Nicolai N. Pisaruk

Mixed Integer Programming in Operations Management (MIPCL-PY manual)

MINSK 2017

N.N. Pisaruk. *Mixed Integer Programming in operations management (MIPCL-PY manual).*

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.

This manual 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 manual are converted into Python programs using the **MIPCL-PY** module 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 for developing applications that support particular operations management decisions.

Aimed at undergraduates, postgraduates, research students and managers, this manual 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. 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.

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:

- 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?
- Where do we sell our products and services?

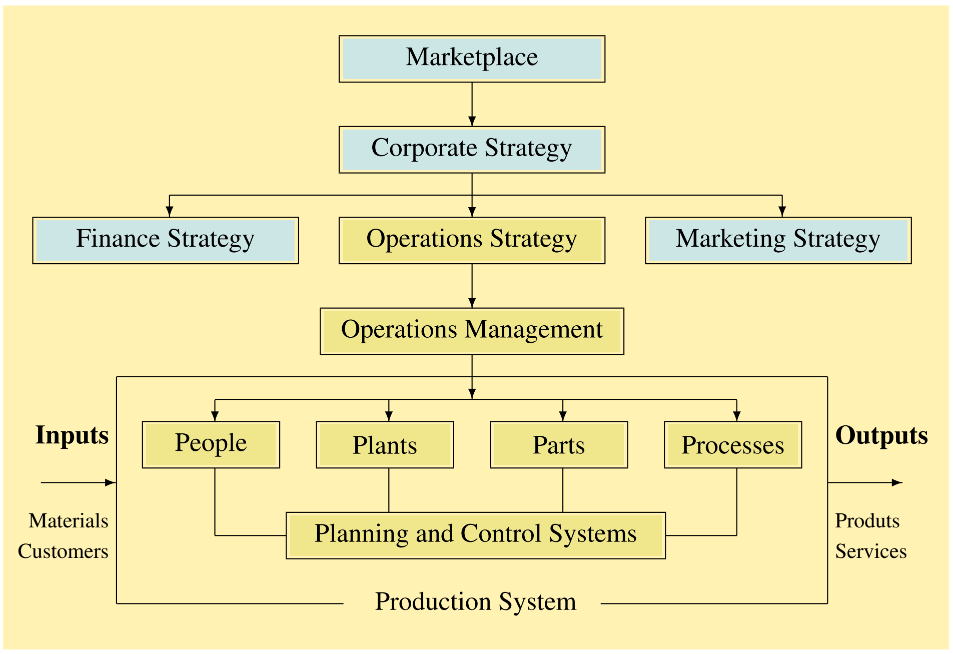

Figure 1: Firm's decision chart

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?

All example applications considered in this manual (and many more) are included into the **MIPCL-PY** installation bundle. Specifically, the python modules are in the `mipcl-py/models` folder, while examples of usage of those modules are in the `mipcl-py/tests` folder. Here `mipcl-py` stands for the installation directory of **MIPCL-PY**. For example, the module `multlotsize`, which implements the multiproduct lot-sizing problem, is in `mipcl-py/models`, and an example, `testMultLotSize.py`, of its usage is in `mipcl-py/test/multlotsize`.

Chapter 1

Aggregate Production Planning

Aggregate production planning is intermediate-range planning (usually covers a planning horizon from one to several 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[1] levels. In this chapter we consider aggregate production planning models 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 Single Product Lot-Sizing

Simple inventory models, such as EOQ and its extensions, are based on the assumption that the consumption is instantaneous and the demand for product is constant per time unit These assumptions 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 product production and storage costs also depends on the season. In this section we consider a few more realistic model.

1.1.1 Basic Single Product Lot-Sizing Problem

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 (includes storage cost, insurance, taxes, and the cost of capital tied up in inventory);

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:

[1]*Inventories* are stocks of goods being held for future use or sale.

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:

$$\sum_{t=1}^{T} p_t x_t + \sum_{t=1}^{T} h_t y_t + \sum_{t=1}^{T} f_t y_t \to \min \tag{1.1a}$$

$$s_{t-1} + x_t = d_t + s_t, \quad t = 1, \dots, T, \tag{1.1b}$$

$$x_t \le u_t y_t, \quad t = 1, \dots, T, \tag{1.1c}$$

$$s_t, x_t \ge 0, \; y_t \in \{0, 1\}, \quad t = 1, \dots, T. \tag{1.1d}$$

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

The balancing constraints (1.1b) ensure a proper transition from any period to the next one: 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 (1.1c) (such constraints are also called *variable upper bounds*) require 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 (1.1a) is to minimize the total production cost.

1.1.2 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 each time period 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 $\tau = 1, \dots, T_p$. As before, h_t and d_t denote the unit storage cost and demand in period $t = 1, \dots, T$.

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 (1.1) is extended as follows:

$$\sum_{\tau=1}^{T_p} \sum_{t=1}^{T} c_{\tau t} x_{\tau t} + \sum_{\tau=1}^{T_p} f_\tau y_\tau \to \min, \tag{1.2a}$$

$$\sum_{\tau=1}^{T_p} x_{\tau t} = d_t, \quad t = 1, \dots, T, \tag{1.2b}$$

$$\sum_{t=1}^{T} x_{\tau t} \le u_\tau y_\tau, \quad \tau = 1,\dots,T_p, \tag{1.2c}$$

$$x_{\tau t} \ge 0, \quad \tau = 1,\dots,T_p,\ t = 1,\dots,T, \tag{1.2d}$$

$$y_\tau \in \{0,1\}, \quad \tau = 1,\dots,T_p. \tag{1.2e}$$

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

$$c_{\tau t} \stackrel{\text{def}}{=} \begin{cases} p_\tau, & t = \mathcal{T}(\tau), \\ p_\tau + b_{\tau t}, & t < \mathcal{T}(\tau), \\ p_\tau + \sum_{\bar{t}=t}^{\tau-1} h_{\bar{t}}, & t > \mathcal{T}(\tau), \end{cases} \tag{1.3}$$

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

The formulation (1.2) does not take into account the initial stock. But we can correct this by introducing in the first time period a production period, say indexed by 0, with production capacity u_0 equal to the initial stock, and production cost $p_0 = 0$.

In the conclusion, let us note that the famous transportation problem is a special case of MIP (1.2) when all fixed costs f_τ are zeroes (what implies that all y_τ may be fixed to 1).

1.1.3 MIPCL-PY Implementation

Our implementation of MIP (1.2) in Listing 1.1 is straightforward.

```
import mipshell
from mipshell import *

class Backloglotsize(Problem):
    def model(self,d,u,f,c):
        self.f, self.c = f, c
        T, Tp = len(d), len(u)
        self.x = x = VarVector((Tp,T),"x")
        self.y = y = VarVector([Tp],"y",BIN)

        minimize(
            sum_(c[tau][t]*x[tau][t] for tau in range(Tp)\
                                 for t in range(T) if c[tau][t] >= 0) +\
            sum_(f[tau]*y[tau] for tau in range(Tp))
        )

        for t in range(T):
            sum_(x[tau][t] for tau in range(Tp)) == d[t]

        for tau in range(Tp):
            sum_(x[tau][t] for t in range(T)) <= u[tau]*y[tau]

        for tau in range(Tp):
            for t in range(T):
                if c[tau][t] < 0:
                    x[tau][t] == 0;
```

Listing 1.1: MIPCL-PY implementation of lot-sizing with backordering: `model`

Input parameters:

- `d`: list, where `d[t]` is demand for product in period `t`;
- `u`: list, where u[t] is is production capacity in period `t`;
- f: list, where f[t] is fixed cost of starting production in period `t`;
- `c`: list of lists, where `c[tau][t]` is cost of producing one unit of product in period `t`, and then supplying that unit to meet demand for period `tau`.

When a solution is found, one may want to print it in a readable way. Listing 3.6 presents a procedure — which is a member of `Backloglotsize` — that does this job.

```
def printSolution(self):
    f, c = self.f, self.c
    x, y = self.x, self.y
    Tp, T = len(x), len(x[0])

    prodCost, fxCost = 0, 0
    print('␣_________________________________')
    print('|␣Prod.␣␣|␣produces␣|␣for␣demand␣|')
    print('|␣period␣|␣(units)␣␣|␣␣␣period␣␣␣|')

    for tau in range(Tp):
        prod=k=0;
        if y[tau].val > 0.5:
            print('|--------+-----------+------------|')
            fxCost += f[tau]

            for t in range(T):
                val = int(x[tau][t].val)
                if val > 0:
                    prod += val

                    prodCost += c[tau][t]*val
                    if k > 0:
                        str = '|␣␣␣␣␣␣␣␣␣|␣'
                    else:
                        str = '|␣␣{:4d}␣␣|␣'.format(tau)
                    print(str + '{:8d}␣|␣␣␣␣{:4d}␣␣␣␣|'.format(val,t+1))
                    k += 1
    print('␣----------------------------------␣')
    print('Production␣cost␣–␣' + repr(prodCost))
    print('Fixed␣cost␣␣␣␣␣␣␣–␣' + repr(fxCost))
```

Listing 1.2: MIPCL-PY implementation of lot-sizing with backordering: `printSolution`

1.1.4 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;

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	100000	200
2	Reg. time	2000	1500	1550	1600	1650	0	1000
	Overtime	2500	2000	2050	2100	2150	100000	400
3	Reg. time	—	2000	1500	1550	1600	0	100000
	Overtime	—	2500	2000	2050	2100	100000	400
4	Reg. time	—	—	2000	1500	1550	0	600
	Overtime	—	—	2500	2000	2050	100000	200
Demand		1000	1000	1000	1000	200	4500	

- 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.

Table 1.1 presents our example as an instance of the lot-sizing problem (1.2). Here we have

- 9 production periods: one artificial period represents the initial stock, the other 8 correspond to regular time and overtime production in each of four weeks;
- 5 consumption periods: each of four weeks is a consumption period, and one artificial period represents the stock at the end of the month.

The numbers in four columns marked as "Sales periods" are the costs $c_{\tau,t}$, calculated by the rule (1.3). A dash in a cell indicates that a car produced in the period that corresponds to the cell row cannot be delivered to a customer that ordered a car to be delivered in the period that corresponds to the cell column.

To solve this example, we wrote the program presented in Listing 1.3.

```
#!/usr/bin/python
from backloglotsize import Backloglotsize

d = [1000, 1000, 1000, 1000, 200]
u = [100, 600, 200, 1000, 400, 1000, 400,  600, 200]
f = [0, 0, 100000, 0, 100000, 0, 100000, 0, 100000]
c = [
    [   0,   50,  100,  150,  200],
    [1500, 1550, 1600, 1650, 1700],
    [2000, 2050, 2100, 2150, 2200],
    [2000, 1500, 1550, 1600, 1650],
    [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],
]
prob = Backloglotsize("test1")
prob.model(d,u,f,c)
prob.optimize()
prob.printSolution()
```

Listing 1.3: Example of usage of class `Backloglotsize`

If we run the above program, we get the following result.

```
| Prod.   | produces | for demand |
| period  | (units)  |   period   |
|---------+----------+------------|
|     0   |     100  |      1     |
|---------+----------+------------|
|     1   |     600  |      1     |
|---------+----------+------------|
|     2   |     200  |      1     |
|---------+----------+------------|
|     3   |     800  |      2     |
|         |     200  |      5     |
|---------+----------+------------|
|     4   |     100  |      1     |
|         |     200  |      2     |
|---------+----------+------------|
|     5   |     600  |      3     |
|         |     400  |      4     |
|---------+----------+------------|
|     6   |     400  |      3     |
|---------+----------+------------|
|     7   |     600  |      4     |
 ---------------------------------
Production cost - 6700000, Fixed cost - 300000
```

Listing 1.4: Output of program from Listing 1.3

1.2 Multiproduct Lot-Sizing

We need to work out an aggregate production plan for n different products processed 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^i_j: initial stock of product j at the beginning of the planning horizon.
- s^f_j: final stock of product j at the end of the planning horizon.

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

1.2.1 Mixed-Integer Programming Formulation

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 machines of type i working in period t.

Now we formulate the problem as follows:

$$\sum_{t=1}^{T}\sum_{j=1}^{n}(h_{jt}\,s_{jt}+c_{jt}\,x_{jt})+\sum_{t=1}^{T}\sum_{i=1}^{m}f_{it}\,y_{it}\to\min, \tag{1.4a}$$

$$s^i_j+x_{j1}=d_{j1}+s_{j1}+\sum_{k=1}^{n}\rho_{jk}\,x_{k,1},\quad j=1,\dots,n, \tag{1.4b}$$

$$s_{j,t-1}+x_{jt}=d_{jt}+s_{jt}+\sum_{k=1}^{n}\rho_{jk}\,x_{k,t},\quad j=1,\dots,n;\ t=2,\dots,T, \tag{1.4c}$$

$$\sum_{j=1}^{n}\tau_{ij}\,x_{jt}\le l_t y_{it},\quad i=1,\dots,m;\ t=1,\dots,T, \tag{1.4d}$$

$$s_{jT}=s^f_j,\quad j=1,\dots,n, \tag{1.4e}$$

$$0\le s_{jt}\le u_j,\ x_{jt}\ge 0,\quad j=1,\dots,n;\ t=1,\dots,T, \tag{1.4f}$$

$$0\le y_{it}\le m_{it},\ y_{it}\in\mathbb{Z},\quad i=1,\dots,m;\ t=1,\dots,T. \tag{1.4g}$$

The objective (1.4a) prescribes to minimize the total production and inventory expenses. The balance equations (1.4b) and (1.4c) 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 (1.4d) require that the working times of all machines 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 $x_{jt}=0$).

1.2.2 MIPCL-PY Implementation

We implemented our multiproduct lot-sizing application as a class named `MultLotSize`, which description is given in listings 1.5–1.7. The function that implements MIP (1.4), as usually, is called `model`. Its imlementation is given in listings 1.5 and 1.6.

```
import mipshell
from mipshell import *

class MultLotSize(Problem):
    def model(self,l,products,machines):
        def s0(j): return products[j][0][0]
        def sf(j): return products[j][0][1]
        def u(j): return products[j][0][2]
        def d(j,t): return products[j][1][t]
        def c(j,t): return products[j][2][t]
        def h(j,t): return products[j][3][t]
        def rho(j,k): return products[j][4][k]
        def mn(i,t): return machines[i][0][t]
        def f(i,t): return machines[i][1][t]
        def tau(i,j): return machines[i][2][j]

        self.products, self.machines = products, machines
        m, n, T = len(machines), len(products), len(l)
```

Listing 1.5: MIPCL-PY implementation of multiproduct lot-sizing: `model` (part 1)

Input parameters:

- `l`: list, where `l[t]` is duration of period `t`;
- `products`: list of lists, where `product = products[j]` has the following fields:
  - `product[0][0]`: initial stock of product `j`;
  - `product[0][1]`: final stock of product `j`;
  - `product[0][2]`: storage capacity for product `j`;
  - `products[1][t]`: demand in period `t` for product `j`;
  - `products[2][t]`: unit production cost in period `t` for product `j`;
  - `product[3][t]`: unit holding cost in period `t` for product `j`;
  - `product[4][k]`: quantity of product `j` used in production of one unit of product `k`;
- `machines`: list of lists, where `machine = machines[i]` has the following fields:
  - `machine[0][t]`: number of machines of type `i` available in period `t`;
  - `machine[1][t]`: fixed cost of running one machine of type `i` in period `t`;
  - `machine[2][j]`: time needed to produce one unit of product `j` on machine `i`.

To keep our **MIPCL-PY**-model closely consistent with the MIP (1.4), we define a number of single-line local functions that extract the values of model parameters from input data. With those functions implementation of (1.4) is straightforward.

```
        self.s = s = VarVector([n,T],"s")
        self.x = x = VarVector([n,T],"x",INT)
        self.y = y = VarVector([m,T],"y",INT)

        minimize(
           sum_(c(j,t)*x[j][t] + h(j,t)*s[j][t] for j in range(n) for t in range(T))
         + sum_(f(i,t)*y[i][t] for i in range(m) for t in range(T))
        )
        for j in range(n):
            s0(j) + x[j][0] == d(j,0) + s[j][0]\
            + sum_(rho(j,k)*x[k][0] for k in range(n) if rho(j,k) > ZERO)
        for t in range(1,T):
            for j in range(n):
                s[j][t-1] + x[j][t] == d(j,t) + s[j][t]\
                + sum_(rho(j,k)*x[k][t] for k in range(n) if rho(j,k) > ZERO)
        for i in range(m):
            for t in range(T):
                sum_(tau(i,j)*x[j][t] for j in range(n) if tau(i,j) > ZERO) \
                        <= l[t]*y[i][t]
                y[i][t] <= mn(i,t)
        for j in range(n):
            for t in range(T-1):
                s[j][t] <= u(j)
            s[j][T-1] == sf(j)
```

Listing 1.6: MIPCL-PY implementation of multiproduct lot-sizing: `model` (part 2)

Let us note that the value of `ZERO` is defined in the `mipshall.py` module to be `1.0e-12`.

