2200	10000	200
2	Reg. time	2000	1500	1550	1600	1650	0	1000
	Overtime	2500	2000	2050	2100	2150	10000	400
3	Reg. time	—	2000	1500	1550	1600	0	1000
	Overtime	—	2500	2000	2050	2100	10000	400
4	Reg. time	—	—	2000	1500	1550	0	600
	Overtime	—	—	2500	2000	2050	10000	200
Demand		1000	1000	1000	1000	200	4500	

```
5 9
1000 1000 1000 1000 200
100  600 200  1000 400  1000 400  600 200
0  0 10000  0 10000  0 10000  0 10000

   0   50  100  150  200
1500 1550 1600 1650 1700
2000 2050 2100 2150 2200
2000 1500 1550 1600 1500
2500 2000 2050 2100 2150
  -1 2000 1500 1550 1600
  -1 2500 2000 2050 2100
  -1   -1 2000 1500 1550
  -1   -1 2500 2000 2050
```

To solve our instance, we first enter the directory `tests` where **`aggrProdPlan.txt`** is stored, and then enter the command

```
../bin/bklotsize aggrProdPlan
```

to get in `tests` the text file named **`aggrProdPlan.sol`** with the content as that in Listig 1.5.

Listing 1.5: Solution for aggregate planning example

```
+--------+----------+------------+
| Prod.  | produces | for demand |
| period | (units)  |   period   |
+--------+----------+------------+
|     0  |      100 |        1   |
+--------+----------+------------+
|     1  |      600 |        1   |
+--------+----------+------------+
|     2  |      200 |        1   |
+--------+----------+------------+
|     3  |      100 |        1   |
|        |      700 |        2   |
|        |      200 |        5   |
+--------+----------+------------+
|     4  |      300 |        2   |
+--------+----------+------------+
|     5  |     1000 |        3   |
+--------+----------+------------+
|     6  |      200 |        4   |
+--------+----------+------------+
|     7  |      600 |        4   |
+--------+----------+------------+
|     8  |      200 |        4   |
+--------+----------+------------+
Production cost - 6660000
Fixed cost      - 40000
```

1.4 Multiproduct Lot-Sizing

Production process

- transform raw materials into end products;
- usually there a series of transformation steps;
- each step consuming and producing intermediate products;
- raw materials, intermediate and end products may be inventoried.

We need to determine an aggregate production plan for n different products on a number of machines of m types for a planning horizon that extends over T periods.

Inputs parameters:

- l_t: duration (length) of period t;
- m_{it}: number of machines of type i available in period t;
- f_{it}: fixed cost of producing on machine of type i in period t;
- $T_i^{\min}, T_i^{\max}$: minimum and maximum working time of one machine of type i;
- c_{jt}: per unit production cost of product j in period t;
- h_{jt}: inventory holding cost per unit of product j in period t;
- d_{jt}: demand for product j in period t;
- ρ_{jk}: number of units of product j used for producing one unit of product k;
- τ_{ij}: per unit production time of product j on machine of type i;
- s_j^i: initial stock of product j at the beginning of the planning horizon.
- s_j^f: final stock of product j at the end of the planning horizon.

We need to determine the production levels for each product and each period so that to minimize the total production and inventory expenses over planing horizon.

We introduce the following variables

x_{jt}: amount of product j, produced in period t;

s_{jt}: amount of product j in stock at the end of period t;

y_{it}: number of machunes of type i working in period t.

Now we formulate the problem as follows:

$$\sum_{t=1}^{T}\sum_{j=1}^{n}(h_{jt}\,s_{jt}+c_{jt}\,x_{jt})+\sum_{t=1}^{T}\sum_{i=1}^{m}f_{it}\,y_{it}\to\min, \tag{1.7a}$$

$$s_j^i+x_{j1}=d_{j1}+s_{j1}+\sum_{k=1}^{n}\rho_{jk}\,x_{k,1},\quad j=1,\dots,n, \tag{1.7b}$$

$$s_{j,t-1}+x_{jt}=d_{jt}+s_{jt}+\sum_{k=1}^{n}\rho_{jk}\,x_{k,t},\quad j=1,\dots,n;\ t=2,\dots,T, \tag{1.7c}$$

$$\sum_{j=1}^{n}\tau_{ij}\,x_{jt}\le l_t y_{it},\quad i=1,\dots,m;\ t=1,\dots,T, \tag{1.7d}$$

$$s_{jT}=s_j^f,\quad j=1,\dots,n, \tag{1.7e}$$

$$0\le s_{jt}\le u_j,\ x_{jt}\ge 0,\quad j=1,\dots,n;\ t=1,\dots,T, \tag{1.7f}$$

$$0\le y_{it}\le m_{it},\ y_{it}\in\mathbb{Z},\quad i=1,\dots,m;\ t=1,\dots,T. \tag{1.7g}$$

The objective (1.7a) prescribes to minimize the total production and inventory expenses. The balance equations (1.7b) and (1.7c) join two adjacent periods: product in stock in period $t-1$ plus that produced in period t equals the demand in period t plus the amount of product used when producing other products, and plus the amount in stock in period t. The inequalities (1.7d) require that the working time of machines of any type be withing given limits; besides, if machine i does not work in period t ($y_{it} = 0$), then no product is produced by this machine (all $x_{jt} = 0$).

1.4.1 MIPSHEL implementation

We the multiproduct lot-size problem as a **C++** class named **Cmultlotsize**. This class has the following members to store problem instances:

- *m_iT*: number of periods in time horizon;
- *m_iProdNum*: number of products;
- *m_iMachTypeNum*: number of machines;
- *m_ipMachNum*: integer array of size *m_iMachTypeNum*× *m_iT*, where *m_dpFixedCost*[t×*m_iProdNum*+i] is number of machibes of type i available in period t;
- *m_dpFixedCost*: real array of size *m_iMachTypeNum*× *m_iT*, where *m_dpFixedCost*[t×*m_iProdNum*+i] is fixed cost of starting production on machine i in period t;
- *m_ipStockCapm_ipInitStock*,*m_ipFinalStock*: arrays of size *m_iProdNum*, where
  - *m_ipStockCap*[j] is stock capacity for product j;
  - *m_ipInitStock*[j] and *m_ipFinalStock*[j] are initial and final stocks of product j;
- *m_dpCost*: real array of size *m_iProdNum*× *m_iT*, where *m_HoldingCost*[t×*m_iProdNum*+j] is inventory holding cost of product j in period t;
- *m_dpHoldingCost*: real array of size *m_iProdNum*× *m_iT*, where *m_dpCost*[t×*m_iProdNum*+j] is cost of producing one unit of product j in period t;
- *m_ipDemand*: integer array of size *m_iProdNum*× *m_iT*, where *m_ipDemand*[t×*m_iProdNum*+j] is demand for product j in period t;
- *m_ipDur*: integer array of size *m_iT*, where *m_ipDur*[t] is duration of period t;

- *m_dpTau*: real array of size *m_iMachTypeNum*×*m_iProdNum*, where *m_dpTau*(i×*m_iProdNum*+j) is per unit production time of product j on machine i;

- *m_dpRho*: real array of size *m_iProdNum*×*m_iProdNum*, where *m_dpRho*[j×*m_iProdNum*+k) is number of units of product j used for producing one unit of product k.

A straightforward implementation of MIP (1.7) is given in Listing 1.6.

Listing 1.6: **MIPshell** model for multiproduct lot-size problem

```
#define si(j) m_ipInitStock[j]
#define sf(j) m_ipFinalStock[j]
#define u(j) m_ipStockCap[j]
#define m(i,t) m_ipMachNum(t*m+i)
#define f(i,t) m_ipFixedCost(t*m+i)
#define c(j,t) m_dpCost[t*n+j]
#define h(j,t) m_dpHoldingCost[t*n+j]
#define d(j,t) m_ipDemand(t*n+j)
#define l(t) m_ipDur[t]
#define tau(i,j) m_dpTau[i*n+j]
#define rho(j,k) m_dpRho[j*n+k]

int Cmultlotsize::model()
{
  int i,j,t, m,n,T;
  m=m_iMachTypeNum; n=m_iProdNum; T=m_iT;
  VAR_VECTOR s("s",REAL_GE,n,T), x("x",REAL_GE,n,T), y("y",INT,m,T);

  minimize(
        sum(j in [0,n), t in [0,n)) h(j,t)*s(j,t)
    + sum(j in [0,n), t in [0,n)) c(j,t)*x(j,t)
    + sum(i in [0,m), t in [0,T)) f(i,t)*y(i,t)
  );

  forall(j in [0,n))
    si(j) + x(j,0) ==
    d(j,0) + s(j,0) + sum(k in [0,n)) rho(j,k)*x(k,0);

  forall(t in [1,T), j in [0,n))
    s(j,t-1) + x(j,t) ==
    d(j,t) + s(j,t) + sum(k in [0,n)) rho(j,k)*x(k,t);

  forall(i in [0,m), t in [0,T)) {
```

Table 1.2: Unit Production Times

Processes	Products						
	1	2	3	4	5	6	7
Grinding	0.5	0.7	—	—	0.3	0.2	0.5
Vertical drilling	0.1	0.2	—	0.3	—	0.6	—
Horizontal drilling	0.2	—	0.8	—	—	—	0.6
Boring	0.05	0.03	—	0.07	0.01	—	0.08
Planing	—	—	0.01	—	0.05	—	0.05
Packing	0.1	0.1	0.1	0.1	0.1	0.1	0.15

```
    sum(j in [0,n)) tau(i,j)*x(j,t) <= l(t)*y(i.t);
    y(i,t) <= m(i,t);
  }
  forall(j in [0,n)) {
    forall(t in [0,T))
      s(j,t) <= u(j);
    s(j,T-1) == sf(j);
  }

  optimize();
  printsol();
  return 0;
} // end of Cmultlotsize::model
```

1.4.2 Example

An engineering factory makes seven products (numbered from 1 to 7) on the following machines: four grinders, two vertical drills, three horizontal drills, one borer, one planer, and two packing devices. Products 5 and 7 are complete sets: one unit of product 5 comprises one unit of product 1 and one units of product 3, and one unit of product 7 comprises one unit of product 1 and two unit of product 6. Unit production times (in hours) required on each process are given in Table 1.2. A dash indicates that a product does not require a process. All required processes on each product unit can be applied in any order. So, no sequencing problem need to be considered.

The planning horizon consists of six months from January to June. The demands for products are given in Table 1.3. The factory works a 5 day week with two shifts of 8 hours each day. The number of working days in each month is also presented in Table 1.3.

Table 1.3: Demands for Products

Months	Working days	Products						
		1	2	3	4	5	6	7
January	19	500	1000	300	300	800	200	100
February	19	600	700	200	0	400	300	150
March	22	300	600	0	0	500	400	100
April	22	200	300	400	500	200	0	100
May	21	0	100	500	100	1000	300	0
June	21	500	500	100	300	1100	500	60

Table 1.4: Machines Available and their Setup Costs

Machine	Setup cost	Available machines					
		Jan	Feb	Mar	Apr	May	Jun
Grinder	500	6	5	5	6	5	5
Ver. driller	450	3	3	2	2	3	3
Hor. driller	450	5	3	5	5	5	5
Borer	550	2	2	1	2	1	2
Planer	1000	1	2	2	1	2	2
Packing dev.	400	2	2	2	2	2	2

During the planning horizon some machines will be down for maintenance: The setup cost of a machine mainly is the fixed salary of an operator. As the planer needs two operators, its setup cost is two times bigger than setup cost of the other machines. Machine setup costs, and the numbers of machines of each type available in each month are given in Table 1.4.

Unit production costs depend on the production period. It is 10 m %ore expensive to produce in the cold months (January, February, and March) than in the warm ones (April, May, and June). These costs are given in Table 1.5.

It is possible to store up to 100 units of each product at a cost of $1 per unit per month. There are not stock at the beginning of the horizon but it is desired to have a stock of 50 units of each product at the end of June.

When and what should the factory produce to maximize the total profit?

To solve this example problem with our program described in the previous section, we prepared an input text file **`test1.txt`** located in the folder

`$MIPDIR/examples/mipshell/multlotsize/tests`

The contents of **`test1.txt`** is given in Listing 1.7.

Table 1.5: Production Costs

Periods	Products						
	1	2	3	4	5	6	7
January, February, March	17	19	16.5	8	9	16	25
April, May, June	16.3	17.1	14.9	7.2	8.1	14.4	22.5

Listing 1.7: Input file for multiproduct aggregate planning example

```
7 6 6

100 0 50  100 0 50  100 0 50  100 0 50
100 0 50  100 0 50  100 0 50

1  304
500 17 1  1000 19 1  300 16.5 1  300 8 1
  800 9 1  200 16 1  100 25 1
500 3  450 2  450 3  550 1  1000 1  400 2

2  304
600 17 1   700 19 1  200 16.5 1    0 8 1
  400 9 1  300 16 1  150 25 1
500 4  450 2  450 1  550 1  1000 1  400 2

3  352
300 17 1   600 19 1    0 16.5 1    0 8 1
  500 9 1  400 16 1  100 25 1
500 4  450 2  450 3  550 0  1000 1  400 2

4  352
200 16.3 1  300 17.1 1  400 14.9 1  500 7.2 1
  200 8.1 1  0 14.4 1  100 22.5 1
500 4  450 1  450 3  550 1  1000 1  400 2

5  336
  0 16.3 1  100 17.1 1  500 14.9 1  100 7.2 1
  1000 8.1 1  300 14.4 1  0 22.5 1
500 3  450 1  450 3  550 1  1000 1  400 2

6  336
500 16.3 1  500 17.1 1  100 14.9 1  300 7.2 1
  1100 8.1 1  500 14.4 1  60 22.5 1
500 4  450 2  450 2  550 1  1000 0  400 2

0.5   0.7   0.0   0.0   0.3   0.2  0.5
0.1   0.2   0.0   0.3   0.0   0.6  0.0
```

```
0.2   0.0   0.8   0.0   0.0   0.0   0.6
0.05  0.03  0.0   0.07  0.01  0.0   0.08
0.0   0.0   0.01  0.0   0.05  0.0   0.05
0.1   0.1   0.1   0.1   0.1   0.1   0.15

0  0  0  0  1  0  1
0  0  0  0  0  0  0
0  0  0  0  1  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0  1
0  0  0  0  0  0  0
```

In **`test1.txt`** input data is structured as follows:

- Line 1: n, m, T;
- Line 2: stock capacities $u_1, \ldots, u_n$;
- Line 3: initial stock $s0_1, \ldots, s0_n$;
- Line 4: final stock $sf_1, \ldots, sf_n$;
- Each period is described by 3 lines, $3T$ lines in total:
  - Line $4 + 3(t-1)$: t, working time (in hours);
  - Line $5 + 3(t-1)$: $d_{1t}, c_{1t}, h_{1t}, \ldots, d_{nt}, c_{nt}, h_{nt}$;
  - Line $6 + 3(t-1)$: $f_{1t}, m_{1t}, \ldots, f_{mt}, m_{mt}$;
- Line $i = 4 + 3T, \ldots, 4 + 3T + m$: $\tau_{i1}, \ldots, \tau_{in}$;
- Line $j = 4 + 3T + m, \ldots, 4 + 3T + m + n$: $rho\_j1, \ldots, \rho_{jn}$.

If we open a console window, enter the directory

```
$MIPDIR/examples/mipshell/multlotsize/tests
```

and then enter the command

```
multlotsize test1
```

we will get the solution in the text file **`test1.sol`**. In a concise form this solution is presented in tables 1.6 and 1.7.

Table 1.6: Machines Used

Machine	Used machines					
	Jan	Feb	Mar	Apr	May	Jun
Grinder	6	5	3	2	3	5
Ver. driller	3	2	2	1	1	3
Hor. driller	5	3	2	2	5	4
Borer	1	1	1	1	1	1
Planer	1	1	1	1	1	1
Packing dev.	2	2	1	1	2	2

Table 1.7: Products Produced and Stocked

Month		Products						
		1	2	3	4	5	6	7
Jan	Produced	1400	1000	1100	300	800	300	100
	Stocked	0	0	0	0	0	0	0
Feb	Produced	1245	700	600	0	400	545	245
	Stocked	0	0	0	0	0	0	95
Mar	Produced	825	600	500	0	500	405	5
	Stocked	0	0	0	0	0	0	0
Apr	Produced	529	340	600	500	200	135	129
	Stocked	0	40	0	0	0	0	29
May	Produced	1104	96	1600	100	1100	294	0
	Stocked	4	36	0	0	100	0	29
Jun	Produced	1677	514	1200	350	1050	631	81
	Stocked	50	50	50	50	50	50	50

Chapter 2

Operations Scheduling

In poorly scheduled job shops, it is not at all uncommon for jobs to wait for 95 percent of their total production cycle. This results in a long workflow cycle. Add inventory time and receivables collection time to this and you get a long cash flow cycle. Thus, workflow equals cash flow, and workflow is driven by the schedule.

A schedule is a timetable for performing activities, utilizing resources, or allocating facilities. In this chapter, we discuss modeling aspects of some short-term scheduling of jobs and processes.

2.1 Generalized Assignment Problem

The generalized assignment problem can be viewed as the following problem of scheduling parallel machines to minimize total cost of processing a number of jobs. Each of m independent jobs is to be processed by exactly one of n unrelated parallel machines; job i takes p_{ij} time units and costs c_{ij} when processed by machine j, $i = 1, \ldots, m$, $j = 1, \ldots, n$. The *workload* (the total working time) of a machine j is at most l_j. A schedule is an $m \times n$-matrix $\{x_{ij}\}$, $x_{ij} = 1$ means that job i is assigned to machine j. The *Generalized Assignment Problem* (GAP) is to find a schedule of minimum cost that obeys all the above requirements. It is modeled as the following IP:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij}\, x_{ij} \to \min \tag{2.1a}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, \ldots, m, \tag{2.1b}$$

$$\sum_{i=1}^{m} p_{ij} x_{ij} \le l_j, \quad j = 1, \ldots, n, \tag{2.1c}$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, \ldots, m;\ j = 1, \ldots, n. \tag{2.1d}$$

Constraints (2.1b) enforce that each job is assigned exactly to one machine. Constraints (2.1c) insure that the total running time of each machine is at most its maximum workload.

2.1.1 MIPshell implementation

A **MIPshell** implementation of GAP is given as a **C++** class named **Cgenassign**. This class describes problem instances via two **MIPshell** arrays:

- *p*: two-dimensional integer array, where *p*(i,j) is processing time of job i on machine j;
- *c*: two-dimensional integer array, where *c*(i,j) is cost of processing job i on machine j.

A **MIPshell** model of IP (2.1) is given in Listing 2.1.

Listing 2.1: **MIPshell** model for generalized assignment problem

```
int Cgenassign::model()
{
  int i, j, m=c.GetSize(0), n=c.GetSize(1);
  VAR_VECTOR x("x",BIN,m,n);

  minimize(sum(i in [0,m), j in [0,n)) c(i,j)*x(i,j));
  forall(i in [0,m))
    sum(j in [0,n)) x(i,j) == 1;
  forall(j in [0,n))
    sum(i in [0,m)) p(i,j)*x(i,j) <= l(j);

  optimize();
  printsol();
  return 0;
}
```

We refer the user to the

$MIPDIR/examples/mipshell/genassign

directory for a complete application that reads data from a file and then solve the instance by calling the *genassign* procedure.

2.1.2 Illustrative example

Suppose that a scheduler has ten jobs that can be performed on any of five machines. Input data to this example is given in Table 2.1. To solve this example by

Table 2.1: GAP instance

J\M	1	2	3	4	5
1	110	16	25	78	59
2	65	69	54	28	71
3	19	93	45	45	9
4	89	31	72	83	20
5	62	17	77	18	39
6	37	115	87	59	97
7	89	102	98	74	61
8	78	96	87	55	77
9	74	27	99	91	5
10	88	97	99	99	51

a) Costs

J\M	1	2	3	4	5
1	5	99	79	34	41
2	46	47	56	74	40
3	97	9	57	58	95
4	28	70	44	28	91
5	56	99	29	87	73
6	80	1	13	48	15
7	28	4	3	27	51
8	25	18	17	56	41
9	32	92	13	26	96
10	31	17	2	12	55

b) Processing times

Machine	1	2	3	4	5
Working time	91	87	109	88	64

c) Machine working times

our program, we have to prepare a text file like one presented in Listing 2.2. In fact, Listing 2.2 presents the contents of the text file **`test1.txt`** from the directory

$MIPDIR/examples/mipshell/genassign/tests

Listing 2.2: Input file for GAP example

```
dim(5): [91,87,109,88,64]

dim(10,5):
[[110,16,25,78,59],
 [65,69,54,28,71],
 [19,93,45,45,9],
 [89,31,72,83,20],
 [62,17,77,18,39],
 [37,115,87,59,97],
 [89,102,98,74,61],
 [78,96,87,55,77],
 [74,27,99,91,5],
 [88,97,99,99,51]]
```

```
dim(10,5):
[[5,99,79,34,41],
[46,47,56,74,40],
[97, 9,57,58,95],
[28,70,44,28,91],
[56,99,29,87,73],
[80, 1,13,48,15],
[28, 4, 3,27,51],
[25,18,17,56,41],
[32,92,13,26,96],
[31,17, 2,12,55]]
```

2.2 Flow Shop Scheduling

Consider a factory that produces n sorts of widgets. Each widget must first be processed in sequence by machines $1, 2, \ldots, m$. Processing time of widget i on machine k is p_{ik}. The factory has orders for batches of widgets, each batch comprises one copy of each widget. The problem is to process all n widgets in a batch as quickly as possible.

We take as variables:

s_{ik}: start time of processing widget i on machine k;

$\delta_{ijk} = 1$ if on machine k widget i is processed before widget j, and $\delta_{ijk} = 0$ otherwise;

τ: makespan, i.e., completion time of the last processed widget on the last machine m.

With these variables the formulation is

$$\tau \to \min \tag{2.2a}$$

$$s_{i,k-1} + p_{i,k-1} \le s_{ik}, \quad i = 1, \ldots, n;\ k = 2, \ldots, m, \tag{2.2b}$$

$$s_{ik} + p_{ik} \le s_{jk} + T\delta_{jik}, \quad i = 1 \ldots, n;$$
$$j = i + 1, \ldots, n;\ k = 1, \ldots, m, \tag{2.2c}$$

$$\delta_{ijk} + \delta_{jik} = 1, \quad i = 1 \ldots, n; j = i + 1, \ldots, n;$$
$$k = 1, \ldots, m, \tag{2.2d}$$

$$s_{im} + p_{im} \le \tau, \quad i = 1, \ldots, n, \tag{2.2e}$$

$$\tau \in \mathbf{R}_+, \tag{2.2f}$$

$$s_{ik} \in \mathbf{R}_+, \quad i = 1, \ldots, n;\ k = 1, \ldots, m, \tag{2.2g}$$

$$\delta_{ijk} \in \{0, 1\}, \quad i, j = 1, \ldots, n;\ k = 1, \ldots, m. \tag{2.2h}$$

Here T is a big number greater than the makespan, for example, we can take $T = \sum_{k=1}^{m} \sum_{i=1}^{n} p_{ik}$.