Printing schedules

When a solution is found, one may want to print it in a readable way. Listing 1.7 presents a procedure, named `printSolution`, — which is a member of `MultLotSize` — that does this job. This procedure displays three tables named "Production", "Stock", and "Machines". All these tables are printed by the `printTbl` procedure, which implementation is given in Listing 1.7 as well.

```
    def printTbl(self, tblName, rowTitle, colTitle, x):
        n, T = len(x), len(x[0])
        print('\n␣␣␣␣' + tblName + ':')
        print('␣' + '_'*(10*n+8))
        print('|␣␣␣␣␣␣␣␣|' + rowTitle.center(10*n-1) + '|')
        print('|' + '-'*(10*n+8) + '|')
        str = '|␣' + colTitle.center(6) + '␣|'
        for j in range(n):
            str += repr(j+1).center(8) + '␣|'
        print(str)
        print('|--------+' + '---------+'*(n-1) + '---------|')
        for t in range(T):
            str = '|␣' + repr(t+1).center(6) + '␣|'
            for j in range(n):
                str += format(x[j][t].val,'.2f').rjust(8) + '␣|'
            print(str)
        print('␣' + '-'*(10*n+8))

    def printSolution(self):
        print('Objective␣value␣=␣{:.4f}'.format(self.getObjVal()))
        self.printTbl('Production','Products','Period',self.x)
        self.printTbl('Stock','Products','Period',self.s)
        self.printTbl('Machines','Machines','Period',self.y)
```

Listing 1.7: MIPCL-PY implementation of multiproduct lot-sizing: auxiliary procedures

1.2.3 Example of Usage

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.

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 the setup costs 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. These costs are given in Table 1.5. We see that it is more expensive to produce in the cold months (January, February, and March) than in the warm ones (April, May, and June).

It is possible to store up to 100 units of each product at a cost of $1 per unit per month. There is not any unit of any product in stock at the beginning of the planning horizon but it is decided 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 get an answer to the above question, we wrote the program displayed in Listing 1.8.

```
#!/usr/bin/python
from multlotsize import MultLotSize

```

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

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

Table 1.5: Production Costs

Periods	Products						
	1	2	3	4	5	6	7
January, February, March	17	19	16	8	9	16	25
April, May, June	16	17	15	7	8	14	22

```
l = [304, 304, 352, 352, 336, 336] # durations of all periods

products = [
  [
    (0, 50, 100),
    [ 500, 600, 300, 200, 0, 500],
    [17, 17, 17, 16, 16, 16],
    [1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 0, 1]
  ],
  [
    (0, 50, 100),
    [1000, 700, 600, 300, 100, 500],
    [19, 19, 19, 17, 17, 17],
    [1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0]
  ],
  [
    (0, 50, 100),
    [300, 200,  0, 400, 500, 100],
    [16, 16, 16, 15, 15, 15],
    [1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 0, 0]
  ],
  [
    (0, 50, 100),
    [300, 0, 0, 500, 100, 300],
    [8, 8, 8, 7, 7, 7],
    [1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0]
  ],
  [
    (0, 50, 100),
    [800, 400, 500, 200, 1000, 1100],
    [9, 9, 9, 8, 8, 8],
    [1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0]
  ],
  [
    (0, 50, 100),
    [200, 300, 400, 0, 300, 500],
    [16, 16, 16, 14, 14, 14],
    [1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 1]
  ],
  [
    (0, 50, 100),
    [100, 150, 100, 100, 0, 60],
    [25, 25, 25, 22, 22, 22],
    [1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0]
  ]
]

machines = [
  [
    [6, 5, 4, 5, 3, 5],
    [500, 500, 500, 500, 500, 500],
    [0.5, 0.7, 0.0, 0.0, 0.3, 0.2, 0.5]
  ],
  [
    [4, 5, 6, 4, 5, 3],
    [450, 450, 450, 450, 450, 450],
    [0.1, 0.2, 0.0, 0.3, 0.0, 0.6, 0.0]
  ],
  [
    [5, 3, 5, 6, 5, 4],
    [450, 450, 450, 450, 450, 450],
    [0.2, 0.0, 0.8, 0.0, 0.0, 0.0, 0.6]
  ],
  [
    [8, 6, 7, 8, 7, 6],
    [550, 550, 550, 550, 550, 550],
    [0.05, 0.03, 0.0,  0.07, 0.01, 0.0, 0.08]
  ],
  [
    [3, 2, 2, 2, 3, 3],
    [600, 600, 600, 600, 600, 600],
    [0.0, 0.0, 0.01, 0.0, 0.05, 0.0, 0.05]
  ],
  [
    [4, 3, 4, 4, 3, 4],
    [400, 400, 400, 400, 400, 400],
    [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.15]
  ]
]

prob = MultLotSize("test1")
prob.model(1,products,machines)
prob.optimize()
prob.printSolution()
```

Listing 1.8: Example of usage of class MultLotSize

The above program describes our example via a number of arrays. Next, an object, named `prob`, of the `MultLotSize` class is created. Calling `prob.model()` and `prob.optimize()`, we solve our example. To display the result, we call `prob.printSolution()`.

If we run the program from Listing 1.8, we get the following result.

```
Objective value = 396464.0000

Production:
 ----------------------------------------------------------------------------
|         |                          Products                                |
|-----------------------------------------------------------------------------|
| Period  |   1     |   2     |   3     |   4     |   5     |   6     |   7     |
|---------+---------+---------+---------+---------+---------+---------+---------|
|    1    | 1438.00 | 1000.00 | 1100.00 |  300.00 |  800.00 |  479.00 |  138.00 |
|    2    | 1246.00 |  800.00 |  700.00 |    0.00 |  500.00 |  539.00 |  121.00 |
|    3    |  802.00 |  500.00 |  536.00 |    0.00 |  436.00 |  642.00 |   91.00 |
|    4    |  572.00 |  304.00 |  662.00 |  500.00 |  264.00 |  140.00 |  100.00 |
|    5    | 1000.00 |   96.00 | 1355.00 |  100.00 |  908.00 |  592.00 |  100.00 |
|    6    | 1702.00 |  550.00 | 1247.00 |  350.00 | 1142.00 |  478.00 |   10.00 |
________________________________________________________________________________

Stock:
 -------------------------------------------------------------------------------
|          |                              Products                              |
|--------------------------------------------------------------------------------|
|  Period  |    1     |    2     |    3     |    4     |    5     |    6     |    7     |
|----------+----------+----------+----------+----------+----------+----------+----------|
|    1     |    0.00  |    0.00  |    0.00  |    0.00  |    0.00  |    3.00  |   38.00  |
|    2     |   25.00  |  100.00  |    0.00  |    0.00  |  100.00  |    0.00  |    9.00  |
|    3     |    0.00  |    0.00  |  100.00  |    0.00  |   36.00  |   60.00  |    0.00  |
|    4     |    8.00  |    4.00  |   98.00  |    0.00  |  100.00  |    0.00  |    0.00  |
|    5     |    0.00  |    0.00  |   45.00  |    0.00  |    8.00  |   92.00  |  100.00  |
|    6     |   50.00  |   50.00  |   50.00  |   50.00  |   50.00  |   50.00  |   50.00  |
________________________________________________________________________________

Machines:
 --------------------------------------------------------------------
|          |                       Machines                          |
|---------------------------------------------------------------------|
|  Period  |    1     |    2     |    3     |    4     |    5     |    6     |
|----------+----------+----------+----------+----------+----------+----------|
|    1     |    6.00  |    3.00  |    5.00  |    1.00  |    1.00  |    2.00  |
|    2     |    5.00  |    2.00  |    3.00  |    1.00  |    1.00  |    2.00  |
|    3     |    3.00  |    2.00  |    2.00  |    1.00  |    1.00  |    1.00  |
|    4     |    2.00  |    1.00  |    2.00  |    1.00  |    1.00  |    1.00  |
|    5     |    3.00  |    2.00  |    4.00  |    1.00  |    1.00  |    2.00  |
|    6     |    5.00  |    2.00  |    4.00  |    1.00  |    1.00  |    2.00  |
____________________________________________________________________
```

Listing 1.9: Output of program from Listing 1.8

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 work flow cycle. Add inventory time and receivables collection time to this and you get a long cash flow cycle. Thus, work flow equals cash flow, and work flow 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 Scheduling Problems

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 job starts at time 0); or to maximize the number of processed jobs.

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 (machines). Let $\mathcal{M}_j \subseteq \mathcal{P} \stackrel{\text{def}}{=} \{1,\dots,m\}$ denote the subset of processors that can fulfill job $j \in \mathcal{J} \stackrel{\text{def}}{=} \{1,\dots,n\}$.

Each job j is characterized by the following parameters:

- w_j: weight;
- r_j, d_j: release and due dates (the job must be processed during the time interval $[r_j, d_j]$);
- p_{ji}: processing time on processor $i \in \mathcal{M}_j$.

Precedence relations between jobs are given by an acyclic 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 $\sum_{j=1}^n w_j(1-U_j)$.

2.1.1 Time-Index Formulation

A time-index formulation is based on time-discretization, i.e., the planning horizon — from time $R_{\min} = \min_{1\le j\le n} r_j$ to time $D_{\max} = \max_{1\le j\le n} d_j$ — is divided into periods, and period t starts at time $t-1$ and ends at time t. We represent a *schedule* by a family of *decision* binary variables $\{x_{jit}\}$, where x_{jit} takes value 1 if job j starts in period t on processor i; otherwise, $x_{jit} = 0$. To formulate precedence relations we need three families of *auxiliary* variables, which are uniquely defined by the decision variables x_{jit}:

- $y_j = 1$ if job j is processed; otherwise, $y_j = 0$;
- s_j: start time of job j;
- p^t_j: processing time (duration) of job j.

We consider the following time-index formulation[1]:

$$\sum_{j=1}^{n} w_j y_j \to \max, \tag{2.1a}$$

$$\sum_{\substack{1\le j\le n:\\ r_j\le t\le d_j-p_{ji}}} \sum_{\tau=\max\{t-p_{ji},r_j\}}^{\min\{t,d_j-p_{ji}\}} x_{ji\tau} \le 1, \quad t = R_{\min},\dots,D_{\max};\ i = 1,\dots,m, \tag{2.1b}$$

$$y_j = \sum_{i\in\mathcal{M}_j} \sum_{t=r_j}^{d_j-p_{ji}} x_{jit}, \quad j = 1,\dots,n, \tag{2.1c}$$

$$s_j = \sum_{i\in\mathcal{M}_j} \sum_{t=r_j}^{d_j-p_{ji}} t\cdot x_{jit}, \quad j = 1,\dots,n, \tag{2.1d}$$

$$p^t_j = \sum_{i\in\mathcal{M}_j} \sum_{t=r_j}^{d_j-p_{ji}} p_{ji}\cdot x_{jit}, \quad j = 1,\dots,n, \tag{2.1e}$$

$$y_{j_1} - y_{j_2} \ge 0, \quad (j_1,j_2)\in E, \tag{2.1f}$$

$$s_{j_2} - s_{j_1} \ge p^t_{j_1}, \quad (j_1,j_2)\in E, \tag{2.1g}$$

$$x_{jit}\in\{0,1\}, \quad i\in\mathcal{M}_j;\ t = r_j,\dots,d_j-p_{ji};\ j = 1,\dots,n, \tag{2.1h}$$

$$y_j\in\{0,1\}, \quad j = 1,\dots,n. \tag{2.1i}$$

The objective (2.1a) is to maximize the weighted number of processed jobs.

The capacity constraints (2.1b) state that any processor fulfills at most one job during any time period.

The equations (2.1c), (2.1d), and (2.1e) determine the values of all auxiliary variables, y_j, s_j, and p^t_j. Simultaneously, the equations (2.1c) imply that each job can start only once (because $y_j\in\{0,1\}$).

The precedence relations are expressed by the inequalities (2.1f) and (2.1g). For each pair of related jobs $(j_1,j_2)\in E$, (2.1f) requires that j_2 is not processed if j_1 is not processed, while (2.1g) requires that, if both jobs, j_1 and j_2, are processed, then j_1 must be finished when j_2 starts.

[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.

The time-index formulation can be easily modified to model many other types of scheduling problems, which is its important advantage. For example, if we set $y_j = 1$ for all j, and redefine the objective as follows

$$\sum_{j=1}^{n} \sum_{i \in \mathcal{M}_j} \sum_{t=l_j}^{u_j - p_{ji}} (t + p_{ji}) x_{jit} \to \min,$$

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 optimal objective value of the LP-relaxation of the time-index formulation provides a strong bound for the scheduling objective value: it dominates the bounds provided by other IP formulations. This is because the LP-relaxations of the time-index formulations are *non-preemptive*. 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 there may be up to nT variables. As a consequence, 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 times will be large.

2.1.2 MIPCL-PY Implementation

Our **MIPCL-PY** implementation of IP (2.1) is presented in Listing2.1.

```
import mipshell
from mipshell import *

class Schedule(Problem):
    def model(self,jobs):
        def r(j): return jobs[j][0]
        def d(j): return jobs[j][1]
        def w(j): return jobs[j][2]
        def p(j,i): return jobs[j][3][i]
        def M(j): return jobs[j][3].keys()
        def prec(j): return jobs[j][4]

        self.jobs = jobs
        n = len(jobs)
        self.m = m = 1 + max(i for j in range(n) for i in M(j))
        Rmin = min(r(j) for j in range(n))
        Dmax = 1 + max(d(j) for j in range(n))

        s = VarVector([n],'s')
        pt = VarVector([n],'pt')
        y = VarVector([n],'y',BIN)
        self.x = x = {(j,i,t): Var('x' + str((j,i,t)),BIN) \
                       for j in range(n) for i in M(j) \
                       for t in range(r(j),d(j)-p(j,i)+1)}

        maximize(sum_(w[j]*y[j] for j in range(n)))

        for i in range(m):
            for t in range(Rmin,Dmax):
                sum_(x[j,i,tau] for j in range(n) if i in M(j) \
                  for tau in range(max(t-p(j,i),r(j)),min(t+1,d(j)-p(j,i)+1))) <= 1
        for j in range(n):
            y[j] == sum_(x[j,i,t] for i in M(j) for t in range(r(j),d(j)-p(j,i)+1))
            s[j] == sum_(t*x[j,i,t] for i in M(j) \
                                        for t in range(r(j),d(j)-p(j,i)+1))
            pt[j] == sum_(p(j,i)*x[j,i,t] for i in M(j) \
                                              for t in range(r(j),d(j)-p(j,i)+1))
            for j1 in prec(j):
                y[j] <= y[j1]
                s[j] >= s[j1] + pt[j1]
```

Listing 2.1: MIPCL-PY implementation of scheduling problem: `model`

Input parameters:

- jobs: list of integers, where, for each task `j=1,...,n=size(jobs)`,
  - r(j)=tasks[j][0] and d(j)=tasks[j][1] are release and due dates;
  - w(j)=tasks[j][2] weight;
  - tasks[j][3] is a dictionary, where
    - ∗ M(j)=tasks[j][3].keys() is a subset of machines (processors) that are able to process task j;
    - ∗ p(j,i)=tasks[j][3][i] is processing time of task j on machine i;
    - ∗ prec(j)=tasks[j][4] is a subset of preceding tasks.

In our implementation we define variables in somewhat unusual way. Starting from an empty dictionary (`x = {}`), for any feasible tuple `(j,i,t)` of indices, we add to the dictionary a binary (of type ”`BIN`”) variable `x[j,i,t]` named as `x(j,i,t)` (this name is only used when printing the solution). Alternatively, we could first determine a set

```
JIT = ((j,i,t) for j in range(T) for i in M(j) for t in range(r(j),d(j)-p(j,i)+1))
```

of tuples for compatible indices, and then define an array of binary variables as follows:

```
x = VarArray(JIT,"x",BIN)
```

Definitely, the latter approach is clearer but less efficient than the former one.

When a solution has been found, we can print it in a readable way by calling the procedure `printSchedule()` — which is a member of `Schedule`) — from Listing 2.2.

```
    def printSolution(self):
        def r(j):
            return jobs[j][0]
        def d(j):
            return jobs[j][1]
        def p(j,i):
            return jobs[j][3][i]
        def M(j):
            return jobs[j][3].keys()

        jobs=self.jobs
        x = self.x
        n = len(jobs)
        print('Sheduling␣cost␣␣=␣{:.4f}'.format(self.getObjVal()))
        print('␣-------------------------------------------')
        print('|␣␣Job␣|␣Processor␣|␣Starts␣|␣␣␣Ends␣|')
        print('+-------------------------------------------+')
        for j in range(n):
            for i in M(j):
                for t in range(r(j),d(j)-p(j,i)+1):
                    if x[j,i,t].val > 0.5:
                        print('|␣{:4d}␣|␣{:9d}␣|␣{:6d}␣|␣{:6d}␣|'.format(
                                                         j+1,i+1,t,t+p(j,i)))
                        break
        print('␣-------------------------------------------')
```

Listing 2.2: MIPCL-PY implementation of scheduling problem: auxiliary procedures

2.1.3 Example of Usage

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, we prepared the program presented in Listing 2.3.

```
#!/usr/bin/python
from schedule import Schedule

jobs = {
  0: [0,  4, 2, {0: 2, 1: 3}, ()],
  1: [0,  6, 1, {0: 1, 1: 4}, ()],
  2: [0,  5, 3, {0: 2, 1: 2}, ()],
  3: [3,  7, 1, {0: 2, 1: 1}, ()],
  4: [2,  8, 3, {0: 4, 1: 3}, ()],
  5: [6, 12, 2, {0: 3, 1: 4}, ()],
  6: [1, 10, 4, {0: 5, 1: 4}, ()],
  7: [2,  8, 2, {0: 4, 1: 2}, ()],
  8: [0,  9, 1, {0: 1, 1: 3}, ()],
  9: [5, 12, 3, {0: 5, 1: 5}, ()]
}

prob = Schedule("test1")
prob.model(jobs)
prob.optimize()
prob.printSolution()
```

Listing 2.3: Example of usage of class Schedule

If we run the program from Listing 2.3, we get the following result.

```
Sheduling cost = 17.0
 ____________________________________________
|  Job  |  Processor  |  Starts  |    Ends  |
+-------------------------------------------+
|    1  |          1  |       0  |       2  |
|    3  |          2  |       0  |       2  |
|    5  |          2  |       3  |       6  |
|    6  |          1  |       9  |      12  |
|    7  |          1  |       3  |       8  |
|   10  |          2  |       7  |      12  |
 ____________________________________________
```

Listing 2.4: Output of program from Listing 2.3

2.2 Electricity Generation Planning

The *unit commitment problem* is to develop an hourly (or half-hourly) electricity production schedule spanning some planning horizon (a day or a week) so as to decide which generators will be producing and at what levels. The very essence of this problem is to appropriately balance of using generators with different capacities: it is cheaper to produce electricity on more powerful generators while less powerful and smaller generators take less time to switch on or off in case of necessity.