A **MIPshell** procedure that corresponds to the MIP (2.2) is presented in Figure 2.3. The only parameter of the procedure is a two-dimensional integer array p of processing times. We define the number of machines m and the number of widgets n by the size of p. There is a minor difference between two models.To strengthen the formulation (2.2), we have added constraints $\delta_{i,j,m-1} = \delta_{ijm}$ to the MIPshell model. These constraints are based on the well known fact that there exists an optimal schedule according to which the widgets are processed in the same order on two last machines.

Listing 2.3: **MIPshell** procedure for flow shop scheduling

```
int Cflowshop::model()
{
  int i,j,k, m,n, T=0;
  VAR_VECTOR start("start",INT_GE,n,m);
  VAR_VECTOR delta("delta",BIN,n,n,m);
  VAR tau("tau",INT_GE);
  m=p.GetSize(0); n=p.GetSize(1);
  T=p.Sum();

  minimize(tau);
  forall(k in [1,m), i in [0,n))
    start(i,k-1) + p(i,k-1) <= start(i,k);
  forall(k in [0,m), i in [0,n), j in [0,n): j != i) {
    start(i,k) + p(i,k) <= start(j,k) + T*delta(j,i,k);
    setpriority(delta(i,j,k),m-k);
  }
  forall(k in [0,m), i in [0,n), j in [i+1,n))
    delta(i,j,k) + delta(j,i,k) == 1;
  k=m-1;
  forall(i in [0,n))
    start(i,k) + p(i,k) <= tau;
  forall(i in [0,n), j in [0,n): i != j)
    delta(i,j,k-1) == delta(i,j,k);
  optimize();
  printsol();
  return 0;
}
```

Any problem instance is given by a two dimensional array of integers p of size $n \times m$, where m and n are defined as follows:

```
n=p.GetSize(0); m=p.GetSize(1);
```

Each row of p gives processing times of a widget on all m machines. Array p is a member of the **Cflowshop** class, and the array is initialized in a **Cflowshop** constructor from an input text file which name (without the ".txt" extension) is passed as the only program parameter.

2.3 A general scheduling problem

In a scheduling problem we have to fulfill a set of jobs on a number of processors using other resources under certain constraints such as restrictions on the job completion times, priorities between jobs (one job cannot start until another one is finished), and etc. The goal is to optimize some criterion, e.g, to minimize the total processing time, which is the completion time of the last job (assuming that the first task starts at time 0); or to maximize the number of processed jobs.

Let us also note that processors are special resources such as machines or workers. We classify a unit of some resource as a processor if job processing times depend on which units of the resource are used when processing that jobs. For instance, when k machines are identical we can treat them as a resource with k units available. But when machines are of different productivity, we can not treat them as a single resource.

Next we formulate a very general scheduling problem which subsumes as special cases a great deal of scheduling problems studied in the literature. We are given n jobs to be processed on m processors. Let $\mathcal{P}_i \subseteq \mathcal{P} \stackrel{\text{def}}{=} \{1, \ldots, m\}$ denote the subset of processors that can fulfill job $j \in \mathcal{J} \stackrel{\text{def}}{=} \{1, \ldots, n\}$. There are also q non-perishable resources, with R_k units of resource k available per unit of time.

Each job j is characterized by the following parameters:

- w_j: weight;
- l_j, u_j: release and due dates (the job must be processed during the time interval $[l_j, u_j]$);
- p_{ij}: processing time on processor $i = 1, \ldots, m$;
- r_{kj}: needed amount of resource $k = 1, \ldots, q$.

Precedence relations between jobs are given by an acyclical digraph $G = (\mathcal{J}, E)$ defined on the set $\mathcal{J}$ of jobs: for any arc $(j_1, j_2) \in \mathcal{J}$, job j_2 cannot start until job j_1 is finished.

In general not all jobs can be processed. For a given schedule, let $U_j = 0$ if job j is processed, and $U_j = 1$ otherwise. Then the problem is to find such a schedule for which the weighted number of not processed jobs, $\sum_{j=1}^{n} w_j U_j$, is minimum. Alternatively, we can say that our goal is to maximize the weighted sum of processed jobs, which is $\sum_{j=1}^{n} w_j (1 - U_j)$.

2.3.1 Event-driven formulation

We represent a schedule by the following two sets of decision variables:

- s_j: start time of job $j = 1, \ldots, n$;
- $x_{ij} = 1$ if job j is accomplished by processor i, otherwise, $x_{ij} = 0$, $j \in \mathcal{J}_i$ and $i = 1, \ldots, m$.

In our formulation we also use four sets of auxiliary variables:

- $y_j = 1$ if job j is processed, otherwise, $y_j = 0$, $j = 1, \ldots, n$;
- p_j: processing time of job $j = 1, \ldots, n$;
- $\delta_{j_1,j_2} = 1$ if job j_1 starts not later than job j_2 ($s_{j_1} \le s_{j_2}$), otherwise, $\delta_{j_1,j_2} = 0$, $j_1, j_2 = 1, \ldots, n$ and $j_1 \ne j_2$;
- $\gamma_{j_1,j_2} = 1$ if job j_1 is active (starts or being processed) when job j_2 starts, otherwise, $\gamma_{j_1,j_2} = 0$, $j_1, j_2 = 1, \ldots, n$ and $j_1 \ne j_2$.

In the above variables our scheduling problem is formulated as follows:

$$\sum_{j=1}^{n} w_j y_j \to \max \tag{2.3a}$$

$$y_j = \sum_{i \in \mathcal{P}_j} x_{ij}\,, \quad j = 1, \ldots, n, \tag{2.3b}$$

$$p_j = \sum_{i \in \mathcal{P}_j} p_{ij} x_{ij}\,, \quad j = 1, \ldots, n, \tag{2.3c}$$

$$s_{j_2} - s_{j_1} \le T\delta_{j_1,j_2}, \quad j_1 \ne j_2,\ j_1, j_2 = 1, \ldots, n, \tag{2.3d}$$

$$\delta_{j_1,j_2} + \delta_{j_2,j_1} = 1, \quad j_2 = j_1 + 1, \ldots, n,\ j_1 = 1, \ldots, n-1, \tag{2.3e}$$

$$s_{j_2} - s_{j_1} \le p_{j_1} + T \cdot (3 - x_{i,j_1} - x_{i,j_2} - \delta_{j_1,j_2}),$$
$$j_2 = j_1 + 1, \ldots, n,\ j_1 = 1, \ldots, n-1,\ i = 1, \ldots, m, \tag{2.3f}$$

$$\delta_{j_1,j_2} = 1, \quad (j_1, j_2) \in E, \tag{2.3g}$$

$$s_{j_2} - s_{j_1} \ge p_{j_1}, \quad (j_1, j_2) \in E, \tag{2.3h}$$

$$y_{j_1} - y_{j_2} \ge 0, \quad (j_1, j_2) \in E, \tag{2.3i}$$

$$\gamma_{j_1,j_2} \le \delta_{j_1,j_2}, \quad j_1 \ne j_2,\ j_1, j_2 = 1, \ldots, n, \tag{2.3j}$$

$$1 - T + T \cdot \gamma_{j_1,j_2} \le s_{j_1} + p_{j_1} - s_{j_2} \le T \cdot (\gamma_{j_1,j_2} + \delta_{j_2,j_1}),$$
$$j_1 \ne j_2,\ j_1, j_2 = 1, \ldots, n, \tag{2.3k}$$

$$r_{k,j_1} + \sum_{j_2 \in \mathcal{J} \setminus \{j_1\}} r_{k,j_2}(\gamma_{j_1,j_2} + \gamma_{j_2,j_1}) \le R_k,$$
$$j_1 = 1, \ldots, n,\ k = 1, \ldots, q, \tag{2.3l}$$

$$s_j \geq l_j y_j, \quad j = 1, \ldots, n, \tag{2.3m}$$
$$s_j + p_j \leq u_j y_j, \quad j = 1, \ldots, n, \tag{2.3n}$$
$$x_{ij} \in \{0, 1\}, \quad j = 1, \ldots, n, \; i = 1, \ldots, m, \tag{2.3o}$$
$$\delta_{j_1,j_2}, \; \gamma_{j_1,j_2} \in \{0, 1\}, \quad j_1 \neq j_2, \; j_1, j_2 = 1, \ldots, n. \tag{2.3p}$$

In this formulation, T denotes an upper bound on the completion time of any job. For example, T may be the minimum of two next bounds:

$$T_1 \stackrel{\text{def}}{=} \max_{1 \leq j \leq n} \left(d_j + \max_{i \in \mathcal{P}_i} t_{ij} \right),$$
$$T_2 \stackrel{\text{def}}{=} \sum_{j=1}^{n} \max_{i \in \mathcal{P}_i} t_{ij}.$$

For the schedule defined by the values of variables (s_j, x_{ij}), the equalities (2.3b) are to determine accomplished jobs: $y_j = 1$ if job j is accomplished by some processor. Therefore, the objective (2.3a) is to maximize the weighted number of accomplished jobs.

The equalities (2.3b) and (2.3c) determine the processing times, τ_j, of all jobs j. Let us note that processing times of all non-processed jobs are zeroes.

The constraints (2.3d), (2.3e) and (2.3f) together forbid any two jobs be simultaneously processed by the same processor. Let us note that the inequality (2.3f), written for tasks j_1, j_2, and processor i, is a real restriction only if $x_{i,j_1} = x_{i,j_2} = \delta_{j_1,j_2}$, i.e., both tasks j_1 and j_2 are assign to processor i, and j_1 precedes task j_2.

Two constraints, (2.3g) and (2.3h), express the precedence relations. The inequalities (2.3i) convey the requirement that, if a job is not accomplished, then all its successors are not accomplished as well.

The inequalities (2.3j), (2.3k), and (2.3l) express the restrictions on the resources.

The inequalities (2.3j) and (2.3k) allow each variable γ_{j_1,j_2} to take value 1 only if job j_1 is active when job j_2 starts. The restrictions on resources are given by the inequalities (2.3l): at the moment when any job starts, the total amount of every resource used by the active jobs must not exceed the limit on this resource.

The restrictions on release and due dates are given by the inequalities (2.3m) and (2.3n).

2.3.2 Time-index formulation

A time-index formulation is based on time-discretization, i.e., the planning horizon is divided into periods, and period t starts at time t and ends at time $t + 1$. Let $L = \min_{1 \leq j \leq n} l_j$ and $U = \max_{1 \leq j \leq n} u_j$. We consider the following time-index formulation[1]:

[1] M.E. Dyer, L.A. Wolsey. Formulating the single-machine sequencing problem with release dates as a mixed integer program. *Discrete Appl. Math.* (1990) **26** 255–270.

$$\sum_{j=1}^{n} \sum_{i \in \mathcal{P}_j} \sum_{t=l_j}^{u_j - p_{ij}} c_{ijt} x_{ijt} \to \max \tag{2.4a}$$

$$\sum_{i \in \mathcal{P}_j}^{m} \sum_{t=l_j}^{u_j - p_{ij}} x_{ijt} \leq 1, \quad j = 1, \ldots, n, \tag{2.4b}$$

$$\sum_{\substack{1 \leq j \leq n: \\ l_j \leq t \leq u_j - p_{ij}}} \sum_{\tau = \max\{t - p_{ij}, l_j\}}^{\min\{t, u_j - p_{ij}\}} x_{ij\tau} \leq 1, \quad t = L, \ldots, U; \; i = 1, \ldots, m, \tag{2.4c}$$

$$\sum_{i=1}^{m} \sum_{\substack{1 \leq j \leq n: \\ l_j \leq t \leq u_j - p_{ij}}} \sum_{\tau = \max\{t - p_{ij}, l_j\}}^{\min\{t, u_j - p_{ij}\}} r_{kj} x_{ij\tau} \leq R_k, \quad t = L, \ldots, U; \; k = 1, \ldots, q, \tag{2.4d}$$

$$y_j = \sum_{i \in \mathcal{P}_j} \sum_{t=l_j}^{u_j - p_{ij}} x_{ijt}, \quad j = 1, \ldots, n, \tag{2.4e}$$

$$s_j = \sum_{i \in \mathcal{P}_j} \sum_{t=l_j}^{u_j - p_{ij}} t \cdot x_{ijt}, \quad j = 1, \ldots, n, \tag{2.4f}$$

$$p_j = \sum_{i \in \mathcal{P}_j} \sum_{t=l_j}^{u_j - p_{ij}} p_{ij} \cdot x_{ijt}, \quad j = 1, \ldots, n, \tag{2.4g}$$

$$y_{j_1} - y_{j_2} \geq 0, \quad (j_1, j_2) \in E, \tag{2.4h}$$

$$s_{j_2} - s_{j_1} \geq p_{j_1}, \quad (j_1, j_2) \in E, \tag{2.4i}$$

$$x_{ijt} \in \{0, 1\}, \quad i \in \mathcal{P}_j; \; t = l_j, \ldots, u_j - p_{ij}; \; j = 1, \ldots, n. \tag{2.4j}$$

Here a binary variable x_{ijt} takes value 1 if job j starts in period t on processor i; otherwise, $x_{ijt} = 0$. The assignment constraints (2.4b) require that each job can start at most once, and the capacity constraints (2.4c) state that any processor fulfills at most one job during any time period. If we define $c_{ijt} = w_j$, due to (2.4b), the objective (2.4a) is to maximize the weighted number of processed jobs.

To formulute precedence relations we need three families of *auxiliary* variables, which are uniquely defined by the *decision* variables x_{ijt}. For each job j, the equations (2.4e), (2.4f), and (2.4g) respectively determine the following values:

- $y_j = 1$ if job j is processed; otherwise, $y_j = 0$;
- s_j: start time of job j;
- p_j: processing time of job j.

In these new variables the precedence relations are expressed by the inequalities (2.4h) and (2.4i). For each pair of related jobs $(j_1, j_2) \in E$, (2.4h) requires that j_2 is not processed if j_1 is not processed, while (2.4i) requires that, if both jobs, j_1 and j_2, are processed, then j_1 must be finished when j_2 starts.

An important advantage of the time-index formulation is that it can be used to model many types of scheduling problems. For examples, if in constraint (2.4b) we replace the inequality sign by the equality, and define $c_{ijt} = -w_j(t + p_{ij})$, we get the problem of scheduling jobs with release and due dates on m machines to minimize weighted completion time, denoted as $m|r_j, d_j|\sum w_j C_j$. In addition, the LP-relaxation of the time-index formulation provides a strong bound: it dominates the bounds provided by other MIP formulations. This is because its LP-relaxation is nonpreemptive. That is, the relaxation is obtained by slicing jobs into pieces so that each piece is processed without interruption.

The main disadvantage of the time-index formulation is its size: even for one machine problems, there are $n + T$ constraints and may be up to nT variables. As a result, for instances with many jobs and large processing intervals $[r_j, d_j]$, the LP's will be very big in size, and their solution time will be large.

MIPshell-implementation

To implement MIP (2.4), we developed a **C++** class named **Ctimeindex**. Its definition and implementation you can find in the folder

$MIPDIR/examples/mipshell/timeindex.

This class has the following member to store problem instances:

- *m_iMachineNum*: number of machines;
- *m_iJobNum*: number of jobs;
- *m_ipRelease*: integer array of size *m_iJobNum*, where *m_ipRelease*[j] is release date of job j;
- *m_ipDead*: integer array of size *m_iJobNum*, where *m_ipDead*[j] is due date of job j;
- *m_dpWeight*: integer array of size *m_iJobNum*, where *m_dpWeight*[j] is weight of job j.
- *m_ipProcTime*: integer array of size *m_iMachineNum*× *m_iJobNum*, where *m_ipProcTime*[i + j× *m_iMachineNum*] is processing time of job j on machine i.

We advise you to use macros to keep your **MIPshell**-implementations close to your MIP-formulations. In this case we define the following macroes:

```
#define r(j) m_ipRelease[j]
#define d(j) m_ipDead[j]
#define w(j) m_dpWeight[j]
#define p(i,j) m_ipProcTime[i*n+j]
```

Problem instances are read from text files by the function ***readData*** which is defined in **Ctimeindex** as follows:

```
void readData(const char *fileName);
```

You can find a number of such files in the folder

$MIPDIR/examples/mipshell/timeindex/tests.

Now, let us turn to the implementation of formulation (2.4). To define decision variables, we first build a set of indexes I representing the set

$$\{(i,j,t):\ t=r_j,\dots,d_j-p_{ij};\ j=1,\dots,n;\ i=1,\dots,m\},$$

and then declare an array of variables

VAR_ARRAY x("x",**BIN**,I);

Now, with *x*(i,j,t) variables the **MIPshell**-formulation of MIP (2.4) is straightforward (see Listing 2.4).

Listing 2.4: **MIPshell** model for $m|r_j,d_j|\sum w_jU_j$

```
#define r(j) m_ipRelease[j]
#define d(j) m_ipDead[j]
#define p(i,j) m_ipProcTime[i+j*m]
#define w(j) m_dpWeight[j]

int Ctimeindex::model()
{
  int i,j,t,tau,
      m=m_iMachineNum,n=m_iJobNum,
      Rmin=std::numeric_limits<int>::max(), Dmax=0;
  INDEX_SET I;
  for (i=0; i < m; ++i) {
    for (j=0; j < n; ++j) {
      for (t=r(j); t <= d(j)-p(i,j); ++t)
        I.add(INDEX(i,j,t));
      if (Rmin > r(j)) Rmin=r(j);
      if (Dmax < d(j)) Dmax=d(j);
    }
  }
```

```
  VAR_ARRAY x("x",BIN,I);
// MIPshell model
  maximize(sum(i in [0,m), j in [0,n), t in [r(j),d(j)-p(i,j)]) w(j)*x(i,j,t));

  forall(j in [0,n))
    sum(i in [0,m), t in [r(j),d(j)-p(i,j)]) x(i,j,t) <= 1;

  forall(i in [0,m))
    forall(t in [Rmin,Dmax])
      sum(j in [0,n), tau in [t-p(i,j),t):
          tau >= r(j) && tau <= d(j)-p(i,j)) x(i,j,tau) <= 1;

  optimize();
  printSol(x);
  return 0;
} // end of Ctimeindex::model
```

When the MIP is solved (on return from *optimize*), we call *printSol*(x) to print the solution to a file which name is the name of an input file (without extention) concatenated with the extention ".sol".

An example

A firm provides copy services. Ten customers submitted their orders at the beginning of the week. Specific scheduling data are as follows:

Name	Processing time (days)	Due date (days)	Profit
1	3	5	3
2	4	6	4
3	2	7	2
4	6	7	5
5	1	2	2
6	1	4	1
7	2	6	2
8	3	7	3
9	4	7	3
10	3	3	4

There are only two copy machines. The firm must decide which orders to process to maximize its total profit.

This is an instance of $2|d_j|\sum w_jC_j$ which is a special case of $2|r_j,d_j|\sum w_jC_j$ when all release dates are zeroes. To solve this instance with our **`timeindex`** application, we prepared an input text file which contents is as follows.

```
2 10
0 5 3 3
0 6 4 4
0 7 2 2
0 7 5 6
0 2 2 1
0 4 1 1
0 6 2 2
0 7 3 3
0 7 3 4
0 3 4 3
```

2.4 Electricity Generation Planning

The *unit commitment problem* is to develop an hourly (or half-hourly) electricity production schedule spanning some period (a day or a week) so as to decide which generators will be producing and at what levels.

Let T be the number of periods. Period 1 follows period T. We know the *demand* d_t for each period t. In each period the capacity of the active generators must be at least q times of the demand (q is a level of reliability).

Let n be the number of generators, and let generator i have the following characteristics:

- l_i, u_i: minimum and maximum levels of production per period;
- r_i^1, r_i^2: ramping parameters (when a generator is on in two successive periods, its output cannot decrease by more than r_i^1, and increase by more than r_i^2);
- g_i: start-up cost (if a generator is off in some period, it costs g_i to start it in the next period);
- f_i, p_i: fixed and variable costs (if in some period a generator is producing at level v, it costs $f_i + p_i v$).

With the natural choice of variables

$x_{it} = 1$ if generator i produces in period t, and $x_{it} = 0$, otherwise;

$z_{it} = 1$ if generator i is switched on in period t, and $z_{it} = 0$, otherwise;

y_{it}: amount of electricity produced by generator i in period t,

we get the typical formulation:

$$\sum_{i=1}^{n}\sum_{t=1}^{T}(g_i z_{it} + f_i x_{it} + p_i y_{it}) \to \min \tag{2.5a}$$

$$\sum_{i=1}^{n} y_{it} = d_t, \quad t = 1, \ldots, T, \tag{2.5b}$$

$$\sum_{i=1}^{n} u_i x_{it} \geq q\, d_t, \quad t = 1, \ldots, T, \tag{2.5c}$$

$$l_i x_{it} \leq y_{it} \leq u_i x_{it}, \quad i = 1, \ldots, n;\ t = 1, \ldots, T, \tag{2.5d}$$

$$-r_i^1 \leq y_{it} - y_{i,((t-2+T) \bmod T)+1} \leq r_i^2, \quad i = 1, \ldots, n;\ t = 1, \ldots, T, \tag{2.5e}$$

$$x_{it} - x_{i,((t-2+T) \bmod T)+1} \leq z_{it}, \quad i = 1, \ldots, n;\ t = 1, \ldots, T, \tag{2.5f}$$

$$z_{it} \leq x_{it},, \quad i = 1, \ldots, n;\ t = 1, \ldots, T, \tag{2.5g}$$

$$x_{it}, z_{it} \in \{0, 1\}, \quad i = 1, \ldots, n;\ t = 1, \ldots, T, \tag{2.5h}$$

$$y_{it} \in \mathbb{R}_+, \quad i = 1, \ldots, n;\ t = 1, \ldots, T. \tag{2.5i}$$

To solve the unit commitment problem, we developed a **C++** class **Cunitcom** which stores a problem instance in the following members:

- *m_iGenNum*: number of generators;
- *m_iT*: number of periods;
- *m_dQ*: level of reliability;
- *m_ipDemand*: array of size *m_iT*, where *m_ipDemand*[t] is demand at period t;
- *m_ipLoCapacity*: array of size *m_iGenNum*, where *m_ipUpCapacity*[i] is minimum level of production per one period;
- *m_ipUpCapacity*: array of size *m_iGenNum*, where *m_ipUpCapacity*[i] is maximum level of production per one period;
- *m_ipRminus*, *m_ipRPlus*: arrays of size *m_iGenNum*, where *m_ipRminus*[i] and *m_ipRPlus*[i] are ramping parameters for generator i;
- *m_ipStartUpCost*: array of size *m_iGenNum*, where *m_ipStartUpCost*[i] is start-up cost of generator i;
- *m_ipFixedCost*, *m_ipUnitCost*: array of size *m_iGenNum*, where *m_ipFixed-Cost*[i] and *m_ipUnitCost*[i] are fixed and variable costs for generator i.