Let T be the number of periods in the planning horizon. 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 $f_i + p_i v$).

2.2.1 Mixed-Integer Programming Formulation

With a 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 following formulation:

$$\sum_{i=1}^{n}\sum_{t=1}^{T}(g_i z_{it} + f_i x_{it} + p_i y_{it}) \to \min \tag{2.2a}$$

$$\sum_{i=1}^{n} y_{it} = d_t, \quad t = 1, \dots, T, \tag{2.2b}$$

$$\sum_{i=1}^{n} u_i x_{it} \geq q\, d_t, \quad t = 1, \dots, T, \tag{2.2c}$$

$$l_i x_{it} \leq y_{it} \leq u_i x_{it}, \quad i = 1, \dots, n;\ t = 1, \dots, T, \tag{2.2d}$$

$$-r_i^1 \leq y_{it} - y_{i,((t-2+T) \bmod T)+1} \leq r_i^2, \quad i = 1, \dots, n;\ t = 1, \dots, T, \tag{2.2e}$$

$$x_{it} - x_{i,((t-2+T) \bmod T)+1} \leq z_{it}, \quad i = 1, \dots, n;\ t = 1, \dots, T, \tag{2.2f}$$

$$z_{it} \leq x_{it},, \quad i = 1, \dots, n;\ t = 1, \dots, T, \tag{2.2g}$$

$$x_{it}, z_{it} \in \{0, 1\}, \quad i = 1, \dots, n;\ t = 1, \dots, T, \tag{2.2h}$$

$$y_{it} \in \mathbb{R}_+, \quad i = 1, \dots, n;\ t = 1, \dots, T. \tag{2.2i}$$

Let us note that period $((t-2+T) \bmod T) + 1$ is immediately followed by period t.

Here the objective (2.2a) is to minimize the total (for all n generators in all T periods) cost of of producing electricity plus the sum of start-up costs. The equations (2.2b) ensure that, for each period, the total amount of electricity produced by all working generators meets the demand at that period. The inequalities (2.2c) require that, in any period, the total capacity of all working generators is at least q times of the demand at that period. The lower and upper bounds (2.2d) impose the capacity restrictions for each generator in each period; simultaneously, these constrains ensure that non-working generators do not produce electricity. The two sided inequalities (2.2e) guarantee that generators cannot increase (ramp up) or decrease (ramp down) their outputs by more than the values of their ramping parameters. The inequalities (2.2g) reflect the fact that any generator is working in a given period only if it has been switched on in this period or it was working in the preceding period.

2.2.2 MIPCL-PY Implementation

A MIPCL-PY implementation of MIP model (2.2) is given in Listing 2.5.

```
import mipshell
from mipshell import *

class Unitcom(Problem):
    def model(self,q,d,units):
        def l(i):
            return units[i][0]
        def u(i):
            return units[i][1]
        def r1(i):
            return units[i][2]
        def r2(i):
            return units[i][3]
        def g(i):
            return units[i][4]
        def f(i):
            return units[i][5]
        def p(i):
            return units[i][6]

        self.units = units
        T, n = len(d), len(units)

        x = VarVector([n,T],"x",BIN)
        self.y = y = VarVector([n,T],"y")
        z = VarVector([n,T],"z",BIN)

        minimize(
          sum_(g(i)*z[i][t] + f(i)*x[i][t] + p(i)*y[i][t] \
                for i in range(n) for t in range(T))
        )

        for t in range(T):
            sum_(y[i][t] for i in range(n)) == d[t]
            sum_(u(i)*x[i][t] for i in range(n))  >= q*d[t]
            for i in range(n):
                y[i][t] >= l(i)*x[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]
```

Listing 2.5: MIPCL-PY implementation of unit commitment problem: `model`

Input parameters:

- `d`: list of demands, where `d[t]` is demand in period `t`;
- `units`: list of tuples of size 7, where, for each unit `i`,
  - `l(i) = units[i][0]` and `u(i) = units[i][1]` are minimum and maximum levels of per period production;
  - `r1(i) = units[i][2]` and `r2(i) = units[i][3]` are ramping parameters;
  - `g(i) = units[i][4]` is start-up cost;
  - `f(i) = units[i][5]` and `p(i) = units[i][6]` are fixed and variable costs.

When a solution has been found, we can print the result using the procedure `printSolution()` — which is a member of `Unitcom` — from Listing 2.6.

```
    def printSolution(self):
        y = self.y
        n, T = len(y), len(y[0])
        print('Optimal objective value: {:.4f}'.format(self.getObjVal()))
        print('\n    Generator powers:')
        print(' ' + '_'*(10*n+9))
        print('|␣␣␣␣␣␣␣␣␣|' + 'Generators'.center(10*n-1) + '|')
        print('+' + '-'*(10*n+9) + '+')
        str = '|␣' + 'Periods'.center(7) + '␣|'
        for i in range(n):
            str += repr(i+1).center(8) + '␣|'
        print(str)
        print('+' + '---------+'*(n+1))
        for t in range(T):
            str = '|␣' + repr(t+1).center(7) + '␣|'
            for i in range(n):
                str += format(y[i][t].val,'.2f').rjust(8) + '␣|'
            print(str)
        print('␣' + '-'*(10*n+9))
```

Listing 2.6: MIPCL-PY implementation of unit commitment problem: `printSolution`

2.2.3 Example of Usage

To test our model, we wrote the following program:

```
#!/usr/bin/python
from unitcom import Unitcom

q = 1.2
d = [50, 60, 50, 100, 120, 90, 90, 100, 90, 120, 110, 70]
units = [
[ 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]
]

prob = Unitcom("test1")
prob.model(q,d,units)
prob.optimize(False)
prob.printSolution()
```

Listing 2.7: Example of usage of class `Unitcom`

Here we want to schedule $n = 5$ generators over $T = 12$ time periods of 2-hours each. The level of reliability is $q = 1.2$.

If we run the program from Listing 2.7, it prints an optimal schedule presented in the following listing.

```
Optimal objective value: 3170.0000

Generator powers:
 -----------------------------------------------------------
|         |                    Generators                    |
+-----------------------------------------------------------+
| Periods |    1     |    2     |    3     |    4     |    5     |
+---------+---------+---------+---------+---------+---------+
|    1    |    0.00 |    0.00 |    5.00 |    0.00 |   45.00 |
|    2    |    0.00 |    0.00 |    5.00 |    0.00 |   55.00 |
|    3    |    0.00 |    0.00 |    5.00 |    0.00 |   45.00 |
|     4      |     0.00 |     0.00 |     5.00 |    30.00 |    65.00 |
|     5      |     0.00 |     0.00 |     5.00 |    40.00 |    75.00 |
|     6      |     0.00 |     0.00 |     5.00 |    20.00 |    65.00 |
|     7      |     0.00 |     0.00 |     5.00 |    20.00 |    65.00 |
|     8      |     0.00 |     0.00 |     5.00 |    20.00 |    75.00 |
|     9      |     0.00 |     0.00 |     5.00 |    20.00 |    65.00 |
|    10      |     0.00 |     0.00 |     5.00 |    40.00 |    75.00 |
|    11      |     0.00 |     0.00 |     5.00 |    30.00 |    75.00 |
|    12      |     0.00 |     0.00 |    10.00 |     0.00 |    60.00 |
  ---------------------------------------------------------------------
```

Listing 2.8: Output of program from Listing 2.7

2.3 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 feed-stocks 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%).
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;

[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.

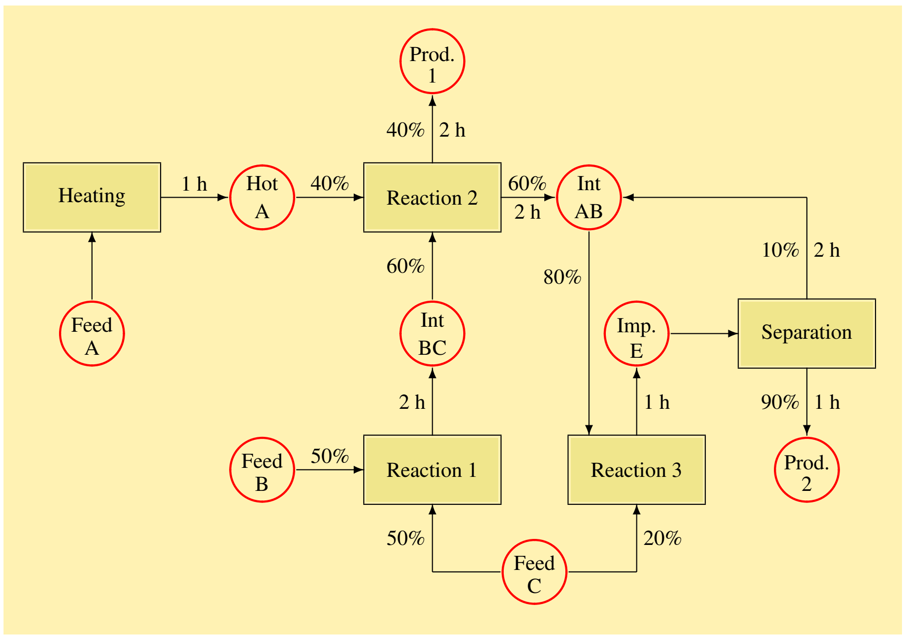

Figure 2.1: State-task network for example process

– *For intermediate E*: 100 kg;

– *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, $d_i \overset{\text{def}}{=} \max_{s \in S_i^{\text{out}}} p_{is}$;

- **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**:** unit cost (price) of product produced in state s;

h_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.

2.3.1 Mixed-Integer Programming 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 $t-1$ and ends at time t. 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:

$$\sum_{s=1}^{q} c_s z_{s,H+1} - \sum_{s=1}^{q}\sum_{t=1}^{H} h_s z_{st} \to \max, \tag{2.3a}$$

$$\sum_{i\in I_j} x_{ijt} \le 1, \quad j = 1,\dots,m;\ t = 1,\dots,H, \tag{2.3b}$$

$$\sum_{i\in I_j}\sum_{\tau=\max\{0,t-d_i+1\}}^{\min\{t,H-d_i\}} x_{ij\tau} \le 1, \quad j=1,\dots,m;\ t=1,\dots,H, \tag{2.3c}$$

$$V_{ij}^{\min} x_{ijt} \le y_{ijt} \le V_{ij}^{\min} x_{ijt}, \quad j=1,\dots,m;\ i\in I_j;\ t=1,\dots,H, \tag{2.3d}$$

$$0 \le z_{st} \le u_s, \quad s=1,\dots,q;\ t=1,\dots,H, \tag{2.3e}$$

$$z_{s,t-1} + \sum_{i\in T_s^{\text{out}}:\ t>p_{is}} \rho_{is}^{\text{out}} \sum_{j\in U_i} y_{ij,t-p_{is}} = z_{st} + \sum_{i\in T_s^{\text{in}}} \rho_{is}^{\text{in}} \sum_{j\in U_i} y_{ijt},$$
$$s=1,\dots,q;\ t=1,\dots,H, \tag{2.3f}$$

$$z_{s,H} + \sum_{i\in T_s^{\text{out}}:\ H>p_{is}} \rho_{is}^{\text{out}} \sum_{j\in U_i} y_{ij,H-p_{is}} = z_{s,H+1}, \quad s=1,\dots,q, \tag{2.3g}$$

$$x_{ijt}=0, \quad t>H-d_i,\ j=1,\dots,m;\ i\in I_j, \tag{2.3h}$$

$$x_{ijt}\in\{0,1\},\ y_{ijt}\in\mathbb{R}_+, \quad j=1,\dots,m;\ i\in I_j; t=1,\dots,H, \tag{2.3i}$$

$$z_{st}\in\mathbb{R}_+, \quad s=1,\dots,q;\ t=1,\dots,H+1. \tag{2.3j}$$

The objective (2.3a) 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 (2.3b) enforce that at any time t, an idle unit j can start only one task. The constraints (2.3c) impose the requirement that at any time any unit processes only one task. The constraints (2.3d) enforce that the batch size of any task must be within the minimum and maximum capacities of the unit performing the task. The constraints (2.3e) impose stock limitations: the amount of material stored in any state s must not exceed the storage capacity for this state. Material balance constraints (2.3f) 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 (2.3g) are specializations of the balance relations for period $H+1$, at this period no task starts. The constraints (2.3h) enforce the tasks to finish within the time horizon.

2.3.2 MIPCL-PY Implementation

Our MIPCL-Py implementation of MIP (2.3) is given in Listing 2.9.

```
import mipshell
from mipshell import *

class Batch(Problem):
    def model(self,H,tasks,states,units):
        def u(s): return states[s][0]
        def z0(s): return states[s][1]
        def h(s): return states[s][2]
        def c(s): return states[s][3]
        def J(i): return tasks[i][0]
        def inS(i): return tasks[i][1].items()
        def outS(i): return tasks[i][2].items()
        def dur(i): return tasks[i][3]
        def I(j): return units[j][0]
        self.H = H
        self.tasks, self.states, self.units = tasks, states, units
        STATES, TASKS, UNITS = states.keys(), tasks.keys(), units.keys()

        self.z = z = {(s,t): Var('z' + str((s,t))) \
                      for s in STATES for t in range(H+1)}
        self.x = x = {(i,j,t): Var('x' + str((i,j,t)),BIN) \
                      for i in TASKS for j in J(i) for t in range(H-dur(i)+1)}
        self.y = y = {(i,j,t): Var('y' + str((i,j,t))) \
                      for i in TASKS for j in J(i) for t in range(H-dur(i)+1)}

        maximize(
            sum_(c(s)*z[s,H] for s in STATES)
            - sum_(h(s)*z[s,t] for s in STATES for t in range(H))
        )

        for t in range(H):
            for j in UNITS:
                sum_(x[i,j,t] for i in I(j) if t+tasks[i][3] <= H) <= 1

        for j in UNITS:
            Vmin, Vmax = units[j][1][0], units[j][1][1]
            for i in I(j):
                for t in range(H-dur(i)+1):
                    y[i,j,t] >= Vmin*x[i,j,t]
                    y[i,j,t] <= Vmax*x[i,j,t]
            for t in range(H):
                sum_(x[i,j,tau] for i in I(j) \
                    for tau in range(max(0,t-dur(i)+1),min(t,H-dur(i)))) <= 1

        for s in STATES:
            for t in range(H):
                z[s,t] <= u(s)
            z[s,0] + sum_(rho*y[i,j,0] for i in TASKS for j in J(i) \
                 for s1,rho in inS(i) if s1 == s) == z0(s)
            for t in range(1,H+1):
                z[s,t-1] + sum_(rho*y[i,j,t-d] for i in TASKS for j in J(i) \
                    for s1,(rho,d) in outS(i) if s1 == s and t >= d) == \
                z[s,t] + sum_(rho*y[i,j,t] for i in TASKS for j in J(i) \
                    for s1,rho in inS(i) if s1 == s and t+dur(i) <= H)
```

Listing 2.9: MIPCL-PY implementation of batch scheduling problem: `model`

Input parameters:

- `H`: length of planning horizon (integer);
- `tasks`: list of tasks, where task i is characterized as follows:
  - `J(i)=tasks[i][0]`: set of units that can process task `i`;
  - `tasks[i][1]`: dictionary that describes states feeding task `i`, we define `inS(i)=tasks[i][1].items()`;
  - `tasks[i][2]`: dictionary that describes states to which task `i` sends its outputs, we define `outS(i)=tasks[i][2].items()`;

  - `dur(i)=tasks[i][3]`: duration of task `i` (integer);

- `states`: list of states, where state `s` is defined by the following parameters:
  - `u(s)=states[s][0]`: storage capacity;
  - `z0(s)=states[s][1]`: initial stock;
  - `h(s)=states[s][2]`: cost of storing one unit of product;
  - `c(s)=states[s][3]`: price of product produced in state `s`;

- `units`: list of units, where each unit `j` is characterized as follows:
  - `I(j)=units[j][0]`: set of tasks that can be performed by unit `j`;
  - `units[j][1][0]` and `units[j][1][1]`: minimum and maximum loads for unit `j`.

When a solution has been found, one may want to print it in a readable format. Listing 2.10 presents a procedure — which is a member of `Batch` — that does this job.

```
    def printSolution(self):
        def dur(i):
            return tasks[i][3]
        def I(j):
            return units[j][0]

        H = self.H
        states, tasks, units = self.states, self.tasks, self.units
        z, y = self.z, self.y
        print('Objective value = {0:.4f}'.format(self.getObjVal()))
        print('\nProducts produced:')
        for s in states.keys():
            print('{0:.4f} in state {1}'.format(z[s,H].val,repr(s)))

        print('\nSchedule:')
        for j in units.keys():
            print('\n' + repr(j) + ':')
            for t in range(H):
                for i in I(j):
                    if t+dur(i) <= H:
                        if y[i,j,t].val > 1.0e-6:
                            print('batch of task {0} starts at {1}, size = {2:.4f}'.
                                format(repr(i),repr(t),y[i,j,t].val))
```

Listing 2.10: MIPCL-PY implementation of batch scheduling problem: `printSolution`

2.3.3 Example of Usage

To optimize our test STN over a horizon of 9 hours, we wrote the program presented in Listing 2.11.

```
#!/usr/bin/python
from batch import Batch

H = 9 # time horizon

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

# Numbers in the four-tuples are:
#   0) storage capacity,
#   1) initial stock,
#   2) cost of storing a unit amount of material,
#   3) price of product produced in the state.
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)
}

# Each unit is described by:
#   1) the set of tasks that this unit is capable to fulfill,
#   2) the 2-tuple which numbers are minimum and maximum unit loads.
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))
}

prob = Batch("STN1")
prob.model(H,tasks,states,units)
prob.optimize()
prob.printSolution()
```

Listing 2.11: Example of usage of class `Batch`

If we run the program from Listing 2.11, we get the following result.

```
Objective value = 4700.0333

Products produced:
803.0000 in state 'Fead_C'
870.0000 in state 'Fead_B'
826.6667 in state 'Fead_A'
0.0000 in state 'Impure_E'
12.5000 in state 'Int_AB'
173.3333 in state 'Prod_1'
0.0000 in state 'Hot_A'
301.5000 in state 'Prod_2'
0.0000 in state 'Int_BC'

Schedule:

'Reactor_2':
batch of task 'Reaction_1' starts at 0, size = 50.0000
batch of task 'Reaction_1' starts at 1, size = 50.0000
batch of task 'Reaction_2' starts at 2, size = 50.0000
batch of task 'Reaction_2' starts at 3, size = 43.3333
batch of task 'Reaction_2' starts at 4, size = 50.0000
batch of task 'Reaction_2' starts at 5, size = 50.0000
batch of task 'Reaction_3' starts at 6, size = 50.0000
batch of task 'Reaction_3' starts at 7, size = 50.0000

'Heater':
batch of task 'Heating' starts at 1, size = 52.0000
batch of task 'Heating' starts at 2, size = 49.3333
batch of task 'Heating' starts at 3, size = 20.0000
batch of task 'Heating' starts at 4, size = 52.0000

'Reactor_1':
batch of task 'Reaction_1' starts at 0, size = 80.0000
batch of task 'Reaction_1' starts at 1, size = 80.0000
batch of task 'Reaction_2' starts at 2, size = 80.0000
batch of task 'Reaction_2' starts at 3, size = 80.0000
batch of task 'Reaction_3' starts at 4, size = 80.0000
batch of task 'Reaction_2' starts at 5, size = 80.0000
batch of task 'Reaction_3' starts at 6, size = 75.0000
batch of task 'Reaction_3' starts at 7, size = 80.0000

'Still':
batch of task 'Separation' starts at 5, size = 80.0000
batch of task 'Separation' starts at 7, size = 125.0000
batch of task 'Separation' starts at 8, size = 130.0000
```

Listing 2.12: Output of program from Listing 2.11

2.4 Project Scheduling

Let us consider a project that consists of n jobs. There are q^r *renewable*[3] (*non-perishable*) resources, with, respectively, R_i^r units of resource i available per unit of time. There are also q^n *nonrenewable*[4] (*perishable*) resources, with, respectively, R_i^n units of resource i available for the entire project. It is assumed that all resources are available when the project starts.

Jobs can be processed in different modes. Execution of any job cannot be interrupted, thus, if processing of a job once started in some mode, it has to be completed in the same mode. If job j is processed in mode m ($m = 1, \dots, M_j$), then it takes p_j^m units of time to process the job, ρ_{jmi}^r units of renewable resource i ($i = 1, \dots, q^r$) are used each period when job j is processed, and ρ_{jmi}^n units of nonrenewable resource i ($i = 1, \dots, q^n$) are totally consumed.

Precedence relations between jobs are given by an acyclic digraph $G = (\{1, \dots, n\}, \mathcal{R})$ defined on the set of jobs: for any arc $(j_1, j_2) \in \mathcal{R}$, job j_2 cannot start until job j_1 is finished.

The goal is to find a schedule[5] with the minimal makespan. The *makespan* of a schedule is defined to be the end time of the lastly finished job.

2.4.1 Integer Programming Formulation

Let us assume that we know an upper bound, H, on the optimal makespan value. We can take as H the makespan of a schedule produced by one of numerous project scheduling heuristics. The planning horizon is divided into H periods numbered from 1 to H, and period t starts at time $t-1$ and ends at time t.

To tighten our formulation, we can estimate (say, by the *critical path method*) the earliest, ef_j and latest, lf_j, finish times of all jobs j.

First, we define the family of decision binary variables:

- $x_{jmt} = 1$ if job j is processed in mode m and completed at the end of period t (at time t); otherwise, $x_{jmt} = 0$.

For modeling purposes, we also need the following families of auxiliary variables:

- T: schedule makespan;
- $d_j \in \mathbb{R}$: duration of job j (d_j depends on the mode in which job j is processed);
- $e_j \in \mathbb{R}$: end time of job j.

[3] Renewable resources are available in the same quantities at any period. Manpower, machines, storage spaces are renewable resources.

[4] In contrast to a renewable resource, which consumption is limited in each period, overall consumption of a nonrenewable resource is limited for the entire project. Money, energy, and raw materials are nonrenewable resources.

[5] A schedule specifies when each job starts and in which mode it is processed.

In these variables the model is written as follows:

$$T \to \min, \tag{2.4a}$$

$$\sum_{m=1}^{M_j} \sum_{t=ef_j}^{lf_j} x_{jmt} = 1, \quad j = 1,\dots,n, \tag{2.4b}$$

$$e_j = \sum_{m=1}^{M_j} \sum_{t=ef_j}^{lf_j} t x_{jmt}, \quad j = 1,\dots,n, \tag{2.4c}$$

$$d_j = \sum_{m=1}^{M_j} \sum_{t=ef_j}^{lf_j} p_j^m x_{jmt}, \quad j = 1,\dots,n, \tag{2.4d}$$

$$T \ge e_j, \quad j = 1,\dots,n, \tag{2.4e}$$

$$\sum_{j=1}^{n} \sum_{m=1}^{M_j} \sum_{t=\max(\tau,ef_j)}^{\min(\tau+p_j^m,lf_j)} \rho^r_{jmi} x_{jmt} \le R^r_i, \quad i = 1,\dots,q^r, \tau = 1,\dots,H, \tag{2.4f}$$

$$\sum_{j=1}^{n} \sum_{m=1}^{M_j} \sum_{t=ef_j}^{lf_j} \rho^n_{jmi} x_{jmt} \le R^n_i, \quad i = 1,\dots,q^n, \tag{2.4g}$$

$$e_{j_2} - e_{j_1} \ge d_{j_2}, \quad (j_1, j_2) \in \mathcal{R}, \tag{2.4h}$$

$$e_j \ge d_j, \quad j = 1,\dots,n, \tag{2.4i}$$

$$x_{jmt} \in \{0,1\}, \quad t = 0,\dots,H,\ m = 1,\dots,M_j, j = 1,\dots,n, \tag{2.4j}$$

$$d_j, e_j \in \mathbb{R}, \quad j = 1,\dots,n. \tag{2.4k}$$

The objective (2.4a) is to minimize the makespan. The equations (2.4b) impose the requirement that each job is processed only in one mode, and it finishes (and starts) only once. For the schedule given by the values of x_{jmt} variables, the equations (2.4c) and (2.4d) calculate the end times, e_j, and durations, d_j, of all jobs. The inequalities (2.4e) imply that, being minimized, T is the makespan of the schedule given by the values of x_{jmt} variables. The inequalities (2.4f) and (2.4g) impose the limitations, respectively, on the renewable and nonrenewable resources. The precedence relations between jobs are given by the inequalities (2.4h). The constraints in (2.4i) are important only for jobs without predecessors; otherwise, starting times of such jobs could be negative.

2.4.2 MIPCL-PY Implementation

To solve instances of the project scheduling problem, we developed a Python class, named `prodSchedule` and presented in listings 2.13, 2.14 and 2.15.

Listing 2.13 describes the constructor and an implementation of IP (2.4), which is, as usually, done in the function named `model`.

```
from mipcl_py.mipshell.mipshell import *

class projectSchedule(Problem):
    def __init__(self, project):
        Problem.__init__(self, project['Name'])
        self.project = project


    def model(self):
        renRes, nonRenRes = self.project['renRes'], self.project['nonRenRes']
        jobs = self.project['Jobs']
        H = self.project['Horizon']
        self.mcp(H)

        T = Var('T',INT)
        e = VarVector([len(jobs)],'e',INT)
        d = VarVector([len(jobs)],'d',INT)
        self.x = x = []
        for j,job in enumerate(jobs):
            ef, lf = job['EF'], job['LF']+1
            x.append([])
            for m in range(len(job['Modes'])):
                x[j].append({})
                for t in range(ef,lf):
                    x[j][m][t] = Var('x({:d},{:d},{:d})'.format(j,m,t),BIN)

        minimize(T)

        for j,job in enumerate(jobs):
            ef, lf = job['EF'], job['LF']+1
# each job starts only in one mode and finishes only once
            sum_(x[j][m][t] for m in range(len(job['Modes'])) for t in range(ef,lf))
                 == 1
            d[j] == sum_(M['Duration']*x[j][m][t] \
                            for m,M in enumerate(job['Modes']) for t in range(ef,lf))
            e[j] == sum_(t*x[j][m][t] for m in range(len(job['Modes'])) \
                                 for t in range(ef,lf))
        T >= e[j]

# renewable resource limits
        for tau in range(1,H+1):
            for r,R in enumerate(renRes):
                sum_(M['renRes'][r]*x[j][m][t] for j,job in enumerate(jobs) \
                     for m,M in enumerate(job['Modes']) \
                     for t in range(max(tau,job['EF']), \
                         min(tau+M['Duration'],job['LF'])+1)) <= R['Quantity']

# nonrenewable resource limits
        for r,R in enumerate(nonRenRes):
            sum_(M['nonRenRes'][r]*x[j][m][t] for j,job in enumerate(jobs) \
                 for m,M in enumerate(job['Modes']) \
                 for t in range(job['EF'],job['LF']+1)) <= R['Quantity']

# precedence relations
        for j2,job in enumerate(jobs):
            if len(job['Prec']) > 0:
                for j1 in job['Prec']:
                    e[j2] - e[j1] >= d[j2]
                else: # j2 has not predecessors
```

```
                    e[j2] >= d[j2]
```

Listing 2.13: MIPCL-PY implementation of project scheduling problem: constructor and `model`

The only parameter of the constructor is

- `project`: data structure that describes a project, its members are
  - `Name`: project name;
  - `Horizon`: length of the planning horizon (an upper bound on the minimum makespan value);
  - `renRes`: list of limits on renewable resources, were
    - `renRes[r]['Name']` is the name of resource `r`,
    - `renRes[r]['Quantity']` gives the available quantity of resource `r`;
  - `nonrenRes`: list of limits on nonrenewable resources, were
    - `nonRenRes[r]['Name']` is the name of resource `r`,
    - `nonRenRes[r]['Quantity']` gives the available quantity of resource `r`;
  - `Jobs`: list of jobs, each job is described by the structure with the following members:
    - `Name`: name, which is used when displaying the schedule;
    - `Modes`: list of modes, each mode `M = Modes[m]` is given by the data structure with the following fields:
      - `M['Duration']`: processing time;
      - `M['renRes']`: list of limits on renewable resources;
      - `M['nonRenRes']`: list of limits on non-renewable resources;
    - `Prec`: list of preceding jobs.

Listing 2.14 presents an auxiliary procedure, `mcp`, that implements the *critical path method*.

```
    def mcp(self, H):
        jobs, renRes = self.project['Jobs'], self.project['renRes']

        for job in jobs:
            es = 0
            for j in job['Prec']:
                ef = jobs[j]['EF']
                if es < ef:
                    es = ef
            job['EF'] = es + min(mode[0] for mode in job['Modes'])
            job['LF'] = H
#           print(repr(job['Job']) + ' early start at ' + repr(es))

        for job in reversed(jobs):
            ls = job['LF'] - min(mode[0] for mode in job['Modes'])
            for j in job['Prec']:
                if jobs[j]['LF'] > ls:
                    jobs[j]['LF'] = ls
```

Listing 2.14: MIPCL-PY implementation of project scheduling problem: `mcp`

For each `job` ∈ `project['Jobs']`, `mcp` computes the early, `job['EF']`, and late, `job['EF']` finish times of this job under the assumptions that all resources are unlimited, and the project has to be fulfilled within the time horizon of H periods, where H is the second input parameter.

When an optimal solution has been found, we can extract the schedule from the values of variables `x[j][m][t]` by calling the procedure `getSchedule`, presented in Listing 2.15. This procedure builds a data structure that is passed to `printSchedule` to display the schedule.

```
    def getSchedule(self):
        schedule = None
        if self.is_solution is not None:
            if self.is_solution == True:
                jobs = self.project['Jobs']
                qr = len(self.project['renRes'])
                x = self.x
                schedule = {}
                schedule['MakeSpan'] = T = int(self.getObjVal() + 0.5)
                schedule['renRes'] = renRes = \
                            [[0 for i in range(qr)] for t in range(T+1)]
                schedule['Mode'] = mode =[]
                schedule['Start'] = start = []
                schedule['End'] = end =[]
                for j,job in enumerate(jobs):
                    stop = False
                    for m,M in enumerate(job['Modes']):
                        for t in range(job['EF'],job['LF']+1):
                            if x[j][m][t].val > 0.5:
                                e = t
                                s = e - M['Duration']
                                mode.append(m)
                                start.append(s)
                                end.append(e)
                                for tau in range(s,e+1):
                                    for r in range(qr):
                                        renRes[tau][r] += M['renRes'][r]
                                stop = True
                                break
                        if stop:
                            break
        return schedule

    def printSchedule(self, schedule):
        jobs = self.project['Jobs']
        qr = len(self.project['renRes'])
        mode, start, end = schedule['Mode'], schedule['Start'], schedule['End']
        rRes = schedule['renRes']
        print('Job schedule:')
        print(' _________________________________')
        print('|   Job  | Mode |  Start |    End |')
        print('|--------+------+--------+--------|')
        for j, job in enumerate(jobs):
            print('| {!s} | {:4d} | {:6d} | {:6d} |'.format(job['Name'].rjust(6)
                [:6],mode[j],start[j],end[j]))
        print(' ---------------------------------')
#
        print('\nResource usage:')
        T = schedule['MakeSpan']
        print('␣____' + '_'*(9*qr))
        str = '|␣␣␣␣␣t␣|'
        for R in self.project['renRes']:
            str += '␣{!s}␣|'.format(R['Name'].rjust(5)[:5])
        print(str)
        print('|--------' + '+---------'*qr) + '|'
        for t in range(1,T+1):
            str = '|␣{:5d}␣|'.format(t)
            for i in range(qr):
                str += '␣{:5d}␣|'.format(rRes[t][i])
            print(str)
        print('␣-----' + '-'*(9*qr))
```

Listing 2.15: MIPCL-PY implementation of project scheduling problem: auxiliary procedures