The default constructor of **Cunitcom**

Cunitcom(**const char** **name*)

just calls the function

void ***readData***(**const char** **fileName*)

to read a text file which name (without extension ".txt") is given by the parameter *fileName*. The data read from the file is stored in the class members described above. The function also allocates memory for the arrays mentioned earlier.

A straightforward MIPshell implementation of MIP model (2.5) is given in Listing 2.5.

Listing 2.5: **MIPshell** model for unit commitment problem

```
#define d(t) m_ipDemand[t]
#define u(i) m_ipUpCapacity[i]
#define l(i) m_ipLoCapacity[i]
#define r1(i) m_ipRminus[i]
#define r2(i) m_ipRPlus[i]
#define g(i) m_ipStartUpCost[i]
#define f(i) m_ipFixedCost[i]
#define p(i) m_ipUnitCost[i]

int Cunitcom::midel()
{
  double q=m_iQ;
  int i,t, n=m_iGenNum, T=m_iT;
  VAR_VECTOR x("x",BIN,n,T),
             y("y",REAL_GE,n,T),
             z("z",BIN,n,T);

  minimize(sum(i in [0,n), t in [0,T)) (g(i)*z(i,t) + f(i)*x(i,t) + p(i)*y(i,t)));
  forall(t in [0,T)) {
    sum(i in [0,n)) y(i,t) == d(t);
    sum(i in [0,n)) u(i)*x(i,t) >= q*d(t);
  }
  forall(i in [0,n), t in [0,T)) {
    l(i)*x(i,t) <= y(i,t);
    y(i,t) <= u(i)*x(i,t);
    -r1(i) <= y(i,t) - y(i,(t-1+T) % T) <= r2(i);
    x(i,t) - x(i,(t-1+T) % T) <= z(i,t);
    z(i,t) <= x(i,t);
  }
  optimize();
  printSol(y);
}
```

2.4.1 Example

To test the model, let us solve an instance of the unit commitment problem described in the text file **test1.txt** which contents is given in Listing 2.6.

Listing 2.6: Unit commitment example

```
5 12 1.2
50 60 50 100 80 70 90 60 50 120 110 70
 2 12 12 12 100  1 10
 2 12 12 12 100  1 10
 5 35 35 35 300  5  4
20 50 50 50 400 10  3
40 75 15 20 800 15  2
```

Three numbers in the first line show that the problem is to schedule $n = 5$ generators over $T = 12$ time periods of 2-hours each. The level of reliability is $q = 1.2$. The second line contains demands fo electricity in all 12 periods. Each of the other five lines — starting from the third — describes a generator in the following order: minimum and maximum levels of production, two ramping parameters, followed by start-up, fixed and variable costs. Let us note that all five generators are of different types, and the ramping constraints only apply to the fifth generator: $r_5^1 = 15$, $r_5^2 = 20$ (for other four generators we set $r_i^1 = r_i^2 = u_i$).

To solve our example, we enter the following commands

cd `$MIPDIR/examples/unitcom/tests/test1`; **unitcom**

The program writes the solution to the file **test1.sol**. In a concise form the solution is presented in Table 2.2.

2.5 Short-Term Scheduling in Chemical Industry

It is easier to start with an example[2] Two products, 1 and 2, are produced from three different feedstocks A,B, and C according to the following recipe:

1. *Heating*: Heat A for 1 h.
2. *Reaction 1*: Mix 50% feed B and 50% feed C and let them for 2 h to form intermediate BC.
3. *Reaction 2*: Mix 40% hot A and 60% intermediate BC and let them react for 2 h to form intermediate AB (60%) and product 1 (40%).

[2] E. Kondili, C.C. Pantelides, and R.W.H. Sargent. A general algorithm for short-term scheduling of batch operations — I. MILP formulation. *Computers chem. Engng.* **17** (1993) 211–227.

Table 2.2: Solution to the unit commitment instance

$t \setminus i$	1	2	3	4	5
0–2	0	0	5	0	45
2–4	0	0	5	0	55
4–6	0	0	5	0	45
6–8	0	2	33	0	65
8–10	0	0	5	0	75
10–12	0	0	5	0	65
12–14	0	0	20	0	70
14–16	0	0	5	0	55
16–18	0	0	5	0	45
18–20	0	0	5	50	65
20–22	0	0	5	30	75
22–24	0	0	10	0	60

4. *Reaction 3*: Mix 20% feed C and 80% intermediate AB and let them react for 1 h to form impure E.

5. *Separation*: Distill impure E to separate pure product 2 (90%, after 1 h) and pure intermediate AB (10% after 2 h). Discard the small amount of residue remaining at the end of the distillation. Recycle the intermediate AB.

The above process is represented by the *State-Task-Network* (STN) shown in Figure 2.1.

The following processing equipment and storage capacity are available.

Equipment :

Heater: Capacity 100 kg, suitable for task 1;

Reactor 1: Capacity 80 kg, suitable for tasks 2,3,4;

Reactor 2: Capacity 50 kg, suitable for tasks 2,3,4;

Still: Capacity 200 kg, suitable for task 5.

Storage capacity :

For feeds A,B,C: unlimited;

For hot A: 100 kg;

For intermediate AB: 200 kg;

For intermediate BC: 150 kg;

For intermediate E: 100 kg;

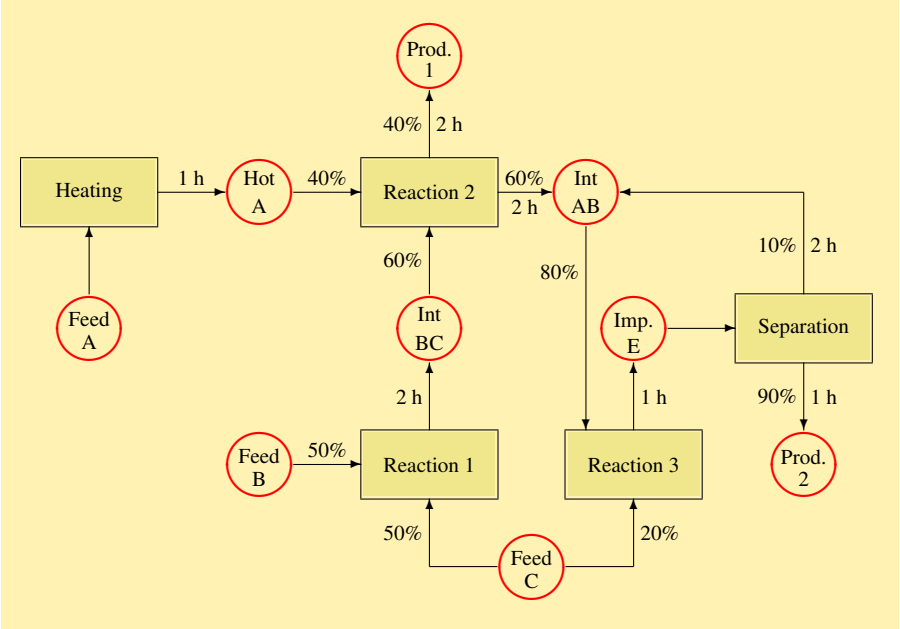

Figure 2.1: State-task network for example process

For products 1,2: unlimited.

A number of parameters are associated with the tasks and the states defining the STN and with the available equipment items. More specifically:

Task i is defined by: U_i**:** set of units capable of performing task i;

S_i^{in}**:** set of states that feed task i;

S_i^{out}**:** set of states that task i produces as its outputs;

ρ_{is}^{in}**:** proportion of input of task i from state $s \in S_i^{\text{in}}$; $\sum_{s \in S_i^{\text{in}}} \rho_{is}^{\text{in}} = 1$;

ρ_{is}^{out}**:** proportion of output of task i to state $s \in S_i^{\text{out}}$; $\sum_{s \in S_i^{\text{out}}} \rho_{is}^{\text{out}} = 1$;

p_{is}**:** processing time for output of task i to state $s \in S_i^{\text{out}}$;

d_i**:** duration (completion time) for task i, $d_i \stackrel{\text{def}}{=} \max_{s \in S_i^{\text{out}}} p_{is}$;

State s is defined by: T_s^{out}**:** set of tasks receiving material from state s;

T_s^{in}**:** set of tasks producing material in state s;

z_s^0**:** initial stock in state s;

u_s**:** storage capacity dedicated to state s;

c_s^{P}**:** unit cost (price) of product produced in state s;

c_s^{S}**:** cost of storing a unit amount of material in state s.

Unit j is characterized by: I_j**:** set of tasks that can be performed by unit j;

$V_{ij}^{\max}, V_{ij}^{\min}$**:** respectively, maximum and minimum load of unit j when used for performing task i.

Let n, q, m denote, respectively, the number of tasks, states, and units. The scheduling problem for batch processing system is stated as:

Given: the STN of a batch process and all the information associated with it, as well as a time horizon of interest.

Determine: the timing of the operations for each unit (i.e. which task, if any, the unit performs at any time during the time horizon); and the flow of materials through the network.

Goal: maximize the total cost of the products produced minus the total storage cost during the time horizon.

2.5.1 MIP formulation

Our formulation is based on a discrete time representation. The time horizon of interest is divided into a number of intervals of equal duration. We number the intervals from 1 to H, and assume that interval t starts at time t and ends at time $t+1$. Events of any type — such as the start or end of processing individual batches of individual tasks, changes in the availability of processing equipment and etc. — are only happen at the interval boundaries.

Pre-emptive operations are not allowed and materials are transferred instantaneously from states to tasks and from tasks to states.

We introduce the following variables:

$x_{ijt} = 1$ if unit j starts processing task i at the beginning of time period t; $x_{ijt} = 0$ otherwise;

y_{ijt}: amount of material (batch size) that starts undergoing task i in unit j at the beginning of time period t.

z_{st}: amount of material stored in state s, at the beginning of time period t. To simplify presentation, we introduce an additional time interval $H + 1$ that represents the end of the time horizon. Then the value of $z_{s,H+1}$ is the amount of material in state s produced during the time horizon.

Now the MIP model is written as follows:

$$\sum_{s=1}^{q} c_s^{\mathrm{P}} z_{s,H+1} - \sum_{s=1}^{q} \sum_{t=1}^{H} c_s^{\mathrm{S}} z_{st} \to \max, \tag{2.6a}$$

$$\sum_{i \in I_j} x_{ijt} \le 1, \quad j = 1, \ldots, m;\ t = 1, \ldots, H, \tag{2.6b}$$

$$\sum_{l\in I_j}\sum_{\tau=t}^{t+d_i-1} x_{lj\tau} \le 1 + M(1-x_{ijt}), \quad j=1,\dots,m;\ i\in I_j;$$

$$t=1,\dots,H, \tag{2.6c}$$

$$V_{ij}^{\min} x_{ijt} \le y_{ijt} \le V_{ij}^{\min} x_{ijt}, \quad j=1,\dots,m;\ i\in I_j;$$

$$t=1,\dots,H, \tag{2.6d}$$

$$0 \le z_{st} \le u_s, \quad s=1,\dots,q;\ t=1,\dots,H, \tag{2.6e}$$

$$z_{s,t-1} + \sum_{i\in T_s^{\text{out}}:\, t>p_{is}} \rho_{is}^{\text{out}} \sum_{j\in U_i} y_{ij,t-p_{is}} = z_{st} + \sum_{i\in T_s^{\text{in}}} \rho_{is}^{\text{in}} \sum_{j\in U_i} y_{ijt},$$

$$s=1,\dots,q;\ t=1,\dots,H, \tag{2.6f}$$

$$z_{s,H} + \sum_{i\in T_s^{\text{out}}:\, t>p_{is}} \rho_{is}^{\text{out}} \sum_{j\in U_i} y_{ij,H-p_{is}} = z_{s,H+1}, \quad s=1,\dots,q, \tag{2.6g}$$

$$x_{ijt} = 0, \quad t > H-d_i,\ j=1,\dots,m;\ i\in I_j, \tag{2.6h}$$

$$x_{ijt}\in\{0,1\},\ y_{ijt}\in\mathbb{R}_+, \quad j=1,\dots,m;\ i\in I_j; t=1,\dots,H, \tag{2.6i}$$

$$z_{st}\in\mathbb{R}_+, \quad s=1,\dots,q;\ t=1,\dots,H+1. \tag{2.6j}$$

The objective (2.6a) is to maximize the total profit that equals the total cost of the produced materials minus the expenses for storing materials during the time horizon. The inequalities (2.6b) enforce that at any time t, an idle unit j can only start one task. The constraints (2.6c) impose the requirement that, if unit j starts performing task i at time t, then it cannot start any other task until i is finished. Used in (2.6c), M is a sufficiently big number so that the constraint is only binding if $x_{ijt} = 1$. The constraints (2.6d) enforce that the batch size of any task must be within the minimum and maximum capacities of the unit performing the task. The constraints (2.6e) impose stock limitations: the amount of material stored in any state s must not exceed the storage capacity for this state. Material balance constraints (2.6f) require that for any state s at each period t, the amount of material entering the state (the stock from the previous period plus the input delivered from the tasks that finished at period t) equals the amount of material leaving the state (the stock at t plus the amount of material consumed by the tasks that started at t). Note that z_{s0} is not a variable but a constant z_s^0, the initial stock at state s. The constraints (2.6g) are specializations of the balance constraints for period $H+1$, at this period no task starts. The constraints (2.6h) enforce tasks finish within the time horizon.

2.5.2 MIPshell Implementation

Our test STN is represented in a file whose contents is as in Listing 2.7. We see that there are five task. For example, the task named "Reaction 2" can be processed by the units Reactor_1 and Reactor_2. This task gets materials from states representing products Hot_A and Int_BC; these materials are mixed in the proportions of 40%

and 60%. Furthermore, the task output materials are directed to states representing products Int_AB and Prod_1 in the proportions of 60% and 40%. Production of both output products takes two hours.

Next we have nine states. For instance, the state representing the intermediate product AB is described by the set $\{3\}$ of task receiving product AB, the set $\{2, 4\}$ of task supplying the state, and the capacity of the state stock that equals 200kg, initial stock 0.0kg, and storing a unit of the product costs 0.1. It is undesirable to have any intermediate remaining in storage at the end of the horizon, and therefore the cost of all the intermediates is a negative value of -1. The three raw materials (Feed A, Feed B, and Feed C) are given costs of zero. A value of one kilogram of each of the two output products (Prod 1 and Prod 2) is 10 units.

The last section of the input file describes equipment units. Say Reactor 1 can fulfill tasks Reaction_1,Reaction_2,Reaction_3, and has lower and upper capacities of zero and 80kg, respectively.

Listing 2.7: Input file for example process

```
10 - time horizon

begin(tasks)
  Heating
  {Heater}
  {Fead_A} 1.0
  {Hot_A} 1.0  1
  Reaction_1
  {Reactor_1, Reactor_2}
  {Fead_B,Fead_C} 0.5 0.5
  {Int_BC} 1.0  2
  Reaction_2
  {Reactor_1,Reactor_2}
  {Hot_A,Int_BC} 0.4 0.6
  {Int_AB,Prod_1} 0.6 2   0.4 2
  Reaction_3
  {Reactor_1, Reactor_2}
  {Fead_C,Int_AB} 0.2 0.8
  {Impure_E} 1.0  1
  Separation
  {Still}
  {Impure_E} 1.0
  {Int_AB,Prod_2} 0.1 2  0.9 1
  end(tasks)

begin(states)
  Fead_A 1000 1000 0.0 0.0
  Fead_B 1000 1000 0.0 0.0
  Fead_C 1000 1000 0.0 0.0
  Hot_A 100 0.0 0.1 -1.0
  Int_AB 200 0.0 0.1 -1.0
```

```
  Int_BC 150 0.0 0.1 -1.0
  Impure_E 100 0.0 0.1 -1.0
  Prod_1 1000 0.0 0.0 10.0
  Prod_2 1000 0.0 0.0 10.0
end(states)

begin(units)
  Heater
  {Heating} 0 100
  Reactor_1
  {Reaction_1,Reaction_2,Reaction_3} 0 80
  Reactor_2
  {Reaction_1,Reaction_2,Reaction_3} 0 50
  Still
  {Separation}  0 200
end(units)
```

To solve the batch scheduling problem we developed a **C++** class, named **Cbatch**, which definition and implementation you can find in the folder

$MIPDIR/examples/mipshell/batch.

This class has the following members to describe problem instances:

- *H*: number of time periods in planning horizon;
- *TASKS*: set of tasks, ;
- *STATES*: set of states;
- *UNITS*: set of units;
- *K*: array of sets, where *K*(i) is set of units capable of performing task i;
- *Sin*: array of sets, where *Sin*(i) is set of states that feed task i;
- *Sout*: array of sets, where *Sout*(i) is set of states to which task i produces as its outputs;
- *Tin*: array of sets, where *Tin*(s) is set of tasks producing material in state s;
- *Tout*: array of sets, where *Tin*(s) is set of tasks receiving material from state s;
- *I*: array of sets, where *I*(j) is set of tasks that can be performed by unit j;
- *rho*: two-dimensional real array, where *rho*(i,j) is proportion of input of task i from state s in *Sin*(i);
- *u*: real array, where *u*(s) is storage capacity dedicated to state s;

- *z0*: real array, where *z0*(s) is initial stock in state s;
- *cs*: real array, where *cs*(s) is cost of storing a unit amount of material in state s;
- *cp*: real array, where *cp*(s) is unit cost (price) of product produced in state s;
- *Vmin*, *Vmax*: two-dimensional real arrays, where *Vmin*(i,j), *Vmax*(i,j) are, respectively, maximum and minimum load of unit j when used for performing task i.
- *p*: two-dimensional integer array, where *p*(i,s) is processing time for output of task i to state s in *Sout*(i);
- *dur*: integer array, where *dur*(i) is duration (completion time) for task i.

Here we presented only three crucial procedures of **Cbatch**:

- Listing 2.8 — procedure that reads input files;
- Listing 2.9 — procedure that implements MIP (2.6);
- Listing 2.10 — procedure that prints optimal schedules.

Listing 2.8: Implementation of **Cbatch**: procedure that reads input files

```
void readSTN(const char *fileName)
{
  std::ifstream fin(fileName);
  if (!fin.is_open()) {
    throw new CFileException("Cbatch::readSTN",fileName);
  }

  char str[256];
  int w,q;
  INDEX state, tsk, unit;

  fin >> H;
// reading tasks
  for (fin.getline(str,255);
       strcmp(str,"begin(tasks)"); fin.getline(str,255));
  for (fin >> str; strcmp(str,"end(tasks)"); fin >> str) {
    TASKS.insert(tsk=str);
    fin >> U.add(tsk) >> Sin.add(tsk);
    for (Sin(tsk).InitIt(); Sin(tsk).GetNext(state);)
      fin >> rho.add(state,tsk);
    fin >> Sout.add(tsk);
    w=0;
```

```
      for (Sout(tsk).InitIt(); Sout(tsk).GetNext(state);) {
         fin >> rho.add(tsk,state) >> q;
         p.add(tsk,state)=q;
         if (q > w)
            w=q;
      }
      dur.add(tsk)=w;
   }
// reading states
   for (fin.getline(str,255);
            strcmp(str,"begin(states)"); fin.getline(str,255));
   for (fin >> str; strcmp(str,"end(states)"); fin >> str) {
      STATES.add(state=str);
      Tin.add(state); Tout.add(state);
      fin >> u.add(state) >> z0.add(state)
          >> cs.add(state) >> cp.add(state);
   }
   for (TASKS.InitIt(); TASKS.GetNext(tsk);) {
      for (Sin(tsk).InitIt(); Sin(tsk).GetNext(state);)
         Tout(state).add(tsk);
      for (Sout(tsk).InitIt(); Sout(tsk).GetNext(state);)
         Tin(state).add(tsk);
   }
// reading units
   for (fin.getline(str,255);
            strcmp(str,"begin(units)"); fin.getline(str,255));
   for (fin >> str; strcmp(str,"end(units)"); fin >> str) {
      UNITS.add(unit=str);
      fin >> I.add(unit) >> Vmin.add(unit) >> Vmax.add(unit);
   }
   fin.close();
}
```

Listing 2.9: Implementation of **Cbatch**: **MIPshell** model for MIP (2.6)

```
int Cbatch::model()
{
   INDEX state, tsk, tsk1, unit;
   int t,tau, M=H*TASKS.GetSize();
   RANGE HORIZON(0,H-1), HORIZON1(0,H);
   VAR_ARRAY x("x",BIN,TASKS,UNITS,HORIZON),
             y("y",REAL_GE,TASKS,UNITS,HORIZON),
             z("z",REAL_GE,STATES,HORIZON1);
```

```
  maximize(sum(state in STATES) cp(state)*z(state,H)
                  - sum(state in STATES, t in [0,H)) cs(state)*z(state,t));
  forall(t in [0,H), unit in UNITS)
    sum(tsk in I(unit)) x(tsk,unit,t) <= 1;
  forall(unit in UNITS, tsk in I(unit), t in [0,H))
    sum(tsk1 in I(unit), tau in [t,t+dur(tsk)): tau < H)
        x(tsk1,unit,tau) <= 1 + M*(1-x(tsk,unit,t));
  forall(unit in UNITS, tsk in I(unit), t in [0,H)) {
    Vmin(unit)*x(tsk,unit,t) <= y(tsk,unit,t);
    y(tsk,unit,t) <= Vmax(unit)*x(tsk,unit,t);
    if (t+dur(tsk) > H) x(tsk,unit,t) == 0;
  }
  forall(state in STATES, t in [0,H))
    z(state,t) <= u(state);
  forall(state in STATES)
    z0(state) == z(state,0) +
        sum(tsk in Tout(state), unit in U(tsk))
            rho(state,tsk)*y(tsk,unit,0);
  forall(state in STATES, t in [1,H))
    z(state,t-1) + sum(tsk in Tin(state), unit in U(tsk): t >= p(tsk,state))
                    rho(tsk,state)*y(tsk,unit,t -p(tsk,state)) ==
    z(state,t) + sum(tsk in Tout(state), unit in U(tsk))
                    rho(state,tsk)*y(tsk,unit,t);
  forall(state in STATES)
    z(state,H) == z(state,H-1) +
        sum(tsk in Tin(state), unit in U(tsk): H >= p(tsk,state))
            rho(tsk,state)*y(tsk,unit,H-p(tsk,state));
  optimize();
  printSol(y,z);
  return 0;
}
```

Listing 2.10: Implementation of **Cbatch**: procedure that prints schedules

```
void Cbatch::printSol(VAR_ARRAY& y, VAR_ARRAY& z)
{
  char fileName[128];
  getprobname(fileName);
  strcat(fileName,".sol");
  std::ofstream fout(fileName);
  if (!fout.is_open()) {
    throw new CFileException("Cbatch::ptintSol",fileName);
```

```
  }

  INDEX state, tsk, unit;
  int t;

  fout << "Objective = " << getobj() << "\nStock: ";
  for (STATES.InitIt(); STATES.GetNext(state);)
    fout << getval(z(state,H)) << " ";
  fout << "\nSchedule:\n";
  for (UNITS.InitIt(); UNITS'GetNext(unit);) {
    fout << unit << ":\n";
    for (t=0; t < H; ++t)
      for (TASKS/InitIt(); TASKS.GetNext(tsk);)
        if (getval(y(tsk,unit,t)) > ZERO)
          << "batch of task " << tsk << ": starts at " << t
          << ", size = " << getval(y(tsk,unit,t)) << std::endl;
  }
  fout.close();
}
```

Running the program to solve our example process, we get the schedule presented in Listing 2.11. We see that the maximum profit is 2717.025 units corresponding to the production of 136.0 kg of Product 1 and 147.375 kg of Product 2, with a residual amount of 89.375 kg of intermediate AB.