2.4.3 Example of Usage

The program presented below uses our `projectScheduling` class to solve an instance of the project scheduling problem encoded in the data structure (dictionary) named `project`.

```
#!/usr/bin/python
from projectScheduling import projectScheduling

project = {
    'Name': 'project0',
    'Horizon': 36,
    'renRes': (
        {'Name': "1", 'Quantity': 6},
        {'Name': "2", 'Quantity': 7},
        {'Name': "3", 'Quantity': 6}
    ),
    'nonRenRes': ({'Nane': "1", 'Quantity': 3},),
    'Jobs': (
        {
            'Name': "0",
            'Modes': (
                {'Duration': 3, 'renRes': (3, 2, 1), 'nonRenRes': (0,)},
                {'Duration': 2, 'renRes': (3, 2, 4), 'nonRenRes': (1,)}
            ),
            'Prec': ()
        },
        {
            'Name': "1",
            'Modes': ({'Duration': 5, 'renRes': (2, 4, 2), 'nonRenRes': (0,)},),
            'Prec': ()
        },
        {
            'Name': "2",
            'Modes': ({'Duration': 6, 'renRes': (3, 1, 2), 'nonRenRes': (0,)},),
            'Prec': (0,)
        },
        {
            'Name': "3",
            'Modes': ({'Duration': 2, 'renRes': (4, 3, 1), 'nonRenRes': (0,)},),
            'Prec': (0,)
        },
        {
            'Name': "4",
            'Modes': ({'Duration': 3, 'renRes': (2, 0, 3), 'nonRenRes': (0,)},),
            'Prec': (3,)
        },
        {
            'Name': "5",
            'Modes': ({'Duration': 3, 'renRes': (1, 1, 1), 'nonRenRes': (0,)},),
            'Prec': (3,)
        },
        {
            'Name': "6",
            'Modes': (
                {'Duration': 4, 'renRes': (3, 1, 1), 'nonRenRes': (0,)},
                {'Duration': 3, 'renRes': (3, 2, 1), 'nonRenRes': (1,)}
            ),
            'Prec': (1,)
        },
        {
            'Name': "7",
            'Modes': (
                {'Duration': 4, 'renRes': (3, 2, 3), 'nonRenRes': (0,)},
                {'Duration': 2, 'renRes': (3, 3, 3), 'nonRenRes': (1,)}
            ),
            'Prec': (2,)
        },
        {
            'Name': "8",
            'Modes': ({'Duration': 2, 'renRes': (4, 1, 0), 'nonRenRes': (0,)},),
            'Prec': (4,7)
        },
        {
            'Name': "9",
            'Modes': (
                {'Duration': 5, 'renRes': (2, 2, 2), 'nonRenRes': (0,)},
                {'Duration': 3, 'renRes': (3, 2, 3), 'nonRenRes': (1,)}
            ),
            'Prec': (5,6)
        },
        {
            'Name': "10",
            'Modes': ({'Duration': 3, 'renRes': (5, 4, 2), 'nonRenRes': (0,)},),
            'Prec': (8,)
        }
    )
}

prob = projectScheduling(project)
prob.model()
prob.optimize(False,3600)
schedule = prob.getSchedule()
if schedule is not None:
    prob.printSchedule(schedule)
else:
    print('No schedule has been found')
```

Listing 2.16: Example of usage of class `projectScheduling`

If we run the above program, it prints two tables: one presents an optimal job schedule, the other one shows the usage of resources in all periods of the planning horizon.

```
Job schedule:
 ---------------------------------
|     Job |  Mode |    Start |     End |
|---------+-------+----------+---------|
|       0 |     1 |        0 |       2 |
|       1 |     0 |        0 |       5 |
|       2 |     0 |        7 |      13 |
|       3 |     0 |        3 |       5 |
|       4 |     0 |       10 |      13 |
|       5 |     0 |       11 |      14 |
|       6 |     1 |        6 |       9 |
|       7 |     1 |       14 |      16 |
|       8 |     0 |       17 |      19 |
|       9 |     0 |       14 |      19 |
|      10 |     0 |       20 |      23 |
 ---------------------------------

Resource usage:
 -------------------------------
|     t |     1 |     2 |     3 |
|-------+-------+-------+-------|
|     1 |     5 |     6 |     6 |
|     2 |     5 |     6 |     6 |
|     3 |     6 |     7 |     3 |
|     4 |     6 |     7 |     3 |
|     5 |     6 |     7 |     3 |
|     6 |     3 |     2 |     1 |
|     7 |     6 |     3 |     3 |
|     8 |     6 |     3 |     3 |
|     9 |     6 |     3 |     3 |
|    10 |     5 |     1 |     5 |
|    11 |     6 |     2 |     6 |
|    12 |     6 |     2 |     6 |
|    13 |     6 |     2 |     6 |
|    14 |     6 |     6 |     6 |
|    15 |     5 |     5 |     5 |
|    16 |     5 |     5 |     5 |
|    17 |     6 |     3 |     2 |
|    18 |     6 |     3 |     2 |
|    19 |     6 |     3 |     2 |
|    20 |     5 |     4 |     2 |
|    21 |     5 |     4 |     2 |
|    22 |     5 |     4 |     2 |
|    23 |     5 |     4 |     2 |
 -------------------------------
```

Listing 2.17: Output of program from Listing 2.16

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, very often, 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 the 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 production (or service) departments in a way that minimizes the movements of materials between the departments. 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).

3.1.1 Integer Programming Formulation

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 departments. The equations (3.1b) require that each department be moved to exactly one site, and the inequalities (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}).

3.1.2 MIPCL-PY Implementation

Our implementation of the process layout problem is presented in Listing 3.1.

```
import mipshell
from mipshell import *

class Proclayout(Problem):
    def model(self,p,dist,flows):
        def c(i,j,s,r):
            if s == r: cost = 0
            else:
                if s > r:
                    k = s
                    s = r
                    r = k
                cost = dist[s][r-(s+1)] * flows[i][j-(i+1)]
            return cost

        self.dist, self.flows = dist, flows
        n, m = len(p), len(p[0])
        self.x = x = VarVector([n,m],"x",BIN)
        y = VarVector([n,n,m,m],"y",BIN)

        minimize(
            sum_(p[i][s]*x[i][s] for i in range(n) for s in range(m)) +\
            sum_(c(i,j,s,r)*y[i][j][s][r] for i in range(n-1) \
                                          for j in range(i+1,n) \
                                          for s in range(0,m) for r in range(0,m))
        )

        for i in range(n):
            sum_(x[i][s] for s in range(m)) == 1

        for s in range(m):
            sum_(x[i][s] for i in range(n)) <= 1

        for i in range(n-1):
            for j in range(i+1,n):
                sum_(y[i][j][s][r] for s in range(m) for r in range(m)) == 1

        for i in range(n-1):
            for j in range(i+1,n):
                 for s in range(m):
                     for r in range(m):
                         2*y[i][j][s][r] - x[i][s] - x[j][r] <= 0
```

Listing 3.1: MIPCL-PY implementation of process layout problem: `model`

Input parameters:

- `p`: list of size $n \times m$, where `p[i][s]` is cost of moving department `i` to site `s`;
- `dist`: list that represents an upper triangular square matrix of size m, where `dist[s][r-(s+1)]` is distance between sites `s` and `r`, `s < r`;
- `flows`: list that represents an upper triangular square matrix of size n, where `flows[i][j-(i+1)]` is the volume of material moving between departments `i` and `j`, `i < j`.

When a solution has been found, one may want to print it in a readable way. Listing 3.2 presents a procedure — which is a member of `Proclayout` — that does this job.

```
  def printSolution(self):
    x = self.x
    n, m = len(x), len(x[0])

    print('Total␣expenses:␣{0:.4f}'.format(self.getObjVal()))
    print('␣____________________')
    print('|␣Depart.␣|␣␣Site␣␣␣|')
    print('|----------+----------|')
```

```
        for i in range(n):
            for s in range(m):
                if x[i][s].val > 0.5:
                    print('| {0:7d} | {1:7d} |'.format(i+1,s+1))
        print(' ----------------------------')
```

Listing 3.2: MIPCL-PY implementation of process layout problem: `printSolution`

3.1.3 Example of Usage

Consider a low-volume toy factory with eight departments:

1) shipping and receiving,
2) sewing,
3) metal forming,
4) plastic molding and stamping,
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.

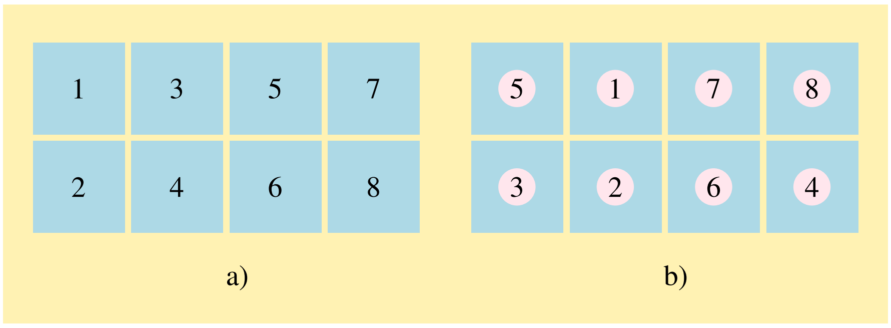

Figure 3.1: Plan of factory building and initial arrangement

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 ($i = 1, \dots, 8$). 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

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

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:

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

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

To solve this instance of the process layout problem, we wrote the program presented in Listing 3.3.

```
#!/usr/bin/python
from proclayout import Proclayout

p = [
    [  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]
]

dist = [
    [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]
]

flows = [
    [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]
]

prob = Proclayout("test1")
prob.model(p,dist ,flows)
prob.optimize(False)
prob.printSolution()
```

Listing 3.3: Example of usage of class `Proclayout`

If we run this program, we get the following result.

```
Total expenses: 18500.0000
 -------------------
| Depart. |   Site   |
|---------+----------|
|       1 |        1 |
|       2 |        2 |
|       3 |        7 |
|       4 |        4 |
|       5 |        6 |
|       6 |        3 |
|       7 |        8 |
|       8 |        5 |
 ___________________
```

Listing 3.4: Output of program from Listing 3.3

We see that, if the factory reallocates its departments as it is prescribed in the above table, 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 operations, necessary to manufacture the product, are performed. The operations in an assembly process usually are interdependent, i.e. there may be precedence requirements that must be enforced. The problem of distributing the operations 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.

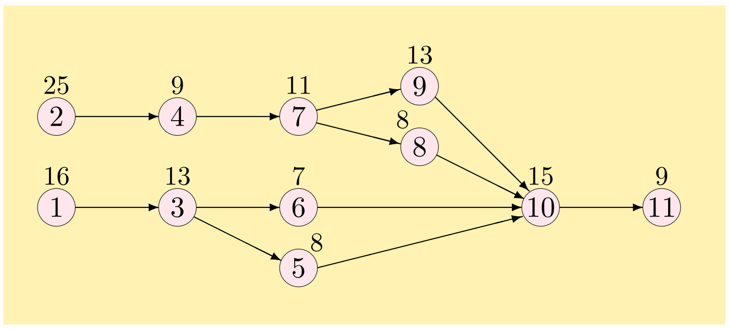

Figure 3.2: Example of SALBP

The manufacturing of some product consists of a set of *operations* $\mathcal{O} = \{1, \dots, n\}$. We denote by t_o the processing time of operation $o \in \mathcal{O}$. There are precedence relations between the operations that are represented by a digraph $G = (\mathcal{O}, E)$, where $(o_1, o_2) \in E$ means that operation o_1 must be finished before operation o_2 starts. 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 the vertices represent operations, and the number over any vertex is the processing time of the corresponding operation.

3.2.1 Integer Programming Formulation

To formulate SALBP as an integer program (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.

To write an IP formulation, we introduce the following variables:

$y_s = 1$ if station s is open (in use), $y_s = 0$ otherwise;

x_{so} if operation o is assigned to station s, $x_{so} = 0$ otherwise;

$z_o = s$ if operation o is assigned to station s.

With these variables the model is

$$\sum_{s=1}^{m} y_s \to \min \tag{3.2a}$$

$$\sum_{s=1}^{m} x_{so} = 1, \quad o = 1,\dots,n, \tag{3.2b}$$

$$\sum_{o=1}^{n} t_o x_{so} \le C y_s, \quad s = 1,\dots,m, \tag{3.2c}$$

$$\sum_{s=1}^{m} s\, x_{so} = z_o, \quad o = 1,\dots,n, \tag{3.2d}$$

$$z_{o_1} \le z_{o_2}, \quad (o_1, o_2) \in E, \tag{3.2e}$$

$$y_{s-1} \ge y_s, \quad s = 2,\dots,m, \tag{3.2f}$$

$$x_{so} \le y_s, \quad o = 1,\dots,n;\ s = 1,\dots,m, \tag{3.2g}$$

$$x_{so} \in \{0,1\}, \quad s = 1,\dots,m;\ o = 1,\dots,n, \tag{3.2h}$$

$$y_s \in \{0,1\}, \quad s = 1,\dots,m, \tag{3.2i}$$

$$z_o \in \mathbb{R}_+, \quad o = 1,\dots,n. \tag{3.2j}$$

The objective (3.2a) is to minimize the number of open stations. The constraints (3.2b) induce that each operation 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 $(o_1, o_2) \in E$ of related operations, operation o_1 is assigned to the same or an earlier station than operation o_2; this guaranties that operation o_1 can be processed before operation o_2 starts. The constraints (3.2f) and (3.2g) enforce that earlier stations are opened first.

3.2.2 MIPCL-PY Implementation

A straightforward **MIPCL-PY** implementation of IP (3.2) is given in Listing 3.5.

```
import mipshell
from mipshell import *

class Assembly(Problem):
    def model(self,m,C,operations):
        def t(o):
            return operations[o][0]
        def prec(o):
            return operations[o][1]

        n = len(operations)
        self.x = x = VarVector([m,n],"x",BIN)
        y, z = VarVector([m],"y",BIN), VarVector([n],"z")

        minimize(sum_(y[s] for s in range(m)))

        for o in range(n):
            sum_(x[s][o] for s in range(m)) == 1
            sum_((s+1)*x[s][o] for s in range(m)) == z[o]
            for o1 in prec(o):
                z[o1] <= z[o]


        for s in range(m):
            sum_(t(o)*x[s][o] for o in range(n)) <= C*y[s]
            for o in range(n):
                x[s][o] <= y[s]

        for s in range(1,m):
            y[s-1] >= y[s]
```

Listing 3.5: MIPCL-PY implementation of SABL: `model`

Input parameters:

- `m`: maximum number of stations (integer);
- `C`: cycle time (integer);
- `operations`: dictionary of operations, which item `operations[o]` is a pair, `(t,prec)`, that describes operation `o`:
  - `t`: processing time (integer);
  - `prec`: tuple (or list) of preceding operations.

When a solution has been found, one may want to print it in a readable format. Listing 3.6 presents a procedure — which is a member of `Assembly` — that does this job.

```
    def printSolution(self):
        x = self.x
        n = len(x[0])

        k = int(self.getObjVal()+0.5)
        print('Number␣of␣stations␣=␣{:d}'.format(k))

        print('Station:␣assigned␣operations')
        for s in range(k):
            str = repr(s) + ':'
            for o in range(n):
                if x[s][o].val > 0.5:
                    str += '␣' + repr(o)
            print(str)
```

Listing 3.6: MIPCL-PY implementation of SABL: `printSolution`

3.2.3 Example of Usage

To test our **MIPCL-PY** model, we wrote a simple program (see Listing 3.7) to balance an assembly line with at most $m = 9$ stations and cycling time $C = 12$, processing times of 17 operations, numbered from 0 to 16, are given by list t, and precedence relations are given by list $prec$.

```
#!/usr/bin/python
from assembly import Assembly

m = 9
C = 12

operations = {
    0: (6, ()),       1: (7, (0,)),      2: (5, ()),
    3: (3, (2,)),     4: (2, ()),        5: (5, (3,)),
    6: (6, (4,)),     7: (3, ()),        8: (4, ()),
    9: (3, ()),      10: (4, (4, 6)),  11: (3, (5,)),
   12: (5, (9,)),    13: (2, (6,)),     14: (6, ()),
   15: (8, (14,)),   16: (3, (3, 7))
}

prob = Assembly("test1")
prob.model(m,C,operations)
prob.optimize(False)
prob.printSolution()
```

Listing 3.7: Example of usage of class `Assembly`

If we run the above program, we get the following result.

```
    Number of stations = 7
    Station: assigned operations
    0: 0 7 9
    1: 4 6 10
    2: 1 12
    3: 13 14
    4: 8 15
    5: 2 3
    6: 5 11 16
```

Listing 3.8: Output of program from Listing 3.7

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 $N = \{1, \dots, n\}$ with b_j customers at location $j \in N$, a set of potential sites $M = \{1, \dots, m\}$ for locating depots, a fixed cost f_i of locating a depot at site i, a capacity u_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.

4.1.1 Integer Programming Formulation

Choosing the following decision variables

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

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

we formulate the FLP as follows:

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

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

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

$$x_{ij} \le \min\{u_i, b_j\} y_i, \quad i = 1, \dots, m,\ j = 1, \dots, n, \tag{4.1d}$$
$$y_i \in \{0, 1\}, \quad i = 1, \dots, m, \tag{4.1e}$$
$$x_{ij} \in \mathbb{Z}_+, \quad i = 1, \dots, m,\ j = 1, \dots, n. \tag{4.1f}$$

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 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.

4.1.2 MIPCL-PY Implementation

Implementation of IP (3.2) is straightforward and it is given in Listing 4.1.

```python
import mipshell
from mipshell import *

class Fl(Problem):
    def model(self,q,b,u,f,c):
        n, m = len(b), len(u)
        self.y = y = VarVector([m],'y',BIN)
        self.x = x = VarVector([m,n],'x',INT)

        minimize(
            sum_(f[i]*y[i] for i in range(m)) + \
            sum_(c[i][j]*x[i][j] for i in range(m) for j in range(n))
        )

        sum_(y[i] for i in range(m)) <= q

        for j in range(n):
            sum_(x[i][j] for i in range(m)) == b[j]

        for i in range(m):
            sum_(x[i][j] for j in range(n)) <= u[i]*y[i]

        for i in range(m):
            for j in range(n):
                x[i][j] <= min(u[i],b[j])*y[i]
```

Listing 4.1: MIPCL-PY implementation of facility location problem: `model`

Input parameters:

- `q`: integer, maximum number of depots;
- `b`: list, where `b[j]` is number of customers at location `j`;
- `u`: list, where `u[i]` is depot capacity at site `i`;
- `f`: list, where `f[i]` is fixed cost of locating depot at site `i`;

- c: list of lists, where c[i][j] is cost of serving from site i one customer from location j.

When a solution has been found, we can print the result calling the procedure printSolution, which is a member of Fl and is presented in Listing 4.2.

```
    def printSolution(self):
        y, x = self.y, self.x
        m, n = len(y), len(x[0])

        print('Objective value: {0:.2f}'.format(self.getObjVal()))
        for i in range(m):
            if y[i].val > 0.5:
                print('\n===== Facility at site {!r} serves customers:'.format(i+1))
                for j in range(n):
                    if x[i][j].val > 0.5:
                        print('\t{:.0f} from loc. {!r}'.format(x[i][j].val,j+1))
```

Listing 4.2: MIPCL-PY implementation of facility location problem: printSolution

4.1.3 Examples of Usage: Locating Two Medical Clinics

Two clinics, both of capacity 40, are to be established to provide medical care for people living in five 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 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.

Table 4.1: Population and distances between communities

Com-munity	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

To solve this example, we prepared the program, presented in Listing 4.3.

```
#!/usr/bin/python
from fl import Fl

q = 2
b = [10, 8, 20, 12, 14]
u = [40, 40, 40, 40, 40]
f = [0, 0, 0, 0, 0]
c = [
    [ 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]
]

prob = Fl("test1")
prob.model(q,b,u,f,c)
prob.optimize()
prob.printSolution()
```

Listing 4.3: Example of usage of class `Fl`

If we run this program, we get the following result.

```
Objective value: 220.00

===== 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
```

Listing 4.4: Output of program from Listing 4.3

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.

4.2.1 Linear Programming Formulation

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

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

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:

$$\sum_{k=1}^{l} s_{i_0 k} u_k \to \max, \tag{4.2a}$$

$$\sum_{j=1}^{m} r_{i_0 j} v_j = 1, \tag{4.2b}$$

$$\sum_{k=1}^{l} s_{ik} u_k \le \sum_{j=1}^{m} r_{ij} v_j, \quad i \in \{1,\dots,n\} \setminus \{i_0\}, \tag{4.2c}$$

$$u_k \ge 0, \quad k = 1,\dots,l, \tag{4.2d}$$

$$v_j \ge 0, \quad j = 1,\dots,m. \tag{4.2e}$$

Let (u^*, v^*) be an optimal solution to problem (4.2). 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.

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

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

$$\begin{aligned} E_1 = 1600u_1 &\to \max, \\ 32v_1 - 3200v_2 &= 1, \\ 400u_1 - 16v_1 - 600v_2 &\le 0, \\ 600u_1 - 24v_1 - 600v_2 &\le 0, \end{aligned}$$

$$\begin{aligned}400u_1 - 24v_1 - 400v_2 &\le 0,\\ 200u_1 - 16v_1 - 160v_2 &\le 0,\\ 80u_1 - 8v_1 - 40v_2 &\le 0,\\ u_1, v_1, v_2 &\ge 0.\end{aligned}$$

4.2.2 MIPCL-PY Implementation

Our implementation of LP (4.2) is given in Listing 4.5.

```
import mipshell
from mipshell import *

class Dea(Problem):
    def model(self,i0,rs):
        def r(i,j):
            return rs[i][0][j]
        def s(i,k):
            return rs[i][1][k]

        self.rs = rs
        n, m, l = len(rs), len(rs[0][0]), len(rs[0][1])

        self.i0 = i0 = i0-1
        self.u = u = VarVector([l],"u")
        self.v = v = VarVector([m],"v")

        maximize(sum_(s(i0,k)*u[k] for k in range(l)))

        sum_(r(i0,j)*v[j] for j in range(m)) == 1

        for i in range(n):
            if i != i0:
                sum_(s(i,k)*u[k] for k in range(l))\
             <= sum_(r(i,j)*v[j] for j in range(m))
```

Listing 4.5: MIPCL-PY implementation of DEA: `model`

Input parameters:

- `i0`: integer, service unit to be estimated;
- `rs`: list, which items are pairs of tuples, where, for one period, service unit `i`
  - uses `rs[i][0][j]` units of resource `j`,
  - and provides `rs[i][1][k]` services of type `k`.

To compute the ratings for all n firms, we need to solve n LPs of type (4.2). Therefore, our **MIPCL-PY** implementation of the DEA problem somewhat differs from most other applications in this manual.

When a solution has been found, we can print the result using the procedure `printSolution`, which is a member of `Dea` and is presented in Listing 4.6.

```
    def printSolution(self):
        def r(i,j):
            return rs[i][0][j]
        def s(i,k):
            return rs[i][1][k]

        u, v, rs = self.u, self.v, self.rs
        n, m, l = len(rs), len(v), len(u)

        print('=>␣Unit␣{!r}␣of␣rating␣{:.4f}'.format(self.i0+1,self.getObjVal()))
        str = 'v␣=␣('
        for j in range(m-1):
            str += '{:.4f},␣'.format(v[j].val)
        print(str + '{:.4f})'.format(v[m-1].val))
        str = 'u␣=␣('
        for k in range(l-1):
            str += '{:.4f},␣'.format(u[k].val)
        print(str + '{:.4f})'.format(u[l-1].val))

        print('␣-------------------------------------')
        print('|␣Unit␣|␣Relative␣|␣␣Weighted␣sum␣of␣␣|')
        print('|␣␣␣␣␣␣|␣␣rating␣␣|␣inputs␣␣|␣outputs␣|')
        print('|------+----------+---------+---------|')
        for i in range(n):
            Ri = 0.0
            for j in range(m):
                Ri += r(i,j)*v[j].val
            Si = 0.0
            for k in range(l):
                Si += s(i,k)*u[k].val
            print('|␣{:4d}␣|␣␣{:6.4f}␣␣|␣{:7.4f}␣|␣{:7.4f}␣|'.format(i+1,Si/Ri,Ri,Si
                ))
        print('␣-------------------------------------')
```

Listing 4.6: MIPCL-PY implementation of DEA: `printSolution`

4.2.3 Example of Usage

A car manufacturer wants to evaluate the efficiency 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 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.

The program from Listing 4.7 computes the efficiency rating of Garage 1.

```
#!/usr/bin/python
from dea import Dea

rs = (
  (( 3.0,  3.6,   3.0,  3.0,  2.5,  1.5), (0.8,  0.2,  0.45)),
  (( 6.0,  7.5,  10.0, 10.0,  7.5,  4.0), (1.5,  0.45, 0.45)),
  (( 7.0,  6.0,   5.0,  7.0,  8.5,  4.5), (1.2,  0.48, 2.0)),
  (( 7.0,  8.0,   7.0,  8.0,  3.0,  2.0), (1.9,  0.7,  0.5)),
  ((14.0,  9.0,  20.0, 25.0, 10.0,  6.0), (2.6,  0.86, 1.9)),
  ((10.0,  9.0,  10.0, 10.0, 11.0,  5.0), (2.4,  1.0,  2.0)),
  (( 3.0,  3.5,   3.0, 20.0,  2.0,  1.5), (0.9,  0.35, 0.5)),
```

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

```
  ((12.0,  8.0,   7.0, 10.0, 12.0,  7.0), (4.5,  2.0,  2.3)),
  (( 5.0,  5.0,  10.0, 10.0,  5.0,  2.5), (2.0,  0.65, 0.9)),
  (( 8.0, 10.0,  30.0, 35.0,  9.5,  4.5), (2.05, 0.75, 1.7)),
  (( 2.0,  3.0,  40.0, 40.0,  2.0,  1.5), (0.8,  0.25, 0.5)),
  (( 5.0,  6.5,   9.0, 12.0,  8.0,  4.5), (1.8,  0.63, 1.4)),
  (( 7.0,  8.0,  10.0, 12.0,  8.5,  4.0), (2.0,  0.6,  1.5)),
  ((11.0,  8.0,   8.0, 10.0, 10.0,  6.0), (2.2,  0.65, 2.2)),
  (( 4.0,  5.0,  10.0, 10.0,  7.5,  3.5), (1.8,  0.62, 1.6)),
  ((24.0, 15.0,  15.0, 13.0, 25.0,  1.9), (8.0,  2.6,  4.5)),
  ((30.0, 29.0, 120.0, 80.0, 35.0, 20.0), (7.0,  2.5,  8.0)),
  (( 4.0,  6.0,   1.0,  1.0,  7.5,  3.5), (1.1,  0.45, 1.7)),
  (( 6.0,  5.5,   2.0,  2.0,  8.0,  5.0), (1.5,  0.55, 1.55)),
  (( 8.0,  7.5,   5.0,  8.0,  9.0,  4.0), (2.1,  0.85, 2.0)),
  (( 5.0,  5.5,   8.0, 10.0,  7.0,  3.5), (1.2,  0.45, 1.3)),
  ((25.0, 16.0, 110.0, 80.0, 27.0, 12.0), (6.5,  3.5,  5.4)),
  ((19.0, 10.0,  90.0, 22.0, 25.0, 13.0), (5.5,  3.1,  4.5)),
  (( 6.0,  6.0,  20.0, 30.0,  9.0,  4.5), (2.3,  0.7,  1.6)),
  (( 6.0,  7.0,  50.0, 40.0,  8.5,  3.5), (2.5,  0.9,  1.6)),
  ((21.0, 12.0,   6.0,  6.0, 15.0,  8.0), (6.0,  0.25, 2.9))
)

prob = Dea("garages")
prob.model(1,rs)
prob.optimize()
prob.printSolution()
```

Listing 4.7: Example of usage of class `Dea`

Running this program, we get the following result.

```
=> Unit 1 of rating 0.8888
v = (0.0181, 0.0000, 0.0000, 0.0106, 0.3656, -0.0000)
u = (0.2944, 0.0474, 1.4306)

 -----------------------------------
| Unit | Relative |  Weighted sum of  |
|      |  rating  |  inputs | outputs |
|------+----------+---------+---------|
|    1 |  0.8888  |  1.0000 |  0.8888 |
|    2 |  0.3744  |  2.9564 |  1.1068 |
|    3 |  0.9785  |  3.3084 |  3.2373 |
|    4 |  1.0000  |  1.3079 |  1.3079 |
|    5 |  0.8445  |  4.1735 |  3.5245 |
|    6 |  0.8391  |  4.3084 |  3.6153 |
|    7 |  1.0000  |  0.9969 |  0.9969 |
|    8 |  1.0000  |  4.7102 |  4.7102 |
|    9 |  0.9422  |  2.0242 |  1.9072 |
|   10 |  0.7701  |  3.9881 |  3.0712 |
|   11 |  0.8088  |  1.1903 |  0.9627 |
|   12 |  0.8155  |  3.1423 |  2.5627 |
|   13 |  0.8221  |  3.3612 |  2.7632 |
|   14 |  0.9660  |  3.9608 |  3.8259 |
|   15 |  0.9754  |  2.9203 |  2.8484 |
|   16 |  0.9181  |  9.7120 |  8.9165 |
|   17 |  0.9605  | 14.1850 | 13.6245 |
|   18 |  0.9831  |  2.8252 |  2.7773 |
|   19 |  0.8790  |  3.0547 |  2.6852 |
|   20 |  1.0000  |  3.5198 |  3.5198 |
|    21 |   0.8109   |   2.7555  |   2.2345  |
|    22 |   0.8778   |  11.1696  |   9.8050  |
|    23 |   0.8443   |   9.7168  |   8.2041  |
|    24 |   0.8071   |   3.7163  |   2.9994  |
|    25 |   0.8430   |   3.6392  |   3.0677  |
|    26 |   1.0000   |   5.9273  |   5.9273  |
 ---------------------------------------------
```

Listing 4.8: Output of program from Listing 4.7

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.

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 demand 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 one of K planes, the seats in all planes are divided into the same number, I, of classes. Plane k ($k = 1, \dots, K$) costs f_k to hire, and has q_{ki} of seats of class i ($i = 1, \dots, I$). For example, plane k may have $q_{k,1} = 30$ first class seats, $q_{k,2} = 40$ business class seats, and $q_{k,3} = 60$ economy class seats. In plane k, up to r^l_{ki} and r^h_{ki} seats of class i can be transformed into seats of lower, $i-1$, and higher, $i+1$, classes, $i = 1, \dots, I$. It is assumed that $t^l_{k,1} = 0$ and $t^h_{kI} = 0$.

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

Demand is uncertain but it 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.

Let us also note that period t starts at time $t-1$ and ends at time t. Therefore, it is assumed that the decision which option to use in period t is made at time $t-1$. The other decision how many seats of each class to sell depends on the demand in this period; therefore, this decision is assumed to be made at time t (the end of period t).

4.3.2 Mixed-Integer Programming Formulation

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

Each node $j \in V_t$ ($t = 1, \dots, T$) corresponds to one of the *histories*, $h(j) = (s_1, s_2, \dots, s_t)$, that may happen after t periods, where s_τ is an index of a scenario for period τ. By definition, the history of the root node is empty, $h(0) = ()$. History $h(j)$ happens with probability $\bar{p}_j \overset{\text{def}}{=} \prod_{\tau=1}^{t} p_{\tau,s_\tau}$, $\bar{p}_0 \overset{\text{def}}{=} 1$. If price option o is used, the demand for seats of class i is $\bar{d}_{joi} \overset{\text{def}}{=} d_{t,s_t,o,i}$, and their price is c_{toi}. Let us define $\bar{c}_{joi} \overset{\text{def}}{=} \bar{p}_j c_{toi}$. The *parent* of node j, denoted by $parent(j)$, is that node in V_{t-1} which history is $(s_1, s_2, \dots, s_{t-1})$, i.e, $h(j) = (h(parent(j)), s_t)$. Note, that the root node 0 is the parent of all nodes in V_1 (of level 1).

Now we define the variables.

The first family of binary variables decides which plane to use. For each plane k we define

$v_k = 1$ if plane k is used, and $v_k =$ otherwise.

One we hired the plane, we have to decide how to transform seats in that plane. So, we define two families of integer valued variables:

w_i^l: number of seats of class i ($i = 2, \dots, I$) to transform into seats of class $i-1$;

w_i^h: number of seats of class i ($i = 1, \dots, I-1$) to transform into seats of class $i+1$.

With each node $j \in V \setminus V_T$ of the scenario tree we associate the following decision variables:

$y_{jo} = 1$ if price option o is used in period $t+1$ when history $h(j)$ happens, and $y_{jo} = 0$ otherwise.

We also introduce two families of auxiliary variables. For each node $j \in V \setminus \{0\}$ we use the following variables:

x_{joi}: number of seats of class i to be sold in period t using price option o, when history $h(j)$ for $j \in V_t$ happens.

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

z_{ji}: total number of seats of class i to be sold when history $h(j)$ happens.

Now we can write the following deterministic model:

$$-\sum_{k=1}^{K} f_k v_k + \sum_{j\in V\setminus V_T} \sum_{o=1}^{O}\sum_{i=1}^{I} \bar{c}_{joi} x_{joi} \to \max, \tag{4.3a}$$

$$\sum_{k=1}^{K} v_k = 1, \tag{4.3b}$$

$$u_i = \sum_{k=1}^{K} q_{ki} v_k, \quad i = 1,\dots,I, \tag{4.3c}$$

$$w_i^l \le \sum_{k=1}^{K} t_{ki}^l v_k, \quad i = 1,\dots,I, \tag{4.3d}$$

$$w_i^h \le \sum_{k=1}^{K} t_{ki}^h v_k, \quad i = 1,\dots,I, \tag{4.3e}$$

$$\sum_{o=1}^{O} y_{jo} = 1, \quad j \in V \setminus V_T, \tag{4.3f}$$

$$x_{joi} \le \bar{d}_{jio}\, y_{parent(j),o}, \quad j \in V\setminus\{0\},\ i = 1,\dots,I,\ o = 1,\dots,O, \tag{4.3g}$$

$$z_{ji} = z_{parent(j),i} + \sum_{o=1}^{O} x_{joi}\,, \quad i = 1,\dots,I,\ j \in V\setminus\{0\}, \tag{4.3h}$$

$$z_{j,1} + w_1^h \le u_1 + w_2^l\,, \quad j \in V_T, \tag{4.3i}$$

$$z_{ji} + w_i^l + w_i^h \le u_i + w_{i-1}^h + w_{i+1}^l\,, \quad i = 2,\dots,I-1,\ j \in V_T, \tag{4.3j}$$

$$z_{jI} + w_I^l \le u_I + w_{I-1}^h\,, \quad j \in V_T, \tag{4.3k}$$

$$x_{joi} \in \mathbb{Z}_+, \quad j \in V\setminus\{0\},\ o = 1,\dots,O,\ i = 1,\dots,I, \tag{4.3l}$$

$$y_{jo} \in \{0,1\}, \quad j \in V\setminus V_T,\ o = 1,\dots,O, \tag{4.3m}$$

$$z_{ji} \in \mathbb{Z}_+, \quad j \in V,\ i = 1,\dots,I, \tag{4.3n}$$

$$z_{0i} = 0, \quad i = 1,\dots,I, \tag{4.3o}$$

$$v_k \in \mathbb{Z}_+\,, \quad k = 1,\dots,K, \tag{4.3p}$$

$$u_i, w_i^l, w_i^h \in \mathbb{Z}_+, \quad i = 1,\dots,I. \tag{4.3q}$$

The objective (4.3a) is to maximize the profit from selling seats minus the expenses for hiring planes. The equality (4.3b) prescribes to hire just one plane. The inequalities (4.3d) and (4.3e) restrict the number of seats in any class that can be transformed into seats of the lower and higher classes. The next family of equalities, (4.3c), determines the capacities of all classes in the hired plane. The equations (4.3f) guarantee that in any of T periods only one price option is chosen for each class. The variable upper bounds (4.3g) 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 (4.3h) count the total number of seats for each class that are sold in any of T periods. The inequalities (4.3i)–(4.3k) require that, for each class, the number of sold seats plus the number of seats transformed into seats of adjacent classes does not exceed the total number of seats of that class plus the number of seats transformed from seats of adjacent classes.