Listing 2.11: Schedule for example process

```
Objective = 2717.03
Stock: 864 898 865.25 0 89.375 0 0 136 147.375
Schedule:
unit 0:
batch of task 0: start at 1, size = 52
batch of task 0: start at 3, size = 32
batch of task 0: start at 7, size = 52
unit 1:
batch of task 1: start at 0, size = 80
batch of task 2: start at 2, size = 80
batch of task 2: start at 4, size = 80
batch of task 1: start at 6, size = 78
batch of task 2: start at 8, size = 80
unit 2:
batch of task 1: start at 0, size = 46
batch of task 2: start at 2, size = 50
```

```
batch of task 3: start at 4, size = 50
batch of task 3: start at 5, size = 13.75
batch of task 3: start at 6, size = 50
batch of task 3: start at 7, size = 50
batch of task 2: start at 8, size = 50
unit 3:
batch of task 4: start at 5, size = 50
batch of task 4: start at 8, size = 113.75
```

Chapter 3

Facility Layout

The formats by which departments are arranged in a facility are defined by the general pattern of work flow; there are three basic formats — process layout, product layout, and fixed-position layout — and one hybrid format called group technology or cellular layout.

A *process layout* (also called *job-shop* or *functional layout*) is a format in which similar equipment or functions are grouped together, such as lathes in one area and all stamping machines in another. A part being worked on then travels, according to the prescribed sequence of operations, from area to area, where the proper machines are located for each operation. This type of layout is typical, for examples, for hospitals, where areas dedicated to particular types of medical care, such as maternity wards and intensive care units.

A *product layout* (also called *flow-shop layout*) is one in which equipment or work processes are arranged according to the progressive steps by which the product is made. The path for each part is, in effect, a straight line. Production lines for shoes, cars, watches, and chemical plants are all product layouts.

A *group technology* (GT) *layout* (also called *cellular layout*) groups dissimilar machines into work centers (or cells) to work on products that have similar shapes and processing requirements. A GT layout is similar to process layout in that cells are designed to perform a specific set of processes, and it is similar to product layout in that the cells are dedicated to a limited range of products.

In a *fixed-position layout*, a product (because of its bulk or weight) remains at one location. Manufacturing equipment is moved to the product rather vice versa. Shipyards, construction sites are examples of this format.

3.1 Process Layout

The most common approach to developing a process layout is to arrange departments consisting of like processes in a way that optimizes their relative placement. In many situations, optimal placement often means placing departments with large amount of interdepartment traffic adjacent to one another.

Suppose that we want to arrange n departments among m sites to minimize

the cost of moving departments to new places and the interdepartment material handling cost. Let p_{is} denote the cost of moving department i to site s. Usually, $p_{is} = 0$ if department i is currently situated at site s, and p_{is} is a rather big number (penalty) if department i cannot be moved to site s. Let us also assume that expected expenses on transporting materials between departments i and j, if they where situated at sites s and r, are c_{ijsr} during a planning horizon (c_{ijsr} depends on the material traffic between the departments as well as the distance between the sites).

To write an IP model, we introduce two sets of binary variables:

$x_{is} = 1$ if department i is allocated at site $s \in S_i$, and $x_{is} = 0$ otherwise;

$y_{ijsr} = 1$ only if $x_{is} = x_{jr} = 1$, and $y_{ijsr} = 0$ otherwise.

With these variables the problem is formulated as follows:

$$\sum_{i=1}^{n}\sum_{s=1}^{m} p_{is}x_{is} + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{s=1}^{m}\sum_{r=1}^{m} c_{ijsr}y_{ijsr} \to \min, \tag{3.1a}$$

$$\sum_{s=1}^{m} x_{is} = 1, \quad i = 1,\dots,n, \tag{3.1b}$$

$$\sum_{i=1}^{n} x_{is} \le 1, \quad s = 1,\dots,n, \tag{3.1c}$$

$$2y_{ijsr} \le x_{is} + x_{jr}, \quad s,r = 1,\dots,m,\ j = i+1,\dots,n,\ i = 1,\dots,n-1, \tag{3.1d}$$

$$\sum_{s=1}^{m}\sum_{r=1}^{m} y_{ijsr} = 1, \quad j = i+1,\dots,n,\ i = 1,\dots,n-1, \tag{3.1e}$$

$$x_{ij} \in \{0,1\}, \quad j = i+1,\dots,n,\ i = 1,\dots,n-1, \tag{3.1f}$$

$$y_{ijsr} \in \{0,1\}, \quad s,r = 1,\dots,m,\ j = i+1,\dots,n,\ i = 1,\dots,n-1. \tag{3.1g}$$

The objective (3.1a) is to minimize the total expenses on moving the departments and the expenses on transporting materials between the all departments. The equations (3.1b) require that each department be moved to exactly one site, and the equations (3.1c) do not allow two departments to share one site. The inequalities (3.1d) and (3.1e) impose the required relation between the x an y variables (see definition of y_{ijsr}).

3.1.1 MIPshell-implementation

We implemented the process layout problem as a C++ class **Cproclayout** which definition is presented in Listing 3.1.

Listing 3.1: **MIPshell** Class **Cproclayout**: definition

```
#include <mipshell.h>

class Cproclayout: public CProblem
{
  int m_iN, m_iM;
  int *m_ipMovingCost, *m_ipDist, *m_ipFlow;
public:
  Cproclayout(const char *name);
#ifdef __THREADS_
  Cproclayout(const Cproclayout &other, int thread);
  CMIP* clone(const CMIP *pMip, int thread);
#endif
  virtual ~Cproclayout();

  int model();

  void readData(const char *fileName);
  void printSol(VAR_VECTOR &x);

  int c(int i, int j, int s, int r);
  int p(int i, int s) return m_ipMovingCost[i*m_iM+s];
};
```

Cproclayout has the following members for storing problem instances:

- *m_iN*: number of departments;
- *m_iM*: number of sites;
- *m_ipMovingCost*: integer array of size *m_iN*·*m_iM*, where *m_ipMovingCost*[i,s] is cost of moving department i to site s;
- *m_ipDist*: integer array of size *m_iM*(*m_iM*-1)/2 to store upper triangle of distance matrix;
- *m_ipFlow*: integer array of size *m_iN*(*m_iN*-1)/2 to store upper triangle of interdepartment material-flow matrix.

The default constructor of **Cproclayout**

Cproclayout(**const char** *name)

gets as its input parameter the name of a text file (without extension **.txt**) describing an instance of the process layout problem, and then calls

```
void readData(const char *fileName)
```

to read the instance.

A **MIPshell** implementation of IP (3.1) is given in Listing 3.2. In place of the standard *printsol* procedure we use a *printSol* procedure that prints the solution found in a more readable way.

Listing 3.2: Process layout **MIPshell** model

```
int Cproclayout::model()
{
  int i,j,s,r, n=m_iN, m=m_iM;
  VAR_VECTOR x("x",BIN,n,m), y("y",BIN,n,n,m,m);

  minimize(sum(i in [0,n), s in [0,m)) p(i,s)*x(i,s) +
           sum(i in [0,n-1), j in [i+1,n), s in [0,m), r in [0,m)) c(i,j,s,r)*y(i,j,s,r));

  forall(i in [0,n))
    sum(s in [0,m)) x(i,s) == 1;

  forall(s in [0,m))
    sum(i in [0,n)) x(i,s) <= 1;

  forall(i in [0,n-1), j in [i+1,n))
    sum(s in [0,m), r in [0,m)) y(i,j,s,r) == 1;

  forall(i in [0,n-1), j in [i+1,n), s in [0,m), r in [0,m))
    2*y(i,j,s,r) - x(i,s) - x(j,r) <= 0;

  optimize();
  printSol(x);
  return 0;
}
```

3.1.2 Illustrative Example

Consider a low-volume toy factory with eight departments:

1) shipping and receiving,

2) sewing,

3) metal forming,

4) plastic molding and stamping,

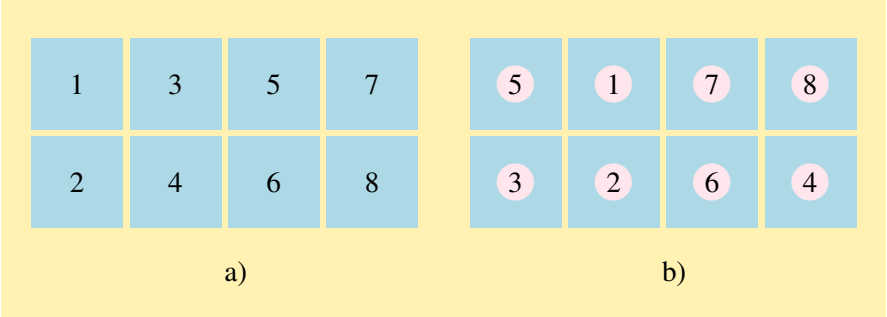

Figure 3.1: Plan of factory building and initial arrangement

Table 3.1: Flows between departments

	2	3	4	5	6	7	8
1	0	60	150	40	180	30	20
2		20	100	20	7	23	0
3			0	90	125	178	98
4				75	90	95	80
5					0	182	162
6						115	325
7							8

5) small toy assembly,

6) large toy assembly,

7) mechanism assembly,

8) painting.

Suppose that we want to arrange these eight departments of a toy factory to minimize the total cost of moving departments to new locations and the interdepartment material handling cost during planning horizon of 5 years.

To make presentation clearer, let us make some simplifying assumptions. The factory building has 8 rooms of equal size, 10 m wide and 10 m long, that are arranged as in Figure 3.1 a). Initially department i is allocated in room i ($i = 1, \ldots, 8$). The cost of moving any department to a new location is \$200. Therefore, moving costs are defined by the rule: $p_{ii} = 0$, and $p_{is} = 200$ for $i \neq s$.

All materials are transported in a standard-size crate by a forklift truck, one crate to a truck. Suppose that transportation costs are \$1 to move one crate between adjacent departments and \$1 extra for each department in between. The

expected material flows between departments for one year of operation are given in Table 3.1.

The per year interdepartment material handling cost for the initial arrangement of departments is calculated as follows:

$$\begin{aligned}
&0\cdot 1+60\cdot 1+150\cdot 2+40\cdot 2+180\cdot 3+\ 30\cdot 3+\ 20\cdot 4\\
&\quad +20\cdot 2+100\cdot 1+20\cdot 3+\ \ 7\cdot 2+\ 23\cdot 4+\ \ 0\cdot 3\\
&\qquad +\ \ 0\cdot 1+90\cdot 1+125\cdot 1+178\cdot 2+\ 98\cdot 3\\
&\qquad\quad +75\cdot 2+\ 90\cdot 1+\ 95\cdot 3+\ 80\cdot 2\\
&\qquad\qquad +\ \ 0\cdot 1+182\cdot 1+162\cdot 2\\
&\qquad\qquad\quad +115\cdot 2+325\cdot 1\\
&\qquad\qquad\qquad +\ \ 8\cdot 1=4075.
\end{aligned}$$

Thus, the material handling cost during the planning horizon is $5\cdot\$4075 = \$20375$.

To solve this instance of the process layout problem, we described it in a text file (named **`test1.txt`**) whose contest is presented in Listing 3.3.

Listing 3.3: Input file for process layout program

```
8 8

  0 200 200 200 200 200 200 200
200   0 200 200 200 200 200 200
200 200   0 200 200 200 200 200
200 200 200   0 200 200 200 200
200 200 200 200   0 200 200 200
200 200 200 200 200   0 200 200
200 200 200 200 200 200   0 200
200 200 200 200 200 200 200   0

1 1 2 2 3 3 4
  2 1 3 2 4 3
    1 1 2 2 3
      2 1 3 2
        1 1 2
          2 1
            1

0 300 750 200 900 150  100
  100 500 100  35 115    0
        0 450 625 890  490
          375 450 475  400
                0 910  840
                  575 1625
                       40
```

To solve our instance, we first enter the directory `tests` where **`test1.txt`** is stored, and then enter the command

../bin/proclayout **test1**

to get in `tests` the text file named **`test1.sol`** with the content as that in Listig 3.4.

Listing 3.4: Solution for process-layout example

```
Total expenses: 18500
i => s  means that department i moves to site s
   1 => 1
   2 => 2
   3 => 7
   4 => 4
   5 => 6
   6 => 3
   7 => 8
   8 => 5
```

We see that, if the factory reallocates its departments as it is prescribed in Listing 3.4, it will profit $20375 - 18500 = \$1875$ during the planning horizon of five years.

3.2 Balancing Assembly Lines

Assembly lines are special product-layout production systems that are typical for the industrial production of high quantity standardized commodities. An assembly line consists of a number of work stations arranged along a conveyor belt. The work pieces are consecutively launched down the conveyor belt and are moved from one station to the next. At each station, one or several tasks necessary to manufacture the product are performed. The tasks in an assembly process are typically ordered, i.e. there may be precedence requirements that must be enforced. The problem of distributing the tasks among the stations with respect to some objective function is called the *assembly line balancing problem* (ALBP). Various classes of assembly line balancing problems have been studied in the literature[1]. We will consider here the simple assembly line balancing problem which is the core of many other ALBP's.

[1] A. Scholl. *Balancing and sequencing of assembly lines*, Heidelberg: Physica-Verlag, 1999.

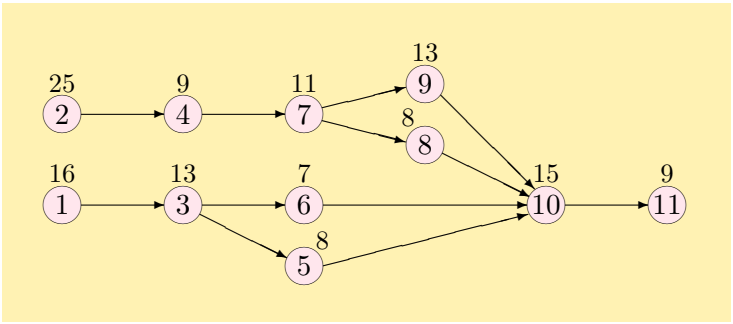

Figure 3.2: Example of SALBP

The manufacturing of some product consists of a set of *tasks* $\mathcal{T} = \{1, \ldots, n\}$. We denote by t_j the processing time of task $j \in \mathcal{T}$. There is a precedence relation on the tasks that are represented by a digraph $G = (\mathcal{T}, E)$, where $(j_1, j_2) \in E$ means that task j_1 is an immediate predecessor of task j_2. Suppose that the demand for the product is such that the assembly line must have a cycle time C. Thus the running time of each station must not exceed the cycle time.

The *simple assembly line balancing problem* (SALBP) is to decide what is the minimum number of stations that is enough for the line running with the given cycle time to fulfill all the operations in an order consistent with the precedence relation.

An example of SALBP is presented in Figure 3.2. Here vertices represent operations, and number over vertices are operation processing times.

To formulate SALBP as an IP, we need to know an upper bound, m, of the number of needed stations. In particular, we can set m to be the the number of station in a solution build by one of the heuristics developed for solving SALBPs.

For example, let us consider the heuristic that assigns operations, respecting precedence relations, first to Station 1, then to Station 2, and so on until all the operations have been assigned to the stations. If we apply this heuristic to the example of Figure 3.2 when the cycling time is $C = 45$, we get the following assignment:

- operations 1 and 2 are accomplished by Station 1,
- operations 3, 4, 5, and 6 — by Station 2,
- operations 7, 8, and 9 — by Station 3,
- operations 10 and 11 — by Station 4.

So in this example we can set $m = 4$.

Let us also note that this heuristic solution is not optimal as there exists an assignment that uses only three stations:

- operations 1, 3, 5, and 6 are accomplished by Station 1,
- operations 2, 4, and 7 — by Station 2,
- operations 8, 9, 10, and 11 — by Station 3.

We introduce the following variables:

$y_i = 1$ if station i is open (used), $y_i = 0$ otherwise;

x_{ij} if task j is assigned to station i, $x_{ij} = 0$ otherwise;

$z_j = i$ if task j is assigned to station i.

With these variables the model is

$$\sum_{i=1}^{m} y_i \to \min \tag{3.2a}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j = 1, \ldots, n, \tag{3.2b}$$

$$\sum_{j=1}^{n} t_j x_{ij} \le C y_i, \quad i = 1, \ldots, m, \tag{3.2c}$$

$$\sum_{i=1}^{m} i\, x_{ij} = z_j, \quad j = 1, \ldots, n, \tag{3.2d}$$

$$z_{j_1} \le z_{j_2}, \quad (j_1, j_2) \in E, \tag{3.2e}$$

$$y_{i-1} \ge y_i, \quad i = 2, \ldots, m, \tag{3.2f}$$

$$x_{ij} \le y_i, \quad j = 1, \ldots, n;\ i = 1, \ldots, m, \tag{3.2g}$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, \ldots, m;\ j = 1, \ldots, n, \tag{3.2h}$$

$$y_i \in \{0, 1\}, \quad i = 1, \ldots, m, \tag{3.2i}$$

$$z_j \in \mathbf{R}_+, \quad j = 1, \ldots, n. \tag{3.2j}$$

The objective (3.2a) is to minimize the number of open stations. The constraints (3.2b) induce that each task is assign to exactly one station. The capacity constraints (3.2c) enforce that the total running time of each open stations does not exceed the cycle time. The constraints (3.2d) establish the relation between the assignment variables, binary x and integer z. The precedence relation constraints (3.2e) induce that, for any pair $(j_1, j_2) \in E$ of related tasks, task j_1 is assigned to the same or an earlier station than task j_2; this guaranties that task j_1 can be processed before task j_2 starts. The constraints (3.2f) and (3.2g) enforce that earlier stations are opened first.

To solve SALBPs, we developed a C++ class **Cassembly** which has the following members for storing problem instances:

- *m_iOpNum*: number of operations;

- *m_iStNum*: number of stations;
- *m_iRewlNum*: number of precedence relations;
- *m_iCycleTime*: cycle time;
- *m_ipTime*: integer array of size *m_iOpNum*, where *m_ipTime*[j] is processing time of operation j;
- *m_ipPrec* and *m_ipSucc*: integer arrays of size *m_iRelNum*, where operation *m_ipPrec*[i] precedes operation *m_ipSucc*[i].

The default constructor of **Cassembly**

Cassembly(**const char** **name*)

gets as its input parameter the name of a text file (without extension **.txt**) describing an instance of SABL, and then calls

void ***readData***(**const char** **fileName*)

to read the instance.

A straightforward **MIPshell** implementation of IP (3.2) is given in Listing 3.5. In place of the standard ***printsol*** procedure we use a ***printSol*** procedure that prints the solution found in a more readable way.

Listing 3.5: **MIPshell** line balancing procedure

```
#define prec(e) m_ipPrec[e]
#define succ(e) m_ipSucc[e]
#define t(j) m_ipTime[j]

void Cassembly::model()
{
  int i,j, m=m_iStNum, n=m_iOpNum, q=m_iRelNum, C=m_iCycleTime;
  VAR_VECTOR x("x",BIN,m,n), y("y",BIN,m), z("z",REAL_GE,n);

  minimize(sum(i in [0,m)) y(i));
  forall(j in [0,n)) {
    sum(i in [0,m)) x(i,j) == 1;
    sum(i in [0,m)) (i+1)*x(i,j) == z(j);
  }
  forall(i in [0,m)) {
    sum(j in [0,n)) t(j)* x(i,j) <= C*y(i);
    forall(j in [0,n))
      x(i,j) <= y(i);
  }
  forall(i in [0,q))
```

```
    z(prec(i)) <= z(succ(i));
  forall(i in [1,m))
    y(i-1) >= y(i);
  optimize();
  printSol(x);
}
```

Chapter 4

Service Management

Most authorities consider services as economic activities whose output is not a physical product but a time-perishable, intangible process performed to a customer acting in the role of co-producer (James Fitzsimmons). In services, location of the service facility and direct customer involvement in creating the output are often essential factors while in goods production they usually are not. Measuring service productivity also has its own specifics.

4.1 Service Facility Location

The problem of facility location is critical to a company's eventual success. Service companies' location decisions are guided by variety of criteria. A location close to the customer is especially important because this enables faster delivery goods and services.

Given a set of customer locations $N = \{1, \ldots, n\}$ with b_j customers at location $j \in N$, a set of potential sites $M = \{1, \ldots, m\}$ for locating depots, a fixed cost f_i of locating a depot at site i, a capacity a_i of the depot at site i, and a cost c_{ij} of serving customer j from site i during some planning horizon. The *facility location problem* (FLP) is to decide where to locate depots so that to minimize the total cost of locating depots and serving customers.

Choosing the following decision variables

$y_i = 1$ if depot is located at site i and $y_i = 0$ otherwise,

x_{ij}: number of cuctomers of location j served from depot established at site i,

we formulate the problem as follows:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} f_i y_i \to \min \tag{4.1a}$$

$$\sum_{i=1}^{m} x_{ij} = b_j, \quad j = 1,\dots,n, \tag{4.1b}$$

$$\sum_{j=1}^{n} x_{ij} \le a_i y_i, \quad i = 1,\dots,m. \tag{4.1c}$$

$$x_{ij} \le \min\{a_i, b_j\} y_i, \quad i = 1,\dots,m,\ j = 1,\dots,n, \tag{4.1d}$$

$$y_i \in \{0,1\}, \quad i = 1,\dots,m, \tag{4.1e}$$

$$x_{ij} \in \mathbb{Z}_+, \quad i = 1,\dots,m,\ j = 1,\dots,n. \tag{4.1f}$$

The constraints (4.1b) insure that each customer is served. The capacity constraints (4.1c) induce that at most a_i customers can be served from site i. The redundant constraints (4.1d) are implied by the capacity constraints. They were introduced to strengthen the IP formulation.

Let us note, that if all $c_{ij} = 0$ and all $f_i = 1$, then problem (4.1) is to minimize the number of depots needed to serve all customers.

4.1.1 MIPshell Implementation

A **MIPshell** application for solving FLP's is located in the following folder

$MIPDIR/examples/mipshell/fl.

The core of this application is a **C++** class **Cfl**. Its members for storing problem instances are:

- *m_iLocationNum*: number of locations;
- *m_iSiteNum*: number of potential sites for locating depots;
- *m_iQ*: maximum number of depots;
- *m_ipPopulation*: integer array of size *m_iLocationNum*, where *m_ipPopulation*[j] is number of customers at location j;
- *m_ipFixedCost*: integer array of size *m_iPlaceNum*, where *m_ipFixedCost*[i] is fixed cost of locating depot at site i;
- *m_ipCapacity*: integer array of size *m_iPlaceNum*, where *m_ipCapacity*[i] is capacity of depotif it is allocated at site i;
- *m_ipCost*: integer array of size *m_iPlaceNum* $\times$ *m_iCustomNum*, where *m_ipCost*[i,j] is cost of serving customer j from depot i;

The default constructor of **Cfl**

Cfl(**const char** **fileName*)

just calls the function

void ***readData***(**const char** **fileName*)

to read a text file which name (without extension ".txt") is given by the parameter *fileName*. The data read from the file is stored in the class members described above. The function also allocates memory for the array members of **Cfl**.

A straightforward **MIPshell** implementation of the IP (4.1) is presented in Listing 4.1.

Listing 4.1: **MIPshell** model for facility location problem

```
#define f(i) m_ipFixedCost[i]
#define a(i) m_ipCapacity[i]
#define b(j) m_ipPopulation[j]
#define c(i,j) m_ipCost[i*n+j]
#define min(a,b) ((a ¡ b)? a: b)
int Cfl::model()
{
  int i,j, m=m_iSiteNum, n=m_iLocationNum, q=m_iQ;
  VAR_VECTOR y("y",BIN,m), x("x",INT_GE,m,n);

  minimize(sum(i in [0,m)) f(i)*y(i) + sum(i in [0,m), j in [0,n)) c(i,j)*x(i,j));
  forall(j in [0,n))
    sum(i in [0,m)) x(i,j) == b(j);
  forall(i in [0,m))
    sum(j in [0,n)) x(i,j) <= b(i)*y(i);
  forall(i in [0,m), j in [0,n))
    x(i,j) <= min(a(i),b(j))*y(i);

  optimize();
  printSol(x,y);
  return 0;
}
```

4.1.2 Locating Two medical Clinics

Two clinics, both of capacity 40, are to be established to provide medical care for people living in four communities, A, B, C, D and E. The population of each community and distances between communities are given in Table 4.1. Assume that clinics can be allocated at any of the communities with the same expenses (so we

Table 4.1: Population and distances between communities

Community	Distance to (km)					Population (thousands)
	A	B	C	D	E	
A	0	11	8	12	15	10
B	11	0	10	7	13	8
C	8	10	0	9	9	20
D	12	7	9	0	6	12
E	15	13	9	6	0	14

may assume that the fixed costs of allocating clinics are zeroes), and the population of each community is evenly distributed within the community's boundaries. The problem is to allocate two clinics so that to minimize the overall travel distance.

To solve this example with our program, we prepared a text file, named **`test1.txt`**, which contents is as the following.

```
5 5 2

10 8 20 12 14

0 40 0 40 0 40 0 40 0 40

 0 11  8 12 15
11  0 10  7 13
 8 10  0  9  9
12  7  9  0  6
15 13  9  6  0
```

If you enter the folder **`$MIPDIR/examples/mipshell/fl/tests`** and run the program **`fl`** with the only parameter **`test1.txt`**, you will get the solution written into the text file **`test1.sol`**.

```
Overall travel distance is 220
===== Facility at site 3 serves customers:
10 from loc. 1
20 from loc. 3
============================================
===== Facility at site 4 serves customers:
8 from loc. 2
12 from loc. 4
14 from loc. 5
============================================
```

4.1.3 Location Of Automated Teller Machines

A number of variations of the base facility location problem (4.1) are known. In this section we consider one of such variations.

A bank is planning to serve n rural communities with automated teller machines (ATMs). Let b_j be the population of community j, and let t_{ij} be the travel time in minutes between communities i and j. The bank wants to place the minimum number of ATMs so that at least p percent of customers will be within T minutes' travel time of the nearest ATMs.

Setting $t^{\max} \stackrel{\text{def}}{=} \max_{1\le i,j\le n} t_{ij}$ and introducing the following decision variables

$y_i = 1$ if ATM is installed at community i, and $y_i = 0$ otherwise,

$x_{ij} = 1$ if community j is served by ATM installed at community i,

$z_j = 1$ if some ATM is within T minutes' travel time of community j, and $z_j = 0$ otherwise,

we formulate the problem as follows:

$$\sum_{i=1}^{n} y_i \to \min \tag{4.2a}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1,\dots,n, \tag{4.2b}$$

$$x_{ij} \le y_i, \quad i,j = 1,\dots,n, \tag{4.2c}$$

$$\sum_{i=1}^{n} t_{ij}x_{ij} + t^{\max}z_j \le T + t^{\max}, \quad j = 1,\dots,n, \tag{4.2d}$$

$$\sum_{j=1}^{n} b_j z_j \ge (p/100)\sum_{j=1}^{n} b_j, \tag{4.2e}$$

$$y_i \in \{0,1\}, \quad i = 1,\dots,n, \tag{4.2f}$$

$$x_{ij} \in \{0,1\}, \quad i,j = 1,\dots,n, \tag{4.2g}$$

$$z_j \in \{0,1\}, \quad j = 1,\dots,n. \tag{4.2h}$$

The objective (4.2a) is to minimize the number of ATMs installed. The constraints (4.1b) insure that each community is served by exactly one ATM. The constraints (4.2c) do not allow customers to be served from those sites where no ATM is installed. Inequality j in (4.2d) is a restriction only if $z_j = 1$, and then community j is within T minutes' travel time of an ATM. The inequality (4.2e) that at least p percent of customers are within T minutes' travel time of an ATM.

It will be a good exercise if you write a **MIPshell**-program that solves the IP (4.2).

4.2 Data Envelopment Analysis

Data Envelopment Analysis (*DEA*) is a technique used to measure the relative performance of branches of multisite service organizations such as banks, public agencies, fast food services, and many others. DEA provides a more comprehensive and reliable measure of efficiency than any measure composed of a set of operating ratios or profit measures. A DEA model compares each branch with all other branches and computes an efficiency rating that is the ratio of weighted product or service outputs to weighted resource inputs. A branch is deemed to be efficient if it is not possible to find a mixture of proportions of other branches whose combined inputs do not exceed those of the branch being estimated, but whose outputs are equal to, or exceed, those of the branch being estimated. Should this not be possible the branch is deemed to be inefficient and the comparator branches can be identified. The key advantage is that DEA permits using multiple inputs such as materials and labor hours, and multiple outputs such as products sold and repeat customers.

Let us assume that there are n service units which are numbered as $1, \dots, n$. For one time period, service unit i ($i = 1, \dots, n$) used r_{ij} units of resource j ($j = 1, \dots, m$), and provided s_{ik} services of type k ($k = 1, \dots, l$). The efficiency of unit i is estimated by the ratio

$$E_i(u, v) \stackrel{\text{def}}{=} \frac{\sum_{k=1}^{l} s_{ik} u_k}{\sum_{j=1}^{m} r_{ij} v_j},$$

where u_k and v_j are weights to be determined by the DEA model.

The rating of service unit i_0 is computed by solving the following problem of fractional linear programming:

$$\max\{E_{i_0}(u, v) : \; E_i(u, v) \le 1, \; i = 1, \dots, n; \; i \ne i_0; \; u \in \mathbb{R}^l_+, \; v \in \mathbb{R}^m_+\}.$$

This problem can be reformulated as the following LP:

$$\sum_{k=1}^{l} s_{i_0 k} u_k \to \max, \tag{4.3a}$$

$$\sum_{j=1}^{m} r_{i_0 j} v_j = 1, \tag{4.3b}$$

$$\sum_{k=1}^{l} s_{ik} u_k \le \sum_{j=1}^{m} r_{ij} v_j, \quad i = 1, \dots, n; \; i \ne i_0 \tag{4.3c}$$

$$u_k \ge 0, \quad k = 1, \dots, l, \tag{4.3d}$$

$$v_j \ge 0, \quad j = 1, \dots, m. \tag{4.3e}$$

Table 4.2: Data for DEA analysis

Service unit	Labour (hours)	Material (dollars)	Meals sold
1	32	3200	1600
2	16	600	400
3	24	600	600
4	24	400	400
5	16	160	200
6	8	40	80

Let (u^*, v^*) be an optimal solution to problem (4.3). If $E_{i_0}(u^*, v^*) < 1$, then service unit i_0 worked inefficiently, and the unit can improve its efficiency by using some experience of more efficient units i for which $E_i(u^*, v^*) = 1$.

To demonstrate, let us consider a small example. A firm established six units each located in strip shopping center parking lot. Only a standard meal consisting of a burger, fries, and a drink is sold in each unit. Management has decided to use DEA to improve productivity by identifying which units are using their resources most efficiently. Table 4.2 presents data for DEA analysis.

To compute the rating of Firm 1, we formulate the following LP:

$$\begin{aligned} E_1 = 1600u_1 &\to \max, \\ 32v_1 - 3200v_2 &= 1, \\ 400u_1 - 16v_1 - 600v_2 &\le 0, \\ 600u_1 - 24v_1 - 600v_2 &\le 0, \\ 400u_1 - 24v_1 - 400v_2 &\le 0, \\ 200u_1 - 16v_1 - 160v_2 &\le 0, \\ 80u_1 - 8v_1 - 40v_2 &\le 0, \\ u_1, v_1, v_2 &\ge 0. \end{aligned}$$

4.2.1 MIPshell implementation

To compute the ratings for all n firms, we need to solve n LPs of type (4.3). Therefore, our **MIPshell** implementation of the DEA problem somewhat differs from most other applications in this manual. To solve one particular LP (4.3), we developed a **C++** class named **Cdea**. Its definition is given in Listing 4.2. But now this class do not allocate memory for storing input data; instead **Cdea** has only a pointer to an array with input parameters r_{ij} and s_{ik}:

- *m_iUnitNum*: number of service units (*n*);
- *m_iResNum*: number of resources (*m*);
- *m_iServiceNum*: number of services provided (*l*);
- *m_dpInOut*: real array of size $n \times (m + l)$, where *m_dpInOut*[i*(*m*+*l*)+j] stores *r*(i,j), and*m_dpInOut*[i*(*m*+*l*)+*m*+k] stores *s*(i,k).

Listing 4.2: Class **Cdea**

```
#include <mipshell.h>

class Cdea: public CProblem
{
  int m_iUnitNum, m_iResNum, m_iServiceNum;
  double *m_dpInOut; // only pointer
public:
  Cdea(const char *name, int unitNum, int resNum, int serviceNum,
       double *dpInOut);
#ifdef __THREADS_
  Cdea(const Cdea &other, int thread);
  CMIP* clone(const CMIP *pMip, int thread);
#endif
  virtual Cdea();

  int model(int i0, double &rating,
       double *dpV, double *dpU, double *dpS, double *dpRes);
};
```

In **Cdea**, the standard declaration of ***model*** was changed to the following

```
int model(int i0, double &rating,
       double *dpV, double *dpU, double *dpS, double *dpRes),
```

where

- *i0*: service unit to be estimated;
- *rating*: rating of unit *i0* (output parameter);
- *dpV*: output real array of size *m_iResNum*, where *dpV*[j] is weight of resource j (v_j);
- *dpU*: output real array of size *m_iServiceNum*, where *dpU*[k] is weight of output k (u_k);

- *dpS*, *dpR*: output real arrays of size *m_iUnitNum*, where *dpS*[i] and *dpR*[i] are, respectively, combined output and input for weights u and v that are optimal for unit *i0*.

Our implementation of ***model*** is presented in Listing 4.3. We first formulate LP (4.3), and then, when the LP has been solved (after calling ***optimize***), we compute the value of the output parameters, *rating*, *dpS*, and *dpR*.

Listing 4.3: **MIPshell** model for DEA

```
#define s(i,k) m_dpInOut[i*(m+l)+m+k]
#define r(i,j) m_dpInOut[i*(m+l)+j]

int Cdea::model(int i0, double &rating,
          double *dpV, double *dpU, double *dpR, double *dpS)
{
   int i,j,k, n=m_iUnitNum, m=m_iResNum, l=m_iServiceNum;
   VAR_VECTOR u("u",REAL_GE,l), v("v",REAL_GE,m);

   maximize(sum(k in [0,l)) s(i0,k)*u(k));

   sum(j in [0,m)) r(i0,j)*v(j) == 1;

   forall(i in [0,n): i != i0)
      sum(k in [0,l)) s(i,k)*u(k) <= sum(j in [0,m)) r(i,j)*v(j);

   optimize();

// preparing the output
   rating=getobj();
   forall(k in [0,l))
      dpU[k]=getval(u(k));
   forall(j in [0,m))
      dpV[j]=getval(v(j));

   forall(i in [0,n)) {
      dpS[i]=dpR[i]=0.0;
      forall(k in [0,l))
         dpS[i]+=s(i,k)*dpU[k];
      forall(j in [0,m))
         dpR[i]+=r(i,j)*dpV[j];
   }

   return 0;
```

} // end of **Cdea**::***model***

Unlike most of the other applications in this manual, a part of functionality of the DEA application is implemented outside of its base class. Here, two functions, ***readData*** and ***solveAll***, are called from the application ***main*** function (see Listing 4.4). The former function, ***readData***, — which implementation is straightforward and, therefore, is not presented here — reads problem instances from a text file which name is passed as the first function argument. The other arguments are output parameters which meanings are the same as the meanings of the same name (but without suffix "m_") members of **Cdea**. The function ***readData*** also allocates memory for the array `dpInOut` that stores parameters s_{ik} and r_{ij}.

Listing 4.4: Implementation of DEA: function ***main***

```
int main(int argc, const char *argv[])
{
  if (argc < 2) {
    std::cerr << "Enter file name!\n";
    return 1;
  }

  int unitNum, resNum, serviceNum;
  double *dpInOut=0;

  try {
    readData(argv[1],unitNum,resNum,serviceNum,dpInOut);
    solveAll(argv[1],unitNum,resNum,serviceNum,dpInOut);
  }
  catch(CException* pe) {
    std::cerr << pe->GetErrorMessage() << std::endl;
    delete pe;
    return 1;
  }
  if (dpInOut)
    delete[] dpInOut;

  return 0;
}
```

The function ***solveAll*** from Listing 4.5, n times calls ***model*** to compute the ratings of all n service units. This function also calls the function ***printSol***,— which implementation is straightforward and, therefore, is not presented here —

that writes units ratings and relative information to a text file, which name is the problem name appended with the extension ".sol".

Listing 4.5: Implementation of DEA: function *solveAll*

```
void solveAll(const char *probName,
              int unitNum, int resNum, int serviceNum, double *dpInOut)
{
  double rating, *dpR, *dpS, *dpU, *dpV;
  if (!(dpR = new(std::nothrow) double[2*unitNum+ resNum+serviceNum])) {
    throw CMemoryException("solveAll (1)");
  }
  dpU=(dpV=(dpS=dpR+unitNum)+unitNum)+resNum;
  char fileName[128];
  strcpy(fileName,probName);
  strcat(fileName,".sol");
  std::ofstream fout(fileName);
  if (!fout.is_open()) {
    throw new CFileException("solveAll",fileName);
  }

  Cdea *pDea;
  int n=unitNum;
  for (int i=0; i < n; ++i) {
    if (!(pDea=new Cdea(probName,unitNum, resNum,serviceNum,
                        dpInOut))) {
      fout.close();
      delete[] dpR;
      throw CMemoryException("solveAll (2)");
    }
    pDea->model(i,rating,dpV,dpU,dpR,dpS);
    printSol(fout,i,unitNum,resNum,serviceNum, rating,dpV,dpU,dpS,dpR);
    delete pDea;
  }

  fout.close();
  delete[] dpR;
} // end of solveAll
```

4.2.2 Example

A car manufacturer wants to evaluate the efficiencies of different garages who have received a franchise to sell its cars. The inputs are: *staff*, *showroom space*, *catchment population* in different economic categories, and *inquiries* for different brands

Table 4.3: Inputs and outputs of franchised garages

Garage	Inputs						Outputs		
	Staff	Space ($100\ m^2$)	Popn. 1 (1000s)	Popn. 2 (1000s)	Model 1 inq. (100s)	Model 2 inq. (100s)	Model 1 sales (1000s)	Model 2 sales (1000s)	Profit (millions)
1	3	3.6	3	3	2.5	1.5	0.8	0.2	0.45
2	6	7.5	10	10	7.5	4	1.5	0.45	0.45
3	7	6	5	7	8.5	4.5	1.2	0.48	2
4	7	8	7	8	3	2	1.9	0.7	0.5
5	14	9	20	25	10	6	2.6	0.86	1.9
6	10	9	10	10	11	5	2.4	1	2
7	3	3.5	3	20	2	1.5	0.9	0.35	0.5
8	12	8	7	10	12	7	4.5	2	2.3
9	5	5	10	10	5	2.5	2	0.65	0.9
10	8	10	30	35	9.5	4.5	2.05	0.75	1.7
11	2	3	40	40	2	1.5	0.8	0.25	0.5
12	5	6.5	9	12	8	4.5	1.8	0.63	1.4
13	7	8	10	12	8.5	4	2	0.6	1.5
14	11	8	8	10	10	6	2.2	0.65	2.2
15	4	5	10	10	7.5	3.5	1.8	0.62	1.6
16	24	15	15	13	25	1.9	8	2.6	4.5
17	30	29	120	80	35	20	7	2.5	8
18	4	6	1	1	7.5	3.5	1.1	0.45	1.7
19	6	5.5	2	2	8	5	1.5	0.55	1.55
20	8	7.5	5	8	9	4	2.1	0.85	2
21	5	5.5	8	10	7	3.5	1.2	0.45	1.3
22	25	16	110	80	27	12	6.5	3.5	5.4
23	19	10	90	22	25	13	5.5	3.1	4.5
24	6	6	20	30	9	4.5	2.3	0.7	1.6
25	6	7	50	40	8.5	3	2.5	0.9	1.6
26	21	12	6	6	15	8	6	0.25	2.9

of car. The outputs are: *number sold* of different brands of car and annual *profit*. Table 4.3 gives the inputs and outputs for each of the 26 franchised garages.

To solve this example, we move data from Table 4.3 to a text file **`garage.txt`** in the directory

`$MIPDIR/examples/mipshell/dea/test`,

then we enter this directory and solve the example with the command

../bin/dea **garage.txt**

Efficiency ratings and other information is written to the output file **garage.sol** which is too long to be presented here. Therefore, Listing 4.6 exibits only its initial part, describing unit 1.

We see that unit 1 with DEA rating of 0.888808 works inefficiently. Five units, 4,7,8,20, and 26, are more efficient than unit 1, and management can suggest changes to improve productivity of unit 1 based on experience of units 4,7,8,20, and 26.

Listing 4.6: Implementation of DEA: function *solveAll*

```
=> Unit 1 of rating 0.888808
 v=(0.0180558,0,0,0.0105713,0.365648,0)
 u=(0.29444,0.0473597,1.43063)
+-------------------------------------+
| Unit | Relative |  Weighted sum of  |
|      |  rating  | inputs  | outputs |
+-------------------------------------+
|    1 | 0.8888   | 1       | 0.8888  |
|    2 | 0.3744   | 2.956   | 1.107   |
|    3 | 0.9785   | 3.308   | 3.237   |
|    4 | 1        | 1.308   | 1.308   |
|    5 | 0.8445   | 4.174   | 3.524   |
|    6 | 0.8391   | 4.308   | 3.615   |
|    7 | 1        | 0.9969  | 0.9969  |
|    8 | 1        | 4.71    | 4.71    |
|    9 | 0.9422   | 2.024   | 1.907   |
|   10 | 0.7701   | 3.988   | 3.071   |
|   11 | 0.8088   | 1.19    | 0.9627  |
|   12 | 0.8155   | 3.142   | 2.563   |
|   13 | 0.8221   | 3.361   | 2.763   |
|   14 | 0.966    | 3.961   | 3.826   |
|   15 | 0.9754   | 2.92    | 2.848   |
|   16 | 0.9181   | 9.712   | 8.916   |
|   17 | 0.9605   | 14.19   | 13.62   |
|   18 | 0.9831   | 2.825   | 2.777   |
|   19 | 0.879    | 3.055   | 2.685   |
|   20 | 1        | 3.52    | 3.52    |
|   21 | 0.8109   | 2.756   | 2.234   |
|   22 | 0.8778   | 11.17   | 9.805   |
|   23 | 0.8443   | 9.717   | 8.204   |
|   24 | 0.8071   | 3.716   | 2.999   |
|   25 | 0.843    | 3.639   | 3.068   |
|   26 | 1        | 5.927   | 5.927   |
```

+----------------------------------+

4.3 Yield management

Yield management is an approach to revenue maximization for service firms that exhibit the following characteristics:

1. *Relatively fixed capacity*. Service firms with substantial investment in facilities (e.g., hotels and airlines) are capacity-constrained (once all the seats on a flight are sold, further demand can be met only by booking passengers on a later flight).