When the problem (4.3) is solved, we know which options to use and how many seats of each class to sell 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, which will be period 1 in the new model. How to build this new model will be explained in details when we will discuss a numeric example in Section 4.3.4. This procedure is then repeated for periods $t = 3, \dots, T$.

4.3.3 MIPCL-PY Implementation

Our **MIPCL-PY** implementation of IP (4.3) is given in Listing 4.9.

```
import mipshell
from mipshell import *

class Revenue(Problem):
    def __init__(self,name,planes,options,nodes):
        Problem.__init__(self,name)
        self.planes, self.options, self.nodes = planes, options, nodes
        self.processData()

    def processData(self):
        nodes = self.nodes
        n0, n, T = -1, len(nodes), len(self.options),
        for j in range(1,n):
            nodes[j]['prob'] *= nodes[nodes[j]['parent']]['prob']
            if n0 < 0:
                if nodes[j]['period'] == T:
                    n0 = j
        self.n0 = n0 # number of internal nodes (not leaves)

    def model(self,planes,options,nodes):
        def c(j,o,i): return nodes[j][3]*options[nodes[j][1]-1][o][i]
        def d(j,i,o): return nodes[j][o+4][i]
        def parent(j): return nodes[j][2]
        def q(k,i): return planes[k]['classes'][i][0]
        def tl(k,i): return planes[k]['classes'][i][1]
        def th(k,i): return planes[k]['classes'][i][2]
        def f(k): return planes[k][1]

        self.planes, self.options, self.nodes = planes, options, nodes
        K, T, O, I = len(planes), len(options), len(options[0]), len(options[0][0])
        n, n0 = len(nodes), self.n0

        self.v = v = VarVector([K],"v",BIN)
        u = VarVector([I],"u",INT)
        wl, wh = VarVector([I],"wl",INT), VarVector([I],"wh",INT)
        x = VarVector([n,O,I],"x",INT) # x[0] is not used
        self.y = y = VarVector([n0,O],"y",BIN)
        z = VarVector([n,I],"z",INT)

        maximize(
              sum_(c(j,o,i)*x[j][o][i] for j in range(1,n)\
                                       for o in range(O) for i in range(I))\
            - sum_(f(k)*v[k] for k in range(K))\
        )

        sum_(v[k] for k in range(K)) == 1
        for i in range(I):
            u[i] == sum_(q(k,i)*v[k] for k in range(K))
            wl[i] <= sum_(tl(k,i)*v[k] for k in range(K))
            wh[i] <= sum_(th(k,i)*v[k] for k in range(K))

        for j in range(n0):
            sum_(y[j][o] for o in range(O)) == 1;

        for j in range(1,n):
            for o in range(O):
                for i in range(I):
                    x[j][o][i] <= d(j,o,i)*y[parent(j)][o]

        for i in range(I): z[0][i] == 0

        for j in range(1,n):
            for i in range(I):
                z[j][i] == z[parent(j)][i] + sum_(x[j][o][i] for o in range(O))

        for j in range(n0,n):
            z[j][0] + wh[0] <= u[0] + wl[1]
            z[j][I-1] + wl[I-1] <= u[I-1] + wh[I-2]
            for i in range(1,I-1):
                z[j][i] + wl[i] + wh[i] <= u[i] + wh[i-1] + wl[i+1]
```

Listing 4.9: MIPCL-PY implementation of yield management problem: model

The constructor

The constructor, `__init__()`, has four arguments (except `self`). The first one, `name`, is the name of a problem to be solved. The other three arguments describe a problem instance:

- `planes`: list of plane types, where `planes[k]` describes planes of type `k`:
  - `f(k)=planes[k]['fixedCost']`: cost to hire;
  - `q(k,i)=planes[k]['classes'][i][0]`: numbers of seats in class `i`;
  - `tl(k,i)=planes[k]['classes'][0][1]`: numbers of seats of class `i` that can be transformed into seats of class `i-1`;
  - `tl(k,i)=planes[k]['classes'][0][2]`: numbers of seats of class `i` that can be transformed into seats of class `i+1`;
- `options`: list of options, where `options[t]` is a list (or tuple) of 3 options for period `t`, where
  - `options[t][o]`: 3-tuple that represents ticket prices for each of 3 classes if option `o` is used;
- `nodes`: list of nodes of a scenario tree, each node j, except for node 0, is characterized by a list (or tuple) nodes[j] which is structured as follows:

- `nodes[j]['name']`: node name;
- `nodes[j]['period']`: period;
- `nodes[j]['parent']`: index of the parent node;
- `nodes[j]['prob']`: probability of reaching this node from its parent;
- `nodes[j]['demands'][0]`: 3-tuple, demands for tickets for each of three classes if Option 1 is applied;
- `nodes[j]['demands'][1]`: 3-tuple, demands for tickets for each of three classes if Option 2 is applied;
- `nodes[j]['demands'][2]`: 3-tuple, demands for tickets for each of three classes if Option 3 is applied.

The constructor invokes the constructor of the base class, stores input arguments in the fields of the created `Revenue` object, and then calls an auxiliary function, `processData()`, to

- computes probabilities of reaching all non-root scenario-tree nodes (the probability of reaching node `j` is the probability of reaching its parent node `parent[j]` multiplied by the probability of moving along the edge `(parent[j],j)`);
- determines the number, `n0`, of internal nodes (not leaves).

Member Functions

The crucial function in the `Problem` class is `model()`, which implements MIP (4.3). To make the implementation straightforwars, we define a number of local functions that map input parameters onto the parameters of MIP (4.3).

When a solution has been found, we can print the result using the procedure `printSolution` — which is a member of `Revenue` — from Listing 4.10.

```
def printSolution(self):
    def parent(j): return self.nodes[j]['parent']
    def name(k,i): return self.planes[k]['classes'][i]['name']

    K, O, I = len(self.planes), len(self.options[0]), len(self.options[0][0])
    x, y, v = self.x, self.y, self.v

    k0, o0 = 0, 0
    for k in range(K):
        if v[k].val > 0.5:
            k0 = k
            break
    for o in range(O):
        if y[0][o].val > 0.5:
            o0 = o
            break

    print('Optimal␣objective␣value:␣{:.4f}'.format(self.getObjVal()))
    print('Plane␣to␣hire:␣{!r}'.format(k0+1))
    print('\nIn␣Period~1␣use␣price␣option␣{!r},␣and␣sell:\n'.format(o+1))
    for j in range(1,self.n0):
        if parent(j) != 0: break
```

```
            print('\tif␣scenario␣{!r}␣happens:\n'.format(j))
            for i in range(I):
                print('\t\t{!r}␣tickets␣of␣{!s}␣class\n'.\
                      format((int)(x[j][o][i].val),name(k0,i)))
```

Listing 4.10: MIPCL-PY implementation of yield management problem: auxiliary procedures

4.3.4 Example of Usage

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

- 35 first class seats, 12 of which can be transformed in seats of business class;
- 40 business class seats, 12 of which can be transformed in seats of first class and other 6 — into seats of economy class;
- 80 economy class seats, 12 of which can be transformed into seats of business class.

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.

Model (4.3) does not allow us to hire more than one plane. Nevertheless, we can still use it assuming that we have three virtual planes and plane k has capacities of k original planes, the cost of hire such a plane is $k \cdot \$75000$ ($k = 1, 2, 3$).

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: 0.1 for Scenario 1, 0.6 for Scenario 2, 0.3 for Scenario 3. The forecast demands are shown in Table 4.5.

To solve our example, we prepared the program, presented in Listing 4.11.

Table 4.4: Price Options

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

Table 4.5: Forecast Demands

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

```
#!/usr/bin/python
from mipcl_py.models.revenue import Revenue

planes = (
  {
    'fixedCost': 75000,
    'classes': (
       {'name': 'first', 'capacity': (30,0,12)},
       {'name': 'busines', 'capacity': (40,12,6)},
       {'name': 'economy', 'capacity': (80,12,0)}
    )
  },
  {
    'fixedCost': 150000,
    'classes': (
       {'name': 'first', 'capacity': (60,0,24)},
       {'name': 'busines', 'capacity': (80,24,12)},
       {'name': 'economy', 'capacity': (160,24,0)}
    )
  },
  {
    'fixedCost': 225000,
    'classes': (
       {'name': 'first', 'capacity': (90,0,36)},
       {'name': 'business', 'capacity': (120,36,18)},
       {'name': 'economy', 'capacity': (240,36,0)}
    )
  }
)

options = (
  ((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))
)

nodes = (
  {'name':  0, 'period': 0, 'parent': -1, 'prob': 1.0},
  {'name':  1, 'period': 1, 'parent': 0,  'prob': 0.1, 'demands': (( 25,  40, 100),
      ( 30,  50, 120), ( 35,  70, 130))},
  {'name':  2, 'period': 1, 'parent': 0,  'prob': 0.6, 'demands': (( 40,  90, 100),
      ( 50,  85, 100), ( 70,  90, 140))},
  {'name':  3, 'period': 1, 'parent': 0,  'prob': 0.3, 'demands': (( 90,  90, 110),
      (100,  95, 105), (120,  95, 135))},
  {'name':  4, 'period': 2, 'parent': 1,  'prob': 0.1, 'demands': (( 40,  85, 100),
      ( 50,  90, 105), ( 65,  85, 125))},
  {'name':  5, 'period': 2, 'parent': 1,  'prob': 0.6, 'demands': (( 20, 100, 120),
      ( 80, 120, 130), (100, 160, 180))},
  {'name':  6, 'period': 2, 'parent': 1,  'prob': 0.3, 'demands': ((100,  40,  25),
      (110,  60,  80), (160, 100, 120))},
  {'name':  7, 'period': 2, 'parent': 2,  'prob': 0.1, 'demands': (( 40,  85, 100),
      ( 50,  90, 105), ( 65,  85, 125))},
  {'name':  8, 'period': 2, 'parent': 2,  'prob': 0.6, 'demands': (( 20, 100, 120),
      ( 80, 120, 130), (100, 160, 180))},
  {'name':  9, 'period': 2, 'parent': 2,  'prob': 0.3, 'demands': ((100,  40,  25),
      (110,  60,  80), (160, 100, 120))},
  {'name': 10, 'period': 2, 'parent': 3,  'prob': 0.1, 'demands': (( 40,  85, 100),
     ( 50,  90, 105), ( 65,  85, 125))},
  {'name': 11, 'period': 2, 'parent': 3,  'prob': 0.6, 'demands': (( 20, 100, 120),
     ( 80, 120, 130), (100, 160, 180))},
  {'name': 12, 'period': 2, 'parent': 3,  'prob': 0.3, 'demands': ((100,  40,  25),
     (110,  60,  80), (160, 100, 120))},
  {'name': 13, 'period': 3, 'parent': 4,  'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 14, 'period': 3, 'parent': 4,  'prob': 0.6, 'demands': (( 60,  20, 100),
     ( 80,  80, 120), (120,  90, 140))},
  {'name': 15, 'period': 3, 'parent': 4,  'prob': 0.3, 'demands': ((100,  80, 120),
     (140,  90, 130), (160, 120, 140))},
  {'name': 16, 'period': 3, 'parent': 5,  'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 17, 'period': 3, 'parent': 5,  'prob': 0.6, 'demands': (( 60,  20, 100),
     ( 80,  80, 120), (120,  90, 140))},
  {'name': 18, 'period': 3, 'parent': 5,  'prob': 0.3, 'demands': ((100,  80, 120),
     (140,  90, 130), (160, 120, 140))},
  {'name': 19, 'period': 3, 'parent': 6,  'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 20, 'period': 3, 'parent': 6,  'prob': 0.6, 'demands': (( 60,  20, 100),
     ( 80,  80, 120), (120,  90, 140))},
  {'name': 21, 'period': 3, 'parent': 6,  'prob': 0.3, 'demands': ((100,  80, 120),
     (140,  90, 130), (160, 120, 140))},
  {'name': 22, 'period': 3, 'parent': 7,  'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 23, 'period': 3, 'parent': 7,  'prob': 0.6, 'demands': (( 60,  20, 100),
     ( 80,  80, 120), (120,  90, 140))},
  {'name': 24, 'period': 3, 'parent': 7,  'prob': 0.3, 'demands': ((100,  80, 120),
     (140,  90, 130), (160, 120, 140))},
  {'name': 25, 'period': 3, 'parent': 8,  'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 26, 'period': 3, 'parent': 8,  'prob': 0.6, 'demands': (( 60,  20, 100),
     ( 80,  80, 120), (120,  90, 140))},
  {'name': 27, 'period': 3, 'parent': 8,  'prob': 0.3, 'demands': ((100,  80, 120),
     (140,  90, 130), (160, 120, 140))},
  {'name': 28, 'period': 3, 'parent': 9,  'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 29, 'period': 3, 'parent': 9,  'prob': 0.6, 'demands': (( 60,  20, 100),
     ( 80,  80, 120), (120,  90, 140))},
  {'name': 30, 'period': 3, 'parent': 9,  'prob': 0.3, 'demands': ((100,  80, 120),
     (140,  90, 130), (160, 120, 140))},
  {'name': 31, 'period': 3, 'parent': 10, 'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 32, 'period': 3, 'parent': 10, 'prob': 0.6, 'demands': (( 60,  20, 100),
     ( 80,  80, 120), (120,  90, 140))},
  {'name': 33, 'period': 3, 'parent': 10, 'prob': 0.3, 'demands': ((100,  80, 120),
     (140,  90, 130), (160, 120, 140))},
  {'name': 34, 'period': 3, 'parent': 11, 'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 35, 'period': 3, 'parent': 11, 'prob': 0.6, 'demands': (( 60,  20, 100),
     ( 80,  80, 120), (120,  90, 140))},
  {'name': 36, 'period': 3, 'parent': 11, 'prob': 0.3, 'demands': ((100,  80, 120),
     (140,  90, 130), (160, 120, 140))},
  {'name': 37, 'period': 3, 'parent': 12, 'prob': 0.1, 'demands': (( 60,  80, 100),
     ( 70, 100, 120), ( 80, 110, 160))},
  {'name': 38, 'period': 3, 'parent': 12, 'prob': 0.6, 'demands': (( 60,  20, 100),
      ( 80,  80, 120), (120,  90, 140))},
  {'name': 39, 'period': 3, 'parent': 12, 'prob': 0.1, 'demands': ((100,  80, 120),
      (140,  90, 130), (160, 120, 140))}
)

prob = Revenue('test1',planes,options,nodes)
prob.model()
prob.optimize()
prob.printSolution()
```

Listing 4.11: Example of usage of class Revenue

If we run this program, we get the answer shown in Listing 4.12.

```
Plane to hire: 3
In Period 1 use price option 1, and sell:
    if scenario 1 happens:
        25 tickets of first class
        30 tickets of business class
        35 tickets of economy class
    if scenario 2 happens:
        40 tickets of first class
        40 tickets of business class
        44 tickets of economy class
    if scenario 3 happens:
        46 tickets of first class
        40 tickets of business class
        44 tickets of economy class
```

Listing 4.12: Output of program from Listing 4.11

This solution suggests to hire three planes, and use option 1 in the first week. The latter means that the airline assigns prices \$1500, \$1000, \$500 for the, respectively, first, business, and economy class tickets.

One week later, the airline managers will decide which option to apply in the second week when it will be known how many tickets of all classes have been sold during the first week. To determine an optimal option for the second week, one can formulate and solve a similar planning problem but only with two-period's horizon (2-nd and 3-rd weeks) and with remaining numbers of seats in three planes. Therefore, IP (4.3), although looking as being dynamic, in fact is not dynamic. They say that such models are *applied dynamically*.

Chapter 5

Logistics management

”Logistics is about getting the right product, to the right customer, in the right quantity, in the right condition, at the right place, at the right time, and at the right cost (the seven Rs of Logistics)” - from Supply Chain Management: A Logistics Perspective By John J. Coyle et al.

Traditionally, logistics focuses on activities such as procurement, inventory management, and distribution. It is defined as follows.

Logistics is the . . .
“process of planning, implementing, and controlling the efficient, effective flow and storage of goods, services, and related information from point of origin to point of consumption for the purpose of conforming to customer requirements.“

Council of Logistics Management

Transport logistics deals with air/sea/land transportation and warehousing.

5.1 Fixed Charge Network Flows

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, or optimization of supply chains.

An instance of FCNF problem is given in Figure 5.1, where the node indices 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 $c_e = 0$.