2. *Ability to segment its market* into different customer classes. Developing various price-sensitive classes of service gives firms more flexibility in different seasons of the year.

3. *Perishable inventory*. Revenue from an unsold seat in a plane or from unsold room in a hotel is lost forever.

4. *Reservation* systems are adopted by service firms to sell capacity in advance of use. However, managers are faced with uncertainty of whether to accept an early reservation at a discount price or to wait in hope to sell later seats or rooms to higher-paying customers.

5. *Fluctuating demand*. To sell more seats or rooms and increase revenue, in periods of slow demand managers can lower prices, while in periods of high demands prices are getting higher.

4.3.1 Yield Management In Airline Industry

Now let us turn to setting of a concrete problem. An airline starts selling tickets for flights to a particular destination D days before the departure. The time horizon of D days are divided into T periods of unequal length (for example, a time horizon of $D = 60$ days can be divided into $T = 4$ periods of length 30, 20, 7 and 3 days). It can be used up to s_k planes of type k each costing f_k to hire, $k = 1, \ldots, K$. Each plane of type k has q_{1k} first class seats (referred to as class 1 in the following), q_{2k} business class seats (class 2), and q_{3k} economy class seats (class 3. Up to r_i percent of seats of class i can be transformed into seats of adjacent categories, $i = 1, 2, 3$.

For administrative simplicity, in each period t ($t = 1, \ldots, T$) only O price options can be used, and let c_{tio} denote the price of a seat of class i ($i = 1, 2, 3$) in period t if option o is used.

Demand is uncertain but is affected by price. Let us assume that S scenarios are possible in each period. Forecasts have been made for these demands for each

scenario in each of T periods. The probability of scenario s ($1 \le s \le S$) in period T is p_{ts}, $\sum_{s=1}^{S} p_{ts} = 1$. If scenario s happens in period t, and price option o is used in this period, then the demand for seats of class i will be d_{tsio}.

We have to decide for each of T periods, price levels, how many seats to sell in each class (depending on demand) to maximize expected yield.

4.3.2 Mip Model

To write a deterministic model for this stochastic problem, we need to describe a scenario tree. In this application the scenario tree has $n+1 = \sum_{t=0}^{T} S^t$ nodes. Let us denote by V_t the set of S^t nodes in level t, $t = 0, 1, \ldots, T$. Let us also assume that the root of the scenario tree is indexed by 0, and then $V_0 = \{0\}$.

Each node $j \in V_t$ ($t = 1, \ldots, T$) corresponds to one of the situations that may happen after t periods, and is characterized by a sequence of integers $(s_1, s_2, \ldots, s_t)$, where $s_\tau \in \{1, \ldots, S_\tau\}$ is an index of a scenario for period τ. The situation of node $j \in V_t$ happens with probability $\bar{p}_j \stackrel{\text{def}}{=} \prod_{\tau=1}^{t} p_{\tau,s_\tau}$ and, if price option o is used, the demand for seats of class i is $\bar{d}_{jio} \stackrel{\text{def}}{=} d_{t,s_t,i,o}$, and their price is c_{tio}. Let us define $\bar{c}_{jio} \stackrel{\text{def}}{=} \bar{p}_j c_{tio}$. The parent of node j, denoted by $parent(j)$, is that node in V_{t-1} which is characterized by the sequence $(s_1, s_2, \ldots, s_{t-1})$. Note, that the root node 0 is the parent of all nodes in V_1 (of level 1).

Now we define the variables. Let v_k denote number of planes of type k used. With each node $j \in V \setminus V_T$ we associate the following variables:

x_{jio}: number of seats of class i to be sold in period t using price option o, when situation of node j will happen;

$y_{jio} = 1$ if price option o is used for class i when situation of node j will happen, and $y_{jio} = 0$ otherwise.

Each node $j \in V$ is associated with the variables:

z_{ji}: number of seats of class i to be sold until situation of node j will happen.

Now we can write the following deterministic model:

$$-\sum_{k=1}^{K} f_k v_k + \sum_{j \in V \setminus V_T} \sum_{o=1}^{O} \sum_{i=1}^{3} \bar{c}_{jio} x_{jio} \to \max, \tag{4.4a}$$

$$\sum_{o=1}^{O} y_{jio} = 1, \quad j \in V \setminus V_T,\ i = 1, 2, 3, \tag{4.4b}$$

$$x_{jio} \le \bar{d}_{jio} y_{jio}, \quad j \in V \setminus V_T,\ i = 1, 2, 3,\ o = 1, \ldots, O, \tag{4.4c}$$

$$z_{ji} = z_{parent(j),i} + \sum_{o=1}^{O} x_{parent(j),i,o}\,, \quad j \in V \setminus \{0\},\ i = 1, 2, 3, \tag{4.4d}$$

$$z_{j1} \le \sum_{k=1}^{K} (q_{1k} + \lfloor r_{2k} q_{2k}/100 \rfloor)\, v_k, \quad j \in V_T, \tag{4.4e}$$

$$z_{j2} \le \sum_{k=1}^{K} (q_{2k} + \lfloor (r_{1k} q_{1k} + r_{3k} q_{3k})/100 \rfloor)\, v_k, \quad j \in V_T, \tag{4.4f}$$

$$z_{j3} \le \sum_{k=1}^{K} (q_{3k} + \lfloor r_{2k} q_{2k}/100 \rfloor)\, v_k, \quad j \in V_T, \tag{4.4g}$$

$$z_{j1} + z_{j3} \le \sum_{k=1}^{K} (q_{1k} + q_{3k} + \lfloor r_{2k} q_{2k}/100 \rfloor)\, v_k, \quad j \in V_T, \tag{4.4h}$$

$$z_{j1} + z_{j2} + z_{j3} \le \sum_{k=1}^{K} (q_{1k} + q_{2k} + q_{3k})\, v_k, \quad j \in V_T, \tag{4.4i}$$

$$x_{jio} \in \mathbb{Z}_+, \quad j \in V \setminus V_T,\ i = 1,2,3,\ o = 1,\dots,O, \tag{4.4j}$$

$$y_{jio} \in \{0,1\}, \quad j \in V \setminus V_T,\ i = 1,2,3,\ o = 1,\dots,O, \tag{4.4k}$$

$$z_{ji} \in \mathbb{Z}_+, \quad j \in V,\ i = 1,2,3, \tag{4.4l}$$

$$z_{0i} = 0, \quad i = 1,2,3, \tag{4.4m}$$

$$v_k \in \mathbb{Z}_+,\ v_k \le s_k, \quad k = 1,\dots,K. \tag{4.4n}$$

The equations (4.4b) are to guarantee that in any of T periods only one price option is chosen for each class. The variable upper bounds (4.4c) guarantee that in any period and for any price option, the number of seats sold for each of three classes does not exceed the demand for these seats. The balance equations (4.4d) count the total number of seats for each of the classes that are sold in any of T periods. The inequalities (4.4e)–(4.4i) require that the total number of seats sold be no more that the number of seats in all the planes hired by the airline.

When the problem (4.4) is sold, we know which options to use and how many seats of each class to be sold in period 1 (this is determined by the values of variables x_{0io}). When period 1 is over, we will know the actual number of seats of each class sold in this period, and we will write down a new model for periods $2,\dots,T$ to determine optimal price option and number of seats of each class to be sold in period 2. This procedure is then repeated for periods $t = 3,\dots,T$.

4.3.3 MIPshell Implementation

To implement IP (4.4), we developed a C++ class named **Cyield** which definition is given in Listing 4.7. First, we consider data structures used to represent problem instances.

Listing 4.7: Definition of **Cyild**

```
#include <mipshell.h>

struct sPlane {
   int num; // number of planes of this type
   double cost; // plane cost
   int q1, q2, q3,
// numbers of seats of first busines and economy classes
       r1, r2, r3;
};

struct sNode {
   int ind, parent, period;
   double prob;
   int *demand; // array of options (of size 3*m_iOptNum)
};

class Cyield: public CProblem
{
   int m_iT;
   int m_iPlaneTypeNum;
   sPlane *m_pPlane;
   int m_iOptNum;
   double *m_dpCost;
   int m_iNodeNum, m_iLeafNum;
   sNode *m_pNode;
public:
   Cyield(const char *name);
#ifdef __THREADS_
   Cyield(const Cyield &other, int thread);
   CMIP* clone(const CMIP *pMip, int thread);
#endif
   virtual ~Cyield();

// implementation
   int model();
private:
   void ReadParameters();
   void ReadOptions();
   void ReadTree();
   void ReadData(const char *name);
};
```

Structure **sPlane** describes planes of a particular type. Its members are:

- *num*: number of planes of this type;
- *cost*: cost to hire;
- *q1*, *q2*, *q3*: number of seats of first, business and economy classes;
- *r1*, *r2*, *r3*: percent of seats of first, business and economy classes that can be transformed into seats of adjacent classes.

Structure **sNode** represents scenario tree nodes. Its memmbers are:

- *ind*: node index;
- *parent*: parent node index;
- *period*: node level;
- *prob*: probability that situation described by this node will happen;
- *demand*: array of price options of size **3****m_iOptNum*, where *demand*[**3***o+i] is demand for tickets of type i if price option o is used.

Next we describe all members of **Cyild**:

- *m_iT*: number of periods;
- *m_iPlaneTypeNum*: number of types of planes;
- *m_pPlane*: array of size *m_iPlaneTypeNum*, where *m_pPlane*[k] describes plane of type k;
- *m_iOptNum*: number of options at any period;
- *m_dpCost*: array of size **3****m_iT***m_iOptNum*, where *m_dpCost*[(t***3**+o)**m_iOptNum*+i] is cost of class i ticket if option o is used in period t;
- *m_iNodeNum*, *m_iLeafNum*: number of nodes and leaves in scenario tree;
- *m_pNode*: array of nodes (of size *m_iNodeNum*) of scenario tree.

To create and initialize an object of **Cyild** we call a constructor which is presented in Listing 4.8. This constructor just calls the function ***ReadData*** to read a number of text files from the folder which name is given by the parameter *name*. Implementation of ***ReadData*** is straightforward and is not discussed here.

Listing 4.8: **Cyild**: constructor

```
Cyield::Cyield(const char *name): CProblem(name)
{
  m_pNode=0;
  m_pPlane=0;
  m_dpCost=0;
  ReadData(name);
}
```

Now it is left to present our **MIPshell** model which is displayed in Listing 4.9. First, we introduce a number of macros that translate object members into parameters of IP (4.4). With these macros is almost identical with its origin.

Listing 4.9: **Cyild**: implementation of *model*

```
#define c(j,i,o) (m_pNode[j].prob*m_dpCost[(m_pNode[j].period*3+o)*m+i])
#define p(j) m_pNode[j].prob
#define d(j,i,o) m_pNode[j].demand[o*m+i]
#define parent(j) m_pNode[j].parent
#define s(k) m_pPlane[k].num
#define q1(k) m_pPlane[k].q1
#define q2(k) m_pPlane[k].q2
#define q3(k) m_pPlane[k].q3
#define r1(k) m_pPlane[k].r1
#define r2(k) m_pPlane[k].r2
#define r3(k) m_pPlane[k].r3
#define f(k) m_pPlane[k].cost

int Cyield::model()
{
  int i,j,k,o,t,
      T=m_iT, n=m_iNodeNum,
      n0=m_iNodeNum-m_iLeafNum,
      m=m_iOptNum, I=3,
      K=m_iPlaneTypeNum;

  VAR_VECTOR v("v",INT_GE,K);
  VAR_VECTOR x("x",INT_GE,n0,I,m), y("y",BIN,n0,m), z("z",INT_GE,n,I);

  maximize(
      sum(j in [0,n0), o in [0,m), i in [0,I)) c(j,i,o)*x(j,i,o)
      - sum(k in [0,K)) f(k)*v(k)
  );
```

```
  forall(j in [0,n0))
  sum(o in [0,m)) y(j,o) == 1;

  forall(j in [1,n), i in [0,I), o in [0,m))
    x(parent(j),i,o) <= d(j,i,o)*y(parent(j),o);

  z(0,0) == 0;
  z(0,1) == 0;
  z(0,2) == 0;

  forall(j in [1,n), i in [0,I))
    z(j,i) == z(parent(j),i) + sum(o in [0,m)) x(parent(j),i,o);

  forall(j in [n0,n)) {
    z(j,0) <= sum(k in [0,K)) (q1(k) + (r2(k)*q2(k))/100)*v(k);
    z(j,1) <= sum(k in [0,K)) (q2(k) + (r1(k)*q1(k)+r3(k)*q3(k))/100)*v(k);
    z(j,2) <= sum(k in [0,K)) (q3(k) + (r2(k)*q2(k))/100)*v(k);
    z(j,1) + z(j,3) <= sum(k in [0,K)) (q1(k)+q3(k)+ (r2(k)*q2(k))/100)*v(k);
    z(j,1) + z(j,2) + z(j,3) <= sum(k in [0,K)) (q1(k)+q2(k)+q3(k))*v(k);
  }

  forall(k in [0,K))
    v(k) <= s(k);

  optimize();
  printsol();
  return 0;
} // end of Cyield::model
```

4.3.4 Example

An airline is selling tickets for flights to a particular destination. The flight will depart in three weeks' time. Up to six planes can be hired. Each plane costs $75000 to hire, and has

- 35 first class seats,
- 40 business class seats,
- 60 economy class seats.

Up to 10 % of seats in any one category can be transferred to an adjacent category.

Table 4.4: Price Options

	Class	Option 1	Option 2	Option 3
Period 1	First	$1500	$1250	$1000
	Busines	$1000	$850	$700
	Economy	$500	$400	$300
Period 2	First	$1750	$1500	$1250
	Busines	$1250	$1000	$850
	Economy	$700	$500	$400
Period 3	First	$1800	$1200	$900
	Busines	$800	$850	$600
	Economy	$450	$500	$450

An airline wishes to decide a price for each of these seats. There will be further opportunities to update these prices in after one week and two weeks. Once a customer has purchased a ticket there is no cancellation option.

For administrative simplicity three price level options are possible in each class (one of which must be chosen). These options are given in Table 4.4 for the current period (perion 1) and two future periods.

Demand is uncertain and is affected by ticket prices. Forecasts have been made, and demand levels have been divided into three scenarios for each period. The probabilities of these scenarios in each period are:

Scenario 1	0.1
Scenario 2	0.6
Scenario 3	0.3

The forecast demands are shown in Table 4.5.

To solve this example with our program, we have to write down input data into a number of text files. These files are situated in the folder

```
$MIPDIR/examples/yield/tests/test1
```

and are presented in Listings 4.10–4.12.

Listing 4.10: Yield management example: file **param.txt**

```
3  1
6  75000  35 10  40 10  60 10
```

Table 4.5: Forecast Demands

			Option 1	Option 2	Option 3
Period 1	Scenar. 1	First	25	30	35
		Busines	40	50	70
		Economy	100	120	130
	Scenar. 2	First	40	50	70
		Busines	90	85	90
		Economy	100	105	140
	Scenar. 3	First	90	100	120
		Busines	90	95	95
		Economy	110	105	135
Period 2	Scenar. 1	First	40	50	65
		Busines	85	90	85
		Economy	100	105	125
	Scenar. 2	First	20	80	100
		Busines	100	120	160
		Economy	120	130	180
	Scenar. 3	First	100	110	160
		Busines	40	60	100
		Economy	25	80	120
Period 3	Scenar. 1	First	60	70	80
		Busines	80	100	110
		Economy	100	120	160
	Scenar. 2	First	60	80	120
		Busines	20	80	90
		Economy	100	120	140
	Scenar. 3	First	100	140	160
		Busines	80	90	120
		Economy	120	130	140

From Listing 4.10 we see that we are to solve an instance with the following parameters: $T = 3$, $K = 1$, $s_1 = 6$, $f_1 = \$75000$, $q_{11} = 35$, $r_{11} = 10$, $q_{12} = 40$, $r_{12} = 10$, $q_{13} = 60$, $r_{13} = 10$.

Listing 4.11: Yield management example: file **options.txt**

```
3
 1500 1000 500
 1250  850 400
 1000  700 300

 1750 1250 700
 1500 1000 500
 1250  850 400

 1800  800 450
 1200  850 500
  900  600 450
```

Listing 4.11 shows that three (the number in the first line) options are available in each of three periods. Starting from line 2, each group of three lines presents price options for each of three periods. Any line in a group presents an option for a particular scenario. An option is described by three numbers that are prices of first, business, and economy class tickets respectively. For example, if scenario 3 happens in period 2, then ticket prices are: \$1250 for first class, \$850 for business class, and \$400 for economy class.

Listing 4.12: Yield management example: file **tree.txt**

```
39
 1   0   0.1   25  40 100   30  50 120   35  70 130
 2   0   0.6   40  90 100   50  85 100   70  90 140
 3   0   0.3   90  90 110  100  95 105  120  95 135

 4   1   0.1   40  85 100   50  90 105   65  85 125
 5   1   0.6   20 100 120   80 120 130  100 160 180
 6   1   0.3  100  40  25  110  60  80  160 100 120
 7   2   0.1   40  85 100   50  90 105   65  85 125
 8   2   0.6   20 100 120   80 120 130  100 160 180
 9   2   0.3  100  40  25  110  60  80  160 100 120
10   3   0.1   40  85 100   50  90 105   65  85 125
11   3   0.6   20 100 120   80 120 130  100 160 180
12   3   0.3  100  40  25  110  60  80  160 100 120
```

```
13   4   0.1   60  80 100   70 100 120   80 110 160
14   4   0.6   60  20 100   80  80 120  120  90 140
15   4   0.3  100  80 120  140  90 130  160 120 140
16   5   0.1   60  80 100   70 100 120   80 110 160
17   5   0.6   60  20 100   80  80 120  120  90 140
18   5   0.3  100  80 120  140  90 130  160 120 140
19   6   0.1   60  80 100   70 100 120   80 110 160
20   6   0.6   60  20 100   80  80 120  120  90 140
21   6   0.3  100  80 120  140  90 130  160 120 140
22   7   0.1   60  80 100   70 100 120   80 110 160
23   7   0.6   60  20 100   80  80 120  120  90 140
24   7   0.3  100  80 120  140  90 130  160 120 140
25   8   0.1   60  80 100   70 100 120   80 110 160
26   8   0.6   60  20 100   80  80 120  120  90 140
27   8   0.3  100  80 120  140  90 130  160 120 140
28   9   0.1   60  80 100   70 100 120   80 110 160
29   9   0.6   60  20 100   80  80 120  120  90 140
30   9   0.3  100  80 120  140  90 130  160 120 140
31  10   0.1   60  80 100   70 100 120   80 110 160
32  10   0.6   60  20 100   80  80 120  120  90 140
33  10   0.3  100  80 120  140  90 130  160 120 140
34  11   0.1   60  80 100   70 100 120   80 110 160
35  11   0.6   60  20 100   80  80 120  120  90 140
36  11   0.3  100  80 120  140  90 130  160 120 140
37  12   0.1   60  80 100   70 100 120   80 110 160
38  12   0.6   60  20 100   80  80 120  120  90 140
39  12   0.3  100  80 120  140  90 130  160 120 140
```

Listing 4.12 describes the scenario tree for our example. This tree has 40 nodes indexed from 0 to 39. All the other nodes are split into three levels: 1 (the root node) in level 0, 3 in level 1, 9 in level 2, and 27 in level 3. In **`tree.txt`** the root node is not presented, the levels are separated by blank lines, and any line, starting from the second, describes a node in the following format:

- node index;
- parent node index;
- probability of reaching this node from its parent;
- demand forecast for each of three options (for example, if option 2 is used at node 27, then a business class ticket price is 90).

Having been prepared these three input files, to solve our example, we enter the commands

cd `$MIPDIR/examples/yield/tests/test1`; **`yield`**

As the result we will get a solution to our example written into the text file **`test1.sol`**.

In fact, we are interested to know the values of nine variables: x(**0**,i,o), i,o=**0**,**1**,**2**. Only three of these nine variables can take nonzero values. If we look into **`test1.sol`**, we will see that these three variables are:

x(**0**,**0**,**0**)=25, x(**0**,**1**,**0**)=40, x(**0**,**2**,**0**)=100.

We see that option 1 (indexed by 0 in **MIPshell** model) must be used in period 1, and then 25, 40, and 95 tickets of, respectively, classes 1, 2, and 3, are expected to be sold.

So let us assume that the airline assigned prices \$1500, \$1000, \$500 for, respectively, first, busines, and economy class tickets. One week later, it turned out that the airline had sold 20, 50, and 100 tickets of, respectively, first, busines, and economy class tickets.

Appendix A

Modelling With MIPshell

MIPshell is an environment that facilitates modelling and solving linear and mixed-integer programming problems with the *Mixed-Integer Programming Class Library* (MIPCL).

We write a *mixed-integer program* (MIP) as

$$\begin{aligned} c^T x &\to \max \\ b^1 \le Ax &\le b^2, \\ d^1 \le x &\le d^2, \\ x_i - \text{integer}, &\qquad i \in S \end{aligned}$$

where A is an $m \times n$-matrix; b^1, b^2 are m-dimensional vectors, and c, d^1, d^2 are n-dimensional vectors; x is an n-dimensional vector of variables or unknowns; S is a subset of the set $N = \{1, \ldots, n\}$ of column indices. A MIP is a *linear program* (LP) if there are not integer variables ($S = \emptyset$). If all the variables are integer ($S = N$), then we have an *integer program* (IP).

MIPshell comprises

- collection of C++ classes that represent variables, constraints, sets, vectors, arrays, and etc.;
- library of functions to simplify posing optimization problems;
- preprocessor that translates new modelling operators into fragments of C++ code.

After preprocessing any **MIPshell** program is converted into a C++ program. **MIPshell** is a tool that simplifies using MIPCL. Having been given a mathematical model of an optimization problem, the user can write down it as a **MIPshell** program very quickly. Another advantage of using **MIPshell** is in that we can relatively easily make changes to a model later, even after several weeks or months.

In **MIPshell** there is no separation between modelling statements and C++ statements. Therefore, the user can easily program complex solution algorithms.

Prototypes for all **MIPshell** classes and functions are contained in the **mipshell.h** header file which needs to be included to any **MIPshell** program.

A.1 Problem Definition

Any optimization problem is represented in **MIPshell** as an instance of the base class **CProblem**. We create a problem by providing to the constructor a problem name. Then we use functions of the base classes **CProblem**, **CMIP**, and **CLP**. Usually **MIPshell** programs start with two statements similar to the following ones:

```
CProblem prob("name");
prob.model();
```

A.2 Sets And Indices

In **MIPshell** we use *sets of integers* (**INT_SET**), *reals* (**REAL_SET**), and *indices* (**INDEX_SET**).

Indices are objects of type **CIndex** (**INDEX** is an alias for **CIndex**). Any object of type **CIndex** represents an index as a string of up to 63 characters. The next example demonstrate the most commonly used operations with indices:

```
INDEX i1(1,2,3), i2("Prod4",5), i[2];
cerr << i1 + i2 << endl;
if (i1.include(3)) {
  i1.split(2,i);
  cerr << i[0] << " " << i[1] << endl;
}
```

First we create two indices `i1`, `i2`, and an array of indices, `i`, of size two. Furthermore, `i1` and `i2` are initialized to store, respectively, the strings "1,2,3" and "Prod4,5". By definition `i1+i2` is an index that stores the string "1,2,3,Prod4,5", which is printed to the standard output stream. Each nonempty index-string consists of indices separated by commas. The last three lines of code do the following. If "3" is a subindex of `i1` (which is true), `i1` is split into subindices "1", "2", "3", and the first two subindices are stored in the array `i`. Therefore, the last statement prints the string "1 2".

In the following example, we declare two sets: `I` of integers, `R` of reals, and `N` of indices:

```
INT_SET I("{1,3,5,7}");
REAL_SET R;
INDEX_SET N;
```

After creation, `R` and `N` are empty set, while `I` is initialized to the set $\{1, 3, 5, 7\}$. We add new items to existing sets using the *add* statements:

```
I.add(5); R.add(3.14); N.add("Prod_1");
```

We may compute *intersection* (operator *), *union* (+), and *difference* (-) of any two sets of the same type. Mixing sets of different types is not allowed. If `I1`, `N1`, and `N2` are declared as in the example below, we can not write `N1=I1-N2`. Given a set `S`, an expression `e` **in** `S` takes value **true** if e ∈ S. Next we give some examples showing manipulations with sets:

```
INT_SET I1,I2("{-3,0,1,4,6}"), I3("{1,2,5,6,9}");
INDEX_SET N1, N2("Paris,Berlin,Minsk"), N3("London,Row");
I1=I2*I3; // I1={1,6}
I1+="0,3"; // I1={0,1,3,6}
N1=N2+N3; // N1={Paris,Berlin,Minsk,London,Rom}
```

Input/output operations with sets are shown in the next paragraph.

A.3 Arrays And Vectors

In **MIPshell** dense arrays are called *vectors*, while sparse arrays (with many zero/empty entries which are not stored) are simply called *arrays*.

A.3.1 Vectors

Two types of **MIPshell** *vectors*, **INT_VECTOR** and **REAL_VECTOR**, are specializations of a single template class **CVector** having the constructor:

```
template <typename Tell>
CVector<Tell>::CVector(int size0, int size1=0, int size2=0, int size3=0);
```

Then **INT_VECTOR** and **REAL_VECTOR** are defined as follows:

```
typedef CVector<int> INT_VECTOR;
typedef CVector<double> REAL_VECTOR;
```

Thus, we declare a two-dimensional vector of integers of size 4×5 as follows

```
INT_SET A(4,5);
```

Vector indices are always integers. Furthermore, vectors are indexed from zero! Note also that **MIPshell** does not perform range checking on vector indices.

A.3.2 Arrays

Six types of **MIPshell** arrays are:

INT_ARRAY : array of integers;

REAL_ARRAY : array of reals;

INDEX_ARRAY : array of indices;

INT_SET_ARRAY : array of sets of integers;

REAL_SET_ARRAY : array of sets of reals;

INDEX_SET_ARRAY : array of sets of indices.

The above **MIPshell** arrays are specializations of a single template class **CArray**:

```
typedef CArray<int> INT_ARRAY;
typedef CArray<double> REAL_ARRAY;
typedef CArray<CIndex> INDEX_ARRAY;
typedef CArray<CSet<int> > INT_SET_ARRAY;
typedef CArray<CSet<double> > REAL_SET_ARRAY;
typedef CArray<CSet<CIndex> > INDEX_SET_ARRAY;
```

The maximum dimension of an array is 4!

Let us remember that **MIPshell** arrays are used to represent sparse arrays with many zero (or empty) elements which are not stored in the computer memory. By convention, A(3,"Prod_10",8) returns a reference to a constant zero (or empty) element if an entry indexed by (3,Prod_10,8) has not been previously added to A. For this reason, we cannot assign values to array entries and instead have to use the *add* statements. In the following fragment we declare an array of reals, add a number of entries, and then print some array values.

```
REAL_ARRAY R;
A(1).add(5.3);
A(2,"unit_4").add(-1.1);
A(0,1,"x").add(5.3);
cout << A(2,"unit_4") << endl; // prints -1.1
cout << A(1,1) << endl; // prints 0
```

When modelling with **MIPshell**, we can use any C++ structures, including arrays, which are one-dimensional (excluding static arrays). Representing multi-dimensional arrays by one-dimensional ones may obscure a model with unnecessary details. Therefore, we recommend using multi-dimensional **MIPshell** vectors and arrays. Besides, using **MIPshell** vectors and arrays is more safe since in this case allocating and freeing memory is done by **MIPshell**.

A.3.3 Input/Output

The next example program illustrates input/output operations with vectors, arrays, and sets.

```
#include <mipshell.h>
#include <iostream>
using namespace std
```

```
int main() {
  REAL_VECTOR A;
  INT_SET S;
  REAL_ARRAY R;
  cerr << "Enter a real vector: ";
  cin >> A;
  cerr << "Enter a set of integers: ";
  cin >> S; S-={2};
  cerr << "Enter an array of reals: ";
  cin >> R;
  cout << A << endl;
  cout << S << endl;
  cout << R << endl;
}
```

When we compile and run this program, if we type in first

dim(2,2): [[2,1.5],[3.1,4]]

then

{1,2,3,5,7}

and then

{(0,3;3.14),(1,2;2.0),(3,1;-1.1)}

as input, the output is

```
dim(2,2):
[[2,1.5],
[3.1,4]]
{1,3,5,7}
{(0,3;3.14),(1,2;2.0),(3,1;-1.1)}
```

A.4 Variables

In **MIPshell**, variables are instances of the **CVar** class. The most commonly used constructor of **CVar** is defined as

CVar(**const char** **name*, **int** *type*=*REAL_GE*);

The first argument is the variable *name* of length up to 63 characters. The name is used when a solution is printed. The second argument defines the *type* and, for integer and binary variable, the priority of the variable. Variables are of one of the following types:

{ *REAL*,*REAL_GE*,*REAL_LE*, **INT**,*INT_GE*,*INT_LE*,*BIN*};

Depending on the type, the *domain* of a variable is as in the following table:

REAL	*REAL_GE*	*REAL_LE*	*INT*	*INT_GE*	*INT_LE*	*BIN*
$\mathbf{R}$	$\mathbf{R}_+$	$\mathbf{R}_-$	$\mathbf{Z}$	$\mathbf{Z}_+$	$\mathbf{Z}_-$	$\{0,1\}$

Here $\mathbf{R}$ (resp. $\mathbf{Z}$) is the set of reals (resp. integers), and

$$\begin{aligned} \mathbf{R}_+ &= \{\alpha \in \mathbf{R} : \ \alpha \geq 0\}, \\ \mathbf{R}_- &= \{\alpha \in \mathbf{R} : \ \alpha \leq 0\}, \\ \mathbf{Z}_+ &= \{\alpha \in \mathbf{Z} : \ \alpha \geq 0\}, \\ \mathbf{Z}_- &= \{\alpha \in \mathbf{Z} : \ \alpha \leq 0\}. \end{aligned}$$

Variables of types *REAL*, *REAL_GE*, and *REAL_LE* are called *real* or *continuous*. Variables of types **int**, *INT_GE*, *INT_LE*, *BIN* are *integer variables*. A *binary variable* of type *BIN* is also an integer variable.

The *priority* of a variable is a nonnegative integer (of length at most 16 bit) that is used when a integer variable with fractional value is selected for branching; the higher priority the higher probability for variable to be selected. The *composed type* of a variable is the sum of its type and priority. For example, we declare a nonnegative integer variable *x* named X and having priority of value 2 as follows:

```
VAR x("X",INT_GE+2);
```

Here we used an alias for **CVar** defined by

```
typedef CVar VAR.
```

A.4.1 Arrays Of Variables

MIPshell provides a specific class, **CVarVector**, for representing (*multidimensional*) *vectors of variables*. Normally we define a vector of variables using the constructor

```
CVarVector(const char *name, int type,
           int size0, int size1=0, int size2=0, int size3=0);
```

The meaning of the first two parameters is the same as for the constructor of **CVar**, the last four parameters define the size of the vector. In **MIPshell** in place of **CVarVector** we can also use the alias name **VAR_VECTOR**. For instance, the declaration

```
VAR_VECTOR x("X",BIN+8,5,4);
```

means that *x* is a (5×4)-vector of binary variables *x*(i,j) each of priority 8, *x*(i,j) is named $X(i,j)$. Let us remember that variable names are strings of up to 63 character. Because of this restriction, if the full name (name itself plus indices) of a vector member has more then 63 characters, the tail string starting from character 64 is truncated.

Vectors of variables in **MIPshell** are indexed from zero!

In **MIPshell** for indexing variables, we can also use sets of indices or integers. To declare an *array of variables*, we use one of the constructors:

```
CVarArray(const char *name, int type, CBasicSet& s0),
CVarArray(const char* name, int type,
          CBasicSet& s0, CBasicSet& s1),
CVarArray(const char *name, int type,
          CBasicSet& s0, CBasicSet& s1, CBasicSet& s2),
CVarArray(const char *name, int type,
          CBasicSet& s0, CBasicSet& s1,
          CBasicSet& s2, CBasicSet& s3).
```

The maximum dimension of an array of variables is 4. The first two parameters are identical to those in the **CVar** constructor. The **CBasicSet** class is the virtual base class for the **CSet** classes. Therefore, the **INT_SET** and **INDEX_SET** objects can be used when declaring variable arrays. For example, the following declarations are legal in **MIPshell**:

```
INT_SET I("{1,3,5,7}");
INDEX_SET Prod("{Gasoline,Oil,JetFuel}");
VAR_ARRAY x("X",BIN,I,Prod);
```

When a vector or array of variables is created, all its elements (variables) are of the same type, and have the same domain. We can change the type of a particular variable, say x(0,0), by calling either

```
x(0,0).SetType(INT_GE);
```

or

```
settype(x(0,0),INT_GE);
```

A.5 Constraints

In **MIPshell** constraints are instances of the **CCtr** class. Usually, we do not declare objects of type **CCtr**. We simply write down constraints and leave the rest to **MIPshell**. Two examples of valid **MIPshell** constraints are

```
0 <= x(i,2) - 4.5*z + 2*y(4) <= 5;
sum(i in S: i < j) a(i,j)*x(i,j) == b(i);
```

The **sum** operator is discussed in section A.6. Here it is enough to say that the latter **MIPshell** constraint in the mathematical notation is written as follows:

$$\sum_{i \in S,\, i<j} a(i,j) \cdot x(i,j) = b(i).$$

Each **MIPshell** constraint is translated into one MIPCL constraint. Therefore, two-sided constraints like the following one are not allowed:

```
x1 <= x2 - x3 <= x4 + 5;
```

Here *x1*,*x2*,*x3*, and *x4* are **MIPshell** variables. This constraint should be rewritten, for example, as follows:

```
x1 <= x2 - x3;
x2 - x3 <= x4 + 5;
```

When we are solving an LP and are interested in having an optimal dual solution, in particular, shadow prices for constraints, we need to assign names to the constraints. The following examples show how it can be done:

```
(x - 2*y <= z).SetName("name");
(sum(int i in [0,n)) x(i,j) == 1).SetName("assign",j);
```

In the latter example the constraint is given the name "assign(j)", where j is an integer. When naming constraints we can use up to three indices. The maximum total name length (with all indices) is 31. Be careful, statements like the following one

```
(x <= 1.0).SetName("name");
```

are not correct. This is because the above statement is used to change the upper bound of *x*, and **MIPshell** does not create any constraint. Thus, there is no object to assign the name.

A.5.1 Discrete Variables

A discrete variable is a real variable restricted to take values from a given set of reals. We declare discrete variables in one of the two following ways:

```
x in "{2,5,9}"; // x ∈ {2, 5, 9}
y in S; // y ∈ S
```

Here S is a set of reals (of type **REAL_SET**). Be careful, in the latter case any changes applied to S until the problem is loaded (see the description of the ***load*** function in Section A.6) will also affect the domain of *y*.

A.5.2 Piecewise-Linear Functions

Let a real variable y be a function of another real variable x. We may approximate this function by a piecewise-linear function and represent the latter in **MIPshell** as follows:

```
y1 == function(x1,"(0,0),(1,1),(2.4),(3,9),(4,16),(5,25)");
y2 == function(x2,A);
```

In the first example *y1* $\approx$ *x1*^{2}, *x1*$\in$ $[0,5]$; in the second `A` is a real (of type **REAL_VECTOR**) vector of size $k \times 2$, *y2* is a piecewise-linear function of *x2*, and

$$(A(0,0),A(0,1)),\ldots,(A(k,0),A(k,1))$$

are its breakpoints.

Similarly, we may define a set of points (x,y) lying on a piecewise-linear curve:

curve(*x1*,*y1*,"(0,0),(1,1),(2.4),(3,9),(4,16),(5,25)");
curve(*x2*,*y2*,`A`);

The difference between two constructs, **function** and **curve**, is in that the pairs of points in the declaration of a **function** must be listed in the increasing order of the x-values.

A.6 MIPshell Functions And Operators

Most of the **MIPshell** functionality is due to operator overloading. Besides there are a few functions and operators that are briefly discussed in this section:

- **void** ***load***(): loads the problem (builds the matrix, does preprocessing, performs scaling, and etc.).
- **void** ***solve***(): solves the problem (the problem must be previously loaded).
- **void** ***optimize***(): loads and solves the problem (***load***() + ***solve***()).
- **void** ***printsol***(**const char** **fileName*=**0**): prints the solution to the file which name is given by the string *fileName*. If *fileName*=**0**, then the output file name is the problem name appended with the extension ".sol".
- **double** ***getobj***(): returns the objective value of the optimal solution.
- **double** ***getval***(**CVar**& *var*): returns the value of the *var* variable.
- **double** ***getredcost***(**CVar**& *var*): returns the reduced cost of the variable *var* (only for LPs).
- **double** ***getprice***(**CCtr**& *ctr*): returns the shadow price of the constraint *ctr* (only for LPs).
- **void** ***preprocoff***(): switches off preprocessing. Normally, preprocessing is switched off when a dual optimal solution is needed.
- **void** ***setcutdepth***(**int** *depth*): cuts will be generated for nodes of depth less than *deph*. If *depth*=**0**, cuts will be generated only for the root LP. To switch off generating cuts, call **setcutdepth** with *depth* set to -1.

- **void** *settype*(**CVar**& *var*, **VARTYPE** *type*): sets the type of the variable var to the value of *type*.

The **forall** and **sum** operators are inherent to practically any optimization modelling language. In **MIPshell** these operators are of the form

forall(i1 **in** S1,...,ik **in** Sk: condition) *operator*,
sum(i1 **in** S1,...,ik **in** Sk: condition) *linear expression*,

where i1,. . .,ik are integers, S1,. . .,Sk are sets of integers, and *operator* is any C++ operator, including the new **MIPshell** operators. A *linear expression* is a multiplicative expresion that is linear with respect to **MIPshell** variables.

In the next code fragment both operators, **forall**, and **sum**, are used:

```
forall(int i in [0,m)) {
  sum(int j in [0,n)) a(i,j)*x(i,j) == 1;
  x(i,j) <= b(i);
}
```

For integers k and l, $[k,l)$ (resp. $[k,l]$) denotes the set $\{k,\ldots,l-1\}$ (resp. $\{k,\ldots,l\}$).

Besides those mentioned above, there are also a few functions which are used when developing applications with user defined cuts. In Chapter **??**, we give an example showing how to write such applications.

A.7 Our First MIPshell Applications

A.7.1 Product Mix

An engineering factory can produce five products by using two production processes: grinding and drilling. Each unit of each product yields the following contribution to profit:

Prod 1	Prod 2	Prod 3	Prod 4	Prod 5
$550	$600	$350	$400	$200

Processing time (in hours) per product units are given below. A dash indicates that a process is not needed.

	Prod 1	Prod 2	Prod 3	Prod 4	Prod 5
Grinding	12	20	—	25	15
Drilling	10	8	16	—	—

In addition the final assembly of each unit of each product requires 20 hours of an employee's time.

The factory has three grinding and two drilling machines, and works a six-day week with two shifts of 8 hours on each day. Eight workers are employed in assembly, each working one shift a day.

The problem is to decide how many units of each product to produce so as to maximize the total factory profit.

To formulate a *mathematical model*, we introduce variables $x_1, x_2, \ldots, x_5$ where x_i is the number of units of product i to be produced in a week. The complete model is as below:

$$550x_1 + 600x_2 + 350x_3 + 400x_4 + 200x_5 \to \max, \tag{A.1a}$$
$$12x_1 + 20x_2 + 25x_4 + 15x_5 \leq 288, \tag{A.1b}$$
$$10x_1 + 8x_2 + 16x_3 \leq 192, \tag{A.1c}$$
$$20x_1 + 20x_2 + 20x_3 + 20x_4 + 20x_5 \leq 384, \tag{A.1d}$$
$$x_1, x_2, x_3, x_4, x_5 \in \mathbf{Z}_+. \tag{A.1e}$$

Given that we have three grinding machines working for a total of 96 hours a week each, we have 288 hours of grinding capacity available. The total amount of grinding capacity that we use in a week is given by the expression on the left hand side of (A.1b).

Similarly, constraint (A.1c) says that we cannot exceed the drilling capacity in 192 hours a week.

The fact that we have only eight workers each working 48 hours a week gives us a labor capacity of 384 hours. Since each unit of each product uses 20 hours of this capacity we have the constraint (A.1d).

Finally, the constraint (A.1e) simply says that all five variable are nonnegative integers.

MIPshell-implementation

The easiest way of using **MIPshell** is to run a shell script `mipinit` to build a skeleton **MIPshell**-application with an empty model, and then extend this skeleton application.

So, first we open a consol window and enter our working directory. Then we use the command

```
mipinit prodmix
```

to create a new directory called `prodmix` and populate it with a number of subdirectories and files. In particular, there are four files in the directory **`sources`**:

`main.cpp`, **`prodmix.h`**, **`prodmix.cpp`**, and **`prodmix.mod`**.

To solve our simple example, we need to modify only one of these files, **`prodmix.mod`**. Its contents is as in Listing A.1.

Listing A.1: **MIPshell**-template

```
int Cprodmix::model()
{
// TODO: write your model here
   optimize();
   printsol();
} // end of Cpromix::model
```

In place of the line "// TODO: write your model here", we need to write our code. A **MIPshell**-implementation of IP (A.1) is given in Listing A.2.

Listing A.2: **MIPshell**-implementation of (A.1)

```
int Cprodmix::model()
{
   VAR_VECTOR x("x",INT_GE,5);
   maximize(550*x(0) + 600*x(1) + 350*x(2) + 400*x(3) + 200*x(4));
   12*x(0) + 20*x(1) + 25*x(3) +15*x(4) <= 288;
   10*x(0) + 8*x(1) + 16*x(2) <= 192;
   20*x(0) + 20*x(1) + 20*x(2) + 20*x(3) + 20*x(4) <= 384;
   optimize();
   printsol("prodmix.sol");
   return 0;
} // end of Cpromix::model
```

It should be noted that a variable x_i in the mathematical model corresponds to the **MIPshell** variable $x(i-1)$. Similar mappings between mathematical and **MIPshell** variables are implicitly assumed in many other examples of this guide.

We build an executable with the commands

```
cd /prodmix/release
make all
```

and then run the executable to solve our instance

```
cd ../bin
promix
```

If the output file **prodmix.sol** looks like the one given below, then you are done; otherwise, check once again you model and correct it.

```
Objective Value = 10800
Variables
                    Name              Value
```

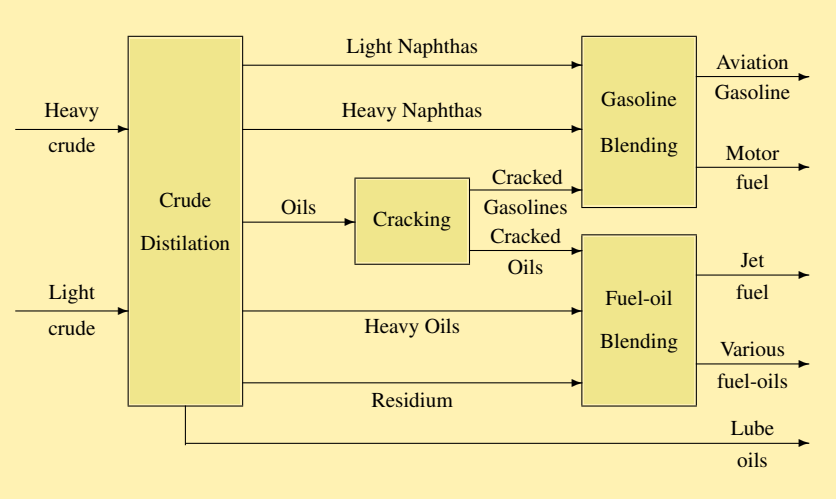

Figure A.1: Simplified refinery example

```
x(4)          0.000000
x(3)          0.000000
x(2)          0.000000
x(1)          7.000000
x(0)         12.000000
```

A.7.2 Oil Refineries

The diagram in Figure A.1 shows a simplified model for planning a day's production of the process of refining crude oil to gasolines, fuels, fuel-oils and lube-oils.

Looking through the model in Listing A.3 we find that there is no restriction on the availability of heavy crudes, while for light crudes we have a restriction of 10 units per day, one unit being 1000 barrels. The distillation column can process a maximum of 25 units per day and the cracking unit can process at most 8 units per day. Crude distillation gives light and heavy naphthas.

Listing A.3: **MIPshell** model for refinery example

```
int Crefining::model()
{
  VAR HeavyCrude("Heavy Crude",REAL_GE),
      LightCrude("Light Crude",REAL_GE),
      LightNaphthas("Light Naphthas",REAL_GE),
      HeavyNaphthas("Heavy Naphthas",REAL_GE),
      Oils("Oils",REAL_GE),
      HeavyOils("Heavy Oils",REAL_GE),
      Residium("Residium",REAL_GE),
      CrackedGasolines("Cracked Gasolines",REAL_GE),
      CrackedOils("Cracked Oils",REAL_GE),
      AviationGasoline("Aviation Gasoline",REAL_GE),
```

```
        MotorFuel("Motor Fuel",REAL_GE),
        JetFuel("Jet Fuel",REAL_GE),
        FuelOils("Fuel Oils",REAL_GE),
        LubeOils("Lube Oils",REAL_GE);

  maximize(6.5*AviationGasoline + 4.6*MotorFuel +
           3.5*JetFuel + 2.5*FuelOils + 0.8*LubeOils
           - 1.5*HeavyCrude - 1.7*LightCrude
           - 0.4*LightNaphthas - 0.4*HeavyNaphthas - 0.9*Oils
           - 0.3*HeavyOils - 0.3*Residium
           - 0.4*CrackedGasolines - 0.3*CrackedOils);

  LightCrude <= 10; // Light crude avail.
  (HeavyCrude + LightCrude <= 25).SetName("Dist. capacity");
  Oils <= 8; // Cracking capacity
  (0.12*HeavyCrude + 0.17*LightCrude ==
    LightNaphtas).SetName("Dist- >LightNaphthas");
  (0.23*HeavyCrude + 0.28*LightCrude ==
    HeavyNaphtas).SetName("Dist- >HevyNaphthas");
  (0.41*HeavyCrude + 0.34*LightCrude == Oils + HeavyOils).SetName("Dist->Oils");
  (0.24*HeavyCrude + 0.21*LightCrude ==
    Residium + LubeOils).SetName("Dist->Residium");
  (CrackedGasolines == 0.65*Oils).SetName("Cracking->Gasoline");
  (CrackedOils == 0.35*Oils).SetName("Cracking->Oils");
  (AviationGasoline + MotorFuel ==
    LightNaphtas + HeavyNaphtas + CrackedGasolines).SetName("Gasoline blending");
  JetFuel + FuelOils <=
    (HeavyOils + Residium + CrackedOils).SetName("Fuel-oil blending");
  (AviationGasoline <= 1.5*LightNaphthas).SetName("Quality");
  (AviationGasoline <= 1.2*LightNaphthas + 0.3*HeavyNaphthas).SetName("Octane number");
  (AviationGasoline <= 0.5*MotorFuel).SetName("Sales limit (av.gas)");
  JetFuel <= 4; // Sales limit
  preprocoff();
  optimize();
  printsol("refining.sol");
  return 0;
}
```

The output of the distillation is in proportion to the inputs and consists of four products as indicated in the diagram. For instance, from the model we see that one unit of light crude input "transforms" into 0.17 units of light naphtha, while the yield from one unit of heavy crude is 0.12 units of light naphtha. From an engineering perspective it is realistic to assume that the technology is "linear" and thus

$$LightNaphthas = 0.12\, HeavyCrude + 0.17\, LightCrude.$$

Likewise one interprets the yield equation for the other three intermediate products.

Note also that in the simplified model we assume no "loss", i.e. every unit of heavy crude is split without loss into four intermediate products. Cracking oils results in cracked gasoline and cracked oils in the proportion as indicated. "Blending" the various inputs results in marketable products, aviation gasoline, motor fuel, jet fuel, various fuel-oils, lube oils.

In addition to the constraints that we have mentioned so far there are quality constraints and sales limitations. For instance, the constraint

$$AviationGasoline \leq 1.5\, LightNaphthas$$

means that one unit of aviation gas requires at least $\frac{2}{3}$ units of light naphtha to assure the necessary quality as measured e.g. by the gas' *volatility*.

The objective is to maximize the sum of the profits of selling output products minus the sum of the costs of buying the input products and producing the intermediates. The profits and costs coefficients are given in dollars per unit.

An optimal solution to the refinery example produced by the MIPCL solver is presented in Listing A.4.

Listing A.4: Solution to the refinery example

```
Objective Value = 51.6683 Variables
                     Name              Value      Reduced Cost
                 Lube Oils           0.000000         -1.400000
                 Fuel Oils           6.050000          0.000000
                  Jet Fuel           4.000000          1.000000
                Motor Fuel           9.966667          0.000000
         Aviation Gasoline           4.983333          0.000000
              Cracked Oils           2.800000          0.000000
         Cracked Gasolines           5.200000          0.000000
                  Residium           5.700000          0.000000
                Heavy Oils           1.550000          0.000000
                      Oils           8.000000          0.811667
             Heavy Naphthas          6.250000          0.000000
             Light Naphthas          3.500000          0.000000
                Light Crude         10.000000          0.063333
                Heavy Crude         15.000000          0.000000
Constraints
                     Name        Shadow price
     Sales limit (av.gas)            1.266667
            Octane number            0.000000
                  Quality            0.000000
        Fuel-oil blending            2.500000
        Gasoline blending            5.233333
           Cracking->Oils            2.200000
       Cracking->Gasoline            4.833333
           Dist->Residium            2.200000
               Dist->Oils            2.200000
       Dist->HevyNaphthas           -4.833333
      Dist->LightNaphthas           -4.833333
           Dist. capacity            1.621667
```

A.8 Fixed Charge Network Flows

In the previous section we are solving particular MIP instances. But usually we need to develop computer programs that solve problems which are subclasses of

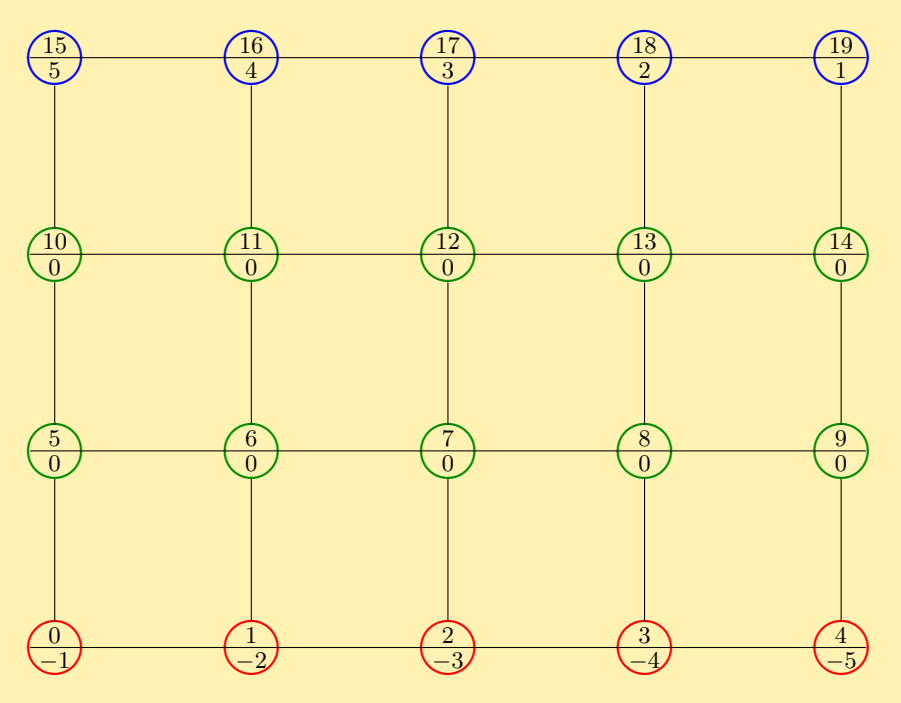

Figure A.2: An instance of FCNF

MIP. In this section we describe a **MIPshell** implementation of the fixed charge network flow problem.

A transportation network is given by a *directed graph* (*digraph*) $G = (V, E)$. For each node $v \in V$, we know the *demand* d_v in some product. If $d_v > 0$, then v is a demand node; if $d_v < 0$, then v is a supply node; $d_v = 0$ for transit nodes. It is assumed that supply and demand are balanced: $\sum_{v \in V} d_v = 0$. The capacity of an arc $e \in E$ is $u_e > 0$, and the cost of shipping $x_e > 0$ units of product along this arc is $f_e + c_e x_e$. Naturally, if the product is not moved through the arc ($x_e = 0$), then nothing is paid. The *fixed charge network flow problem* (FCNF) is to decide on how to transport the product from supply nodes to demand nodes so that the transportation expenses are minimum.

The FCNF problem appears as a subproblem in many practical applications such as design of transportation and telecommunication networks, production planning problems.

An instance of FCNF problem is given in Figure A.2, where the node numbers and demands are depicted in the upper and lower segments of the node circles. Any (undirected) edge $e = (i, j)$ represents two (directed) arcs $e_1 = (i, j)$ and $e_2 = (j, i)$. All horizontal arcs have capacity $u_e = 3$, fixed cost $f_e = 1$ and per-unit cost $c_e = 0$, and all vertical arcs have capacity $u_e = 4$, fixed cost $f_e = 2$ and per-unit cost $c_e = 0$.

Introducing variables

x_e: *flow* (quantity of shipping product) through arc $e \in E$,

$y_e = 1$, if product is shipped ($x_e > 0$) through arc e, and $y_e = 0$ otherwise,

we formulate the FCNF problem as follows:

$$\sum_{e \in E} (f_e y_e + c_e x_e) \to \min \tag{A.2a}$$

$$\sum_{e \in E(V,v)} x_e - \sum_{e \in E(v,V)} x_e = d_v, \quad v \in V, \tag{A.2b}$$

$$0 \le x_e \le u_e y_e, \quad e \in E, \tag{A.2c}$$

$$y_e \in \{0, 1\}, \quad e \in E. \tag{A.2d}$$

Here the objective (A.2a) minimizes transportation expenses. The *balance equations* (A.2b) require that the number of flow units entering each particular node equals the number of flow units leaving the node. The *variable upper bounds* (A.2c) are *capacity restrictions* with the following meaning:

- the flow value through any arc cannot exceed the capacity of the arc;
- if some arc is not used for shipping product ($y_e = 0$), then the flow value through this arc is zero ($x_e = 0$).

A.8.1 MIPshell implementation

A straightforward MIPshell implementation of IP (A.2a)–(A.2d) is given in Listings A.5–A.10.

The function ***main*** from Listing A.5 creates an object named `prob` of class **Cfcnf** for an FCNF instance from a file passed to ***main*** as its only parameter. Then ***main*** calls the function ***model*** (of the base class **CProblem**) to build and then solve a model for the FCNF instance stored in `prob`.

Listing A.5: Fixed Charge Network Flows: ***main*** function

```
#include <iostream>
#include "fcnf.h"

int main(int argc, const char *argv[])
{
  try {
    Cfcnf prob(argv[1]);
    prob.model();
  }
  catch(CException *pe) {
```

```
    std::cerr << pe->GetErrorMessage() << std::endl;
    delete pe;
    return 1;
  }
  return 0;
}
```

Class **Cfcnf** is defined in Listing A.6. Its members are:

- *m_iVertNum*: number of nodes;
- *m_iEdgeNum*: number of arcs;
- *m_ipTail*[e], *Head*[e]: tail and head of arc e=(*m_ipTail*[e],*m_ipHead*[e]);
- *m_ipDemand*[v]: demand for flow at node v;
- *m_ipCapacity*[e], *m_ipCost*[e], *m_ipFixedCost*[e]: respectively, capacity, cost and fixed cost of arc e.

Listing A.6: Fixed Charge Network Flows: file **fcnf.h**,

```
#include <mipshell.h>

class Cfcnf: public CProblem
{
  int m_iVertNum, m_iEdgeNum,*m_ipTail,*m_ipHead,
      *m_ipCapacity,*m_ipFixedCost,*m_ipCost,*m_ipDemand;
public:
  Cfcnf(const char *fileName);
#ifdef __THREADS_
  Cfcnf(const Cfcnf &other, int thread);
  CMIP* clone(const CMIP *pMip, int thread);
#endif
  virtual ~Cfcnf();
/////////////////////////// implementation
  int model();
  void PrintSolution(VAR_VECTOR &flow);
private:
  void ReadNet(const char *fileName);
};
```

Implementation of constructors and the destructor of **Cfcnf** is given in Listing A.7. The constructor

Cfcnf::***Cfcnf***(**const char** **fileName*)

calls ***ReadNet***(*fileName*) (see Listing A.8) to read the description of an FCNF instance from the file which name is passed as the only input parameter *fileName*.

Implementation of the clone constructor

Cfcnf::***Cfcnf***(**const Cfcnf** &*other*, **int** *thread*)

and the clone-function

CMIP* **Cfcnf**::***clone***(**const CMIP** **pMip*, **int** *thread*)

is standard for all **MIPCL**-applications that do not overload any function of the base classes **CMIP** or **CLP**.

The destructor

Cfcnf::∼***Cfcnf***()

frees the memory allocated in ***ReadNet*** for the arrays describing the network. Since these arrays are used only by the root thread to pose a **MIPshell**-model and then to write an optimal solution to a file, we do not need to share these arrays with the other threads.

Listing A.7: Fixed Charge Network Flows: file **`fcnf.cpp`** (Part 1)

```
#include "fcnf.h"
#include <fstream>
#include <cstring>
// Cfcnf::Cfcnf(const char *fileName): CProblem("fcnf")
{
  m_ipTail=m_ipHead=m_ipCapacity=
  m_ipFixedCost=m_ipCost=m_ipDemand=0;
  ReadNet(fileName);
}

#ifdef __THREADS_
Cfcnf::Cfcnf(const Cfcnf &other, int thread): CProblem(other,thread)
{
} // Cfcnf::Cfcnf(const char *fileName)

CMIP* Cfcnf::clone(const CMIP *pMip, int thread)
{
  return static_cast<CMIP*>(new Cfcnf(*static_cast<Cfcnf*>(
           const_cast<CMIP*>(pMip)),thread));
}
#endif
```

```
Cfcnf::~Cfcnf()
{
#ifdef __THREADS_
  if (!GetParent()) {
#endif
    if (m_ipTail)
      delete[] m_ipTail;
#ifdef __THREADS_
  }
#endif
}
//////////////////////////
#include "fcnf.xyz"
```

The function *ReadNet* from Listing A.8 reads an instance of FCNF from a text file of the following format: Line 1 contains 2 integer numbers n and m (number of nodes and number of edges); staring from Line 2, node demands are written as n integer numbers, the i-th number is the demand at node i; demands are followed by m lines with arc parameters, the i-th of these lines contains 5 numbers, respectively, the head, tail, capacity, fixed cost, and cost of arc i. Our test example of Figure A.2 is described in the file "**`grid.txt`**" which can be found in the folder

$MIPDIR/examples/mipshell/fcfn/tests

reserved for our FCFN application.

Listing A.8: Fixed Charge Network Flows: file **`fcnf.cpp`** (Part 2),

```
void Cfcnf::ReadNet(const char *fileName)
{
  int n,m;
  std::ifstream fin(fileName);
  if (!fin.is_open()) {
    throw CFileException("ReadNet",fileName);
  }
  fin >> n >> m;
  m_iVertNum=n; m_iEdgeNum=m;
  m_ipTail = new(std::nothrow) int[5*m+n];
  if (!m_ipTail) {
    throw CMemoryException("Cfcnf::ReadNet");
  }
  m_ipDemand=(m_ipCost=(m_ipFixedCost= (m_ipCapacity=
        (m_ipHead = m_ipTail+m)+m)+m)+m)+m;
  for (int i=0; i < n; ++i) {
```

```
    fin >> m_ipDemand[i];
  }
  for (int i=0; i < m; ++i) {
    fin >> m_ipTail[i] >> m_ipHead[i] >> m_ipCapacity[i]
        >> m_ipFixedCost[i] ¿¿ m_ipCost[i];
  }
  fin.close();
} // end of Cfcnf::ReadNet
```

The standard **MIPshell**-function ***printsol*** writes to an output file the values of all the problem variables. In most cases we can write down solutions in a more readable form. The function ***PrintSolution*** from Listing A.9 writes down only nonzero arc flows.

Listing A.9: Fixed Charge Network Flows: file **`fcnf.cpp`** (Part 3)

```
void Cfcnf::PrintSolution(VAR_VECTOR &flow)
{
  int m=m_iEdgeNum;
  char FileName[128];
  GetProblemName(FileName);
  std::strcat(FileName,".sol");
  std::ofstream fout(FileName);
  if (IsSolution()) {
    fout << "Nonzero flows:\n";
    for (int e=0; e < m; ++e) {
      if (getval(flow(e)) > 0.5) {
        fout << "flow(" << m_ipTail[e] << ","
             << m_ipHead[e] << ")="
             << getval(flow(e)) << std::endl;
      }
    }
  }
  else {
    fout << "Problem has no solution!\n";
  }
  fout.close();
} // end of Cfcnf::PrintSolution
//////////////////////////
//
```

The function

```
int Cfcnf::model()
```

translates IP model (A.2) into a **MIPshell** model. For this translation be more transparent, we introduced a number of macroes that map the parameters of the MIP model into the program parameters.

After the **MIPshell** model has been built, we call first ***optimize*** to solve the problem, and then ***PrintSolution*** to write an optimal solution to a file.

MIPshell translates the model stored in **`fcnf.map`** into a C++ file **`fcnf.xyz`** which is included into **`fcnf.cpp`**.

Listing A.10: Fixed Charge Network Flows: file **`fcnf.cpp`**

```
// macros for improving readability
#define t(e) m_ipTail[e]
#define h(e) m_ipHead[e]
#define c(e) m_ipCost[e]
#define f(e) m_ipFixedCost[e]
#define u(e) m_ipCapacity[e]
#define d(v) m_ipDemand[v]

int Cfcnf::model()
{
  int v,e, n=m_iVertNum, m=m_iEdgeNum;
  VAR_VECTOR x("x",REAL_GE,m);
  VAR_VECTOR y("y",BIN,m);

  minimize(sum(e in [0,m)) (f(e)*y(e) + c(e)*x(e)));
  forall(v in [0,n))
    sum(e in [0,m): h(e)==v) x(e) - sum(e in [0,m): t(e)==v) x(e) == d(v);

  forall(e in [0,m))
    x(e) <= u(e)*y(e);

  optimize();
  PrintSolution(x);
}
// end of Cfcnf::model
```

Our test example of Figure A.2 is described in the file "**`grid.txt`**" which can be found in the folder

$MIPDIR/examples/mipcl/fcfn

reserved for our FCFN application.

After compiling and then running our test program with input parameter set to "**`grid.txt`**", we get the answer written into the file named "**`grid.txt.sol`**", and depicted in Figure A.3.

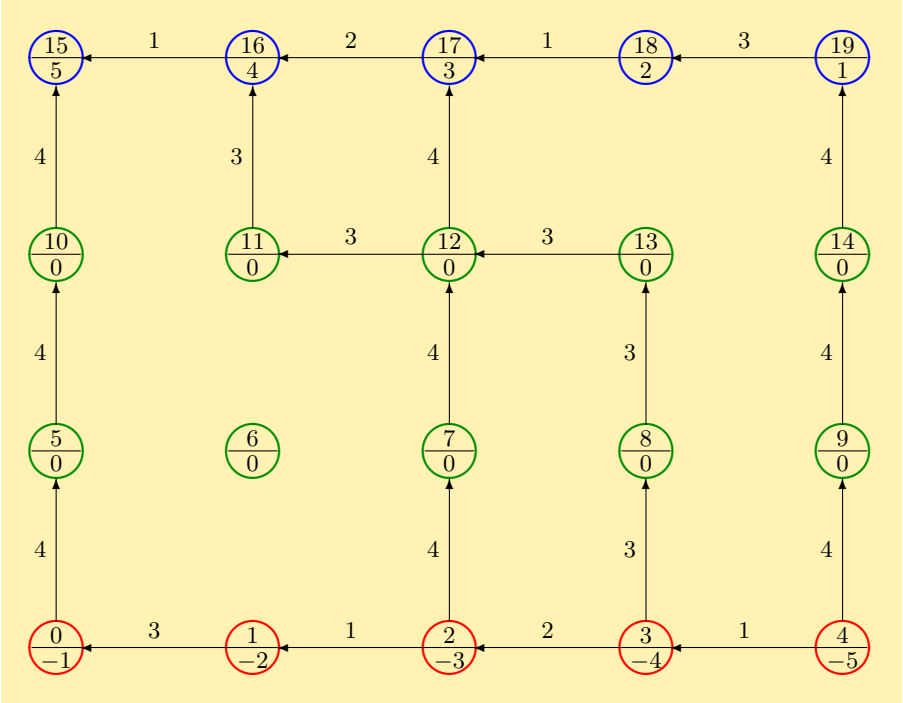

Figure A.3: Solution to instance of Figure A.2

A.9 MIPshell Console Applications

In this section we show how to build a *MipConsoleAppWz application* in Microsoft Visual Studio .Net using. We will solve problem (**??**). Please follow the instructions below to get at the end the problem solved.

1. Click: File → New → Projects
   1.1. Project name: type in ”prodmix”
   1.2. Location: choose a directory for the application
   1.3. Click on ”MIP Console Application”
   1.4. Do not check the ”Check Box”, click on ”Finish”
   1.5. Click again on ”OK”
2. Click on ”FileView” to see four files created by Application Wizard:
   - main.cpp
   - prodmix.h
   - prodmix.cpp
   - prodmi[.mod

3. Open the ”prodmix.mod” file which contents is as below:

   #include ”prodmix.h”

   int model()
   {
   // TODO: Add here code for your model
   }

4. In place of the ”TODO” line, type in the same code as in Listing A.2.

5. Compile and run the program.

6. To see the result, open the file **`prodmix.sol`**.

A.9.1 MIP-MFC Applications

If you want to develop an MFC based application, just choose MIP-MFC AppWizard (exe) + MIP to create the skeleton application with the following additional features (as compared to the skeleton application created by MFC AppWizard (exe)):

- two libraries, mipdll.lib and mipshell.lib, added to project library list;
- custom build step to process **.mod** files;
- include directories for MIPCL and **MIPshell**;
- skeleton for new problem class derived from **CProblem** (see files

  ”C”+Name+”Mod.h” and **”C”+Name+”Mod.mod”**

  where *Name* is project name).

Bibliography

[1] V. Chvatal. Linear Programming, Freeman, New York (1983).

[2] G.L. Nemhauser, L.A. Wolsey. Integer and Combinatorial Optimization, Wiley (1988).

[3] M. Padber. Linear Optimization and Extensions, Springer-Verlag Berlin Heidelberg (1995).

[4] N.N. Pisaruk. Models and Methods of Mixed Integer Programming (in Russian), Belarus State University, Minsk (2010).

[5] A. Schriver. Theory of Linear and Integer Programming. Wiley, Chichester (1986).

[6] H.P. Williams. Model building in mathematical programming. Wiley (1999).

[7] L.A.Wolsey. Integer Programming, Wiley (1998).

Index