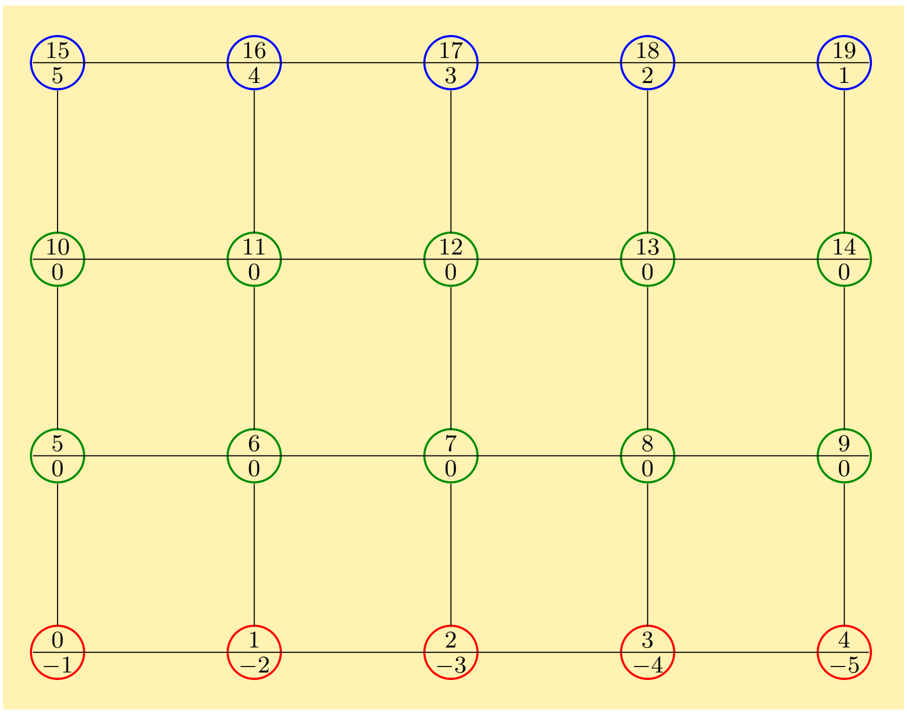

Figure 5.1: An instance of FCNF

5.1.1 Mixed-Integer Programming Formulation

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:

$$\sum_{e \in E} (f_e y_e + c_e x_e) \to \min, \tag{5.1a}$$

$$\sum_{e \in E(V,v)} x_e - \sum_{e \in E(v,V)} x_e = d_v, \quad v \in V, \tag{5.1b}$$

$$0 \le x_e \le u_e y_e, \quad e \in E, \tag{5.1c}$$

$$y_e \in \{0,1\}, \quad e \in E. \tag{5.1d}$$

Here the objective (5.1a) is to minimize transportation expenses. The *balance equations* (5.1b) require that the number of flow units entering each particular node equals the number of flow units leaving the node. The *variable upper bounds* (5.1c) 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 ($x_e = 0$).

5.1.2 MIPCL-PY Implementation

A straightforward MIPCL-PY implementation of MIP (5.1)) is given in Listing 5.1.

```
import mipshell
from mipshell import *

def t(e): return e[0]
def h(e): return e[1]
def u(e): return e[2]
def c(e): return e[3]
def f(e): return e[4]

class Fcnf(Problem):
    def model(self,G):
        self.G = G
        d, E = G['Demands'], G['Arcs']
        m, n = len(E), len(d)

        self.x = x = VarVector([m],"x")
        y = VarVector([m],"y",BIN)

        minimize(sum_(f(e)*y[j] + c(e)*x[j] for j,e in enumerate(E)))

        for v in range(n):
            sum_(x[j] for j,e in enumerate(E) if  h(e)==v)\
          - sum_(x[j] for j,e in enumerate(E) if  t(e)==v) == d[v]

        for j,e in enumerate(E):
            x[j] <= u(e)*y[j]
```

Listing 5.1: MIPCL-PY implementation of FCNF problem: `model`

Input parameters:

- `G`: flow network:
  - `G['Demands']`: list of demands at nodes;
  - `G['Arcs']`: list of arcs, where arc `e = G['Arcs'][j]` is represented as a tuple of five integers:
    - `e[0]`: tail;
    - `e[1]`: head;
    - `e[2]`: capacity;
    - `e[3]`: fixed cost;
    - `e[4]`: cost.

When a solution has been found, we can print the result calling `printSolution()` — a member of `Fcnf` —, which is shown in Listing 5.2.

```
    def printSolution(self):
        if self.is_solution is not None:
            if self.is_solution == True:
                d, E = self.G['Demands'], self.G['Arcs']
                x = self.x
                print('Flow cost = {:d}'.format(int(self.getObjVal() + 0.5)))
                for j,e in enumerate(E):
                    if x[j].val > 0.5:
                        print('flow({:d},{:d}) = {:d}'.format(\
                                     t(e),h(e),int(x[j].val + 0.5)))
            else:
                print('Problem has no solution!')
        else:
            print('Please run optimize first')
```

Listing 5.2: MIPCL-PY implementation of FCNF problem: `printSolution`

5.1.3 Example of Usage

To test class `Fcnf`, we wrote the following program that solves our FCNF-instance from Figure 5.1.

```
#!/usr/bin/python
from fcnf import Fcnf

G = {
    'Demands': [-1, -2, -3, -4, -5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 4, 3, 2, 1],
    'Arcs': [
        (0, 1, 3, 1, 0), (1, 0, 3, 1, 0), (1, 2, 3, 1, 0), (2, 1, 3, 1, 0),
        (2, 3, 3, 1, 0), (3, 2, 3, 1, 0), (3, 4, 3, 1, 0), (4, 3, 3, 1, 0),
        (5, 6, 3, 1, 0), (6, 5, 3, 1, 0), (6, 7, 3, 1, 0), (7, 6, 3, 1, 0),
        (7, 8, 3, 1, 0), (8, 7, 3, 1, 0), (8, 9, 3, 1, 0), (9, 8, 3, 1, 0),
        (10, 11, 3, 1, 0), (11, 10, 3, 1, 0), (11, 12, 3, 1, 0), (12, 11, 3, 1, 0),
        (12, 13, 3, 1, 0), (13, 12, 3, 1, 0), (13, 14, 3, 1, 0), (14, 13, 3, 1, 0),
        (15, 16, 3, 1, 0), (16, 15, 3, 1, 0), (16, 17, 3, 1, 0), (17, 16, 3, 1, 0),
        (17, 18, 3, 1, 0), (18, 17, 3, 1, 0), (18, 19, 3, 1, 0), (19, 18, 3, 1, 0),
        (0, 5, 4, 2, 0), (5, 0, 4, 2, 0), (1, 6, 4, 2, 0), (6, 1, 4, 2, 0),
        (2, 7, 4, 2, 0), (7, 2, 4, 2, 0), (3, 8, 4, 2, 0), (8, 3, 4, 2, 0),
        (4, 9, 4, 2, 0), (9, 4, 4, 2, 0), (5, 10, 4, 2, 0), (10, 5, 4, 2, 0),
        (6, 11, 4, 2, 0), (11, 6, 4, 2, 0), (7, 12, 4, 2, 0), (12, 7, 4, 2, 0),
        (8, 13, 4, 2, 0), (13, 8, 4, 2, 0), (9, 14, 4, 2, 0), (14, 9, 4, 2, 0),
        (10, 15, 4, 2, 0), (15, 10, 4, 2, 0), (11, 16, 4, 2, 0), (16, 11, 4, 2, 0),
        (12, 17, 4, 2, 0), (17, 12, 4, 2, 0), (13, 18, 4, 2, 0), (18, 13, 4, 2, 0),
        (14, 19, 4, 2, 0), (19, 14, 4, 2, 0)
    ]
}

prob = Fcnf("gridNet")
prob.model(G)
prob.optimize(False)
prob.printSolution()
```

Listing 5.3: Example of usage of class `Fcnf`

If we run the program from Listing 5.3, it prints the following result.

```
Flow cost = 34

flow(1,0) = 3, flow(2,1) = 1, flow(3,2) = 2, flow(4,3) = 1,
flow(12,11) = 3, flow(13,12) = 3, flow(16,15) = 1, flow(17,16) = 2,
flow(18,17) = 1, flow(19,18) = 3, flow(0,5) = 4, flow(2,7) = 4,
flow(3,8) = 3, flow(4,9) = 4, flow(5,10) = 4, flow(7,12) = 4,
flow(8,13) = 3, flow(9,14) = 4, flow(10,15) = 4, flow(11,16) = 3,
flow(12,17) = 4, flow(14,19) = 4
```

Listing 5.4: Output of program from Listing 2.7

A graphical representation of the result from Listing 5.4 is given in Figure 5.2.

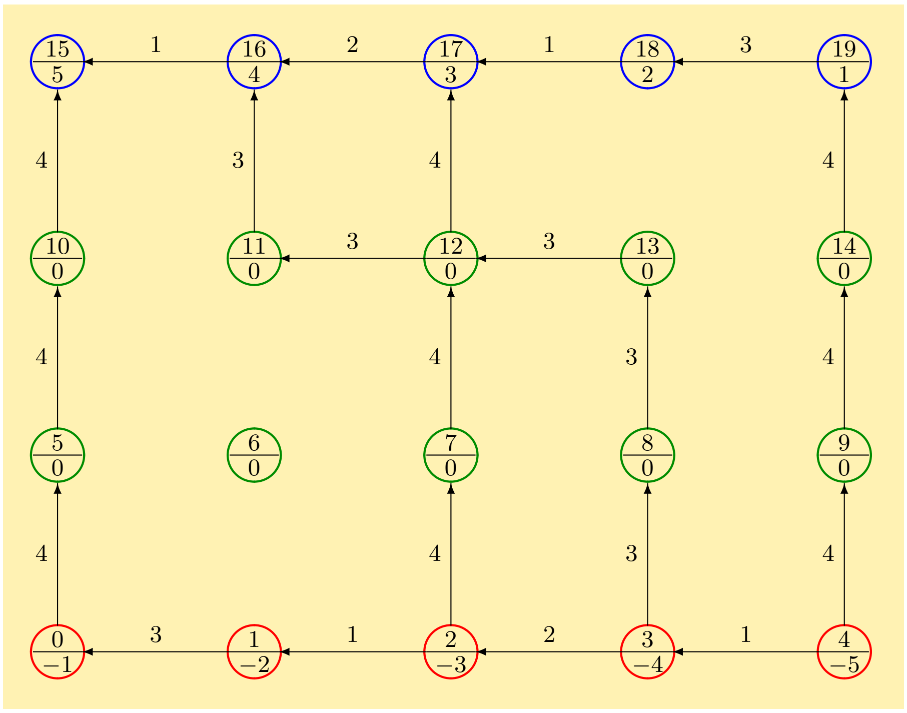

Figure 5.2: Solution to FCNF-instance from Figure 5.1

5.2 Single Depot Vehicle Routing problem

There is a depot that supplies customers with some goods. This depot has a fleet of vehicles of K different types. There are q_k vehicles of type k, and each such a vehicle is of *capacity* u_k (maximum weight to carry). We also know the *fixed cost*, f_k, of using one vehicle of type k during a day.

At a particular day, n customers have ordered some goods to be delivered from the depot to their places: customer i is expecting to get goods of total weight d_i. To simplify the notations, we will also consider the depot as a customer with zero demand. For vehicle of type k, it costs c_{ij}^k to travel from customer i to customer j.

A *feasible route* for vehicle of type k is given by a list $(i_0 = 0, i_1, \dots, i_r, i_{r+1} = 0)$ of customers such that the total demand of the customers on this route does not exceeds the vehicle capacity, $\sum_{s=1}^{r} d_s \le u_k$. The cost of assigning a vehicle of type k on this route is $f_k + \sum_{s=1}^{r} c^k(i, j)$.

The *vehicle routing problem* (*VRP*) is to select a subset of vehicles (from the depot fleet) and then assign any chosen vehicle to a feasible route so that each customer is visited by just one vehicle and the total cost of assigning vehicles to the routes is minimal.

5.2.1 Mixed Integer Programming Formulation

To formulate the VRP as an IP, we need the following family of decision binary variables:

- for $k = 1, \dots, K$ and $i, j \in \{0, 1, \dots, n\}$, $x^k_{ij} = 1$ if some vehicle of type k travels directly from customer i to customer j, and $x^k_{ij} = 0$ otherwise.

In addition, we also need one family of auxiliary variables:

- for $i, j \in \{0, 1, \dots, n\}$, y_{ij} is amount of goods that are carried by the vehicle assigned on the route from customer i to customer j.

In these variables the model is written as follows:

$$\sum_{k=1}^{K}\sum_{j=1}^{n}(f_k + c^k_{0,j})x^k_{0,j} + \sum_{k=1}^{K}\sum_{i=1}^{n}\sum_{j\in\{0,\dots,n\}\setminus\{i\}} c^k_{ij}x^k_{ij} \to \min, \tag{5.2a}$$

$$\sum_{j=1}^{n} x^k_{0,j} \le q_k, \quad k = 1, \dots, K, \tag{5.2b}$$

$$\sum_{k=1}^{K}\sum_{i=0}^{n} x^k_{ij} = 1, \quad j = 1, \dots, n, \tag{5.2c}$$

$$\sum_{i=0}^{n} x^k_{ij} - \sum_{i=0}^{n} x^k_{ji} = 0, \quad j = 1, \dots, n,\ k = 1, \dots, K, \tag{5.2d}$$

$$\sum_{i=0}^{n} y_{ij} - \sum_{i=0}^{n} y_{ji} = d_j, \quad j = 1, \dots, n, \tag{5.2e}$$

$$0 \le y_{ij} \le \sum_{k=1}^{K}(u_k - d_i)x^k_{ij}, \quad i, j = 0, \dots, n, \tag{5.2f}$$

$$x^k_{ii} = 0, \quad i = 0, \dots, n,\ k = 1, \dots, K, \tag{5.2g}$$

$$x^k_{ij} \in \{0, 1\}, \quad i, j = 0, \dots, n,\ k = 1, \dots, K. \tag{5.2h}$$

The objective (5.2a) is to minimize the total fixed cost of using vehicles plus the traveling cost of all used vehicles along their routes. The inequalities (5.2b) require that the number of vehicles of any particular type that leave the depot does not exceed the number of such vehicles in the depot fleet. The equations (5.2b) ensure that each customer will be visited by just one vehicle, while the equations (5.2d) guarantee that any vehicle that arrives to a customer will leave that customer. The next family of equations, (5.2e), reflects the fact that any vehicle before leaving a customer must upload the goods ordered by that customer. Each inequality from (5.2f) imposes the capacity restriction: if a vehicle of type k travels the route from customer i to customer j, then the total cargo weight on its board cannot exceed $u_k - d_i$, but, if none of vehicles (of any type) travels along the route (i, j), then $y_{ij} = 0$. One can easily argued that these capacity restrictions guarantee that each used vehicle will never carry goods of total weight greater than the vehicle capacity.

A note of precaution is appropriate here. Because of the capacity constraints (5.2f), the entire formulation (5.2) may be *weak*. In our case, the reason for its weakness is in the following. Let us assume that some inequality $y_{ij} \le \sum_{k=1}^{k}(u_k - d_i)x_{ij}^k$ holds as equality. If the capacity, u_k, is big, and the demand, d_i, is small, then $u_k - d_i$ is big. If we further assume that the sum of demands of all customers that are on the same route with customer i and that are visited after customer i is small, then y_{ij} will also be small and, therefore, at least one binary variable x_{ij}^k will take a small fractional value. Many binary variables taking fractional values is usually an indicator of a hard to solve problem. Therefore, one can hardly expect that formulation (5.2) can be used for solving to optimality even VRPs of moderate size. Nevertheless, if implemented properly, an application based on this formulation can produce rather good approximate solutions for VRPs of practical importance.

5.2.2 MIPCL-PY Implementation

A Python module designed for solving VRPs is presented in listings 5.5 and 5.6.

Listing 5.5 presents the definition of class `VRP` and an implements of IP (5.2), which is given, as usually, in the function named `model`.

```
from mipcl_py.mipshell.mipshell import *
from math import sqrt

class VRP(Problem):
    def __init__(self, name, instance):
        Problem.__init__(self,name)
        self.instance = instance

    def model(self):
        def d(j): return customer[j]['Demand']
        def q(k): return vehicleType[k]['Quantity']
        def u(k): return vehicleType[k]['Capacity']
        def c(k,i,j):
            d = customer[i]['Coord'][0]-customer[j]['Coord'][0]
            cost = d*d
            d = customer[i]['Coord'][1]-customer[j]['Coord'][1]
            cost += d*d
            cost = round(sqrt(cost))
            cost *= vehicleType[k]['UnitDistCost']
            if i == 0: cost += vehicleType[k]['FixedCost']
            return cost

        vehicleType = self.instance['VehicleTypes']
        customer = self.instance['Customers']
        K, n = len(vehicleType), len(customer)
        self.x = x = VarVector((K,n,n),"x",BIN)
        y = VarVector((n,n),"y",INT)

        minimize(sum_(c(k,i,j)*x[k][i][j] for k in range(K) \
                                          for i in range(n) for j in range(n)))
        for k in range(K):
            sum_(x[k][0][j] for j in range(1,n)) <= q(k)
            for j in range(1,n):
                sum_(x[k][i][j] for i in range(n)) == 1
                sum_(x[k][i][j] for i in range(n)) - \
                      sum_(x[k][j][i] for i in range(n)) == 0
            for i in range(n):
                x[k][i][i] == 0

            for j in range(1,n):
                sum_(y[i][j] for i in range(n)) - \
                    sum_(y[j][i] for i in range(n)) == d(j)
                for i in range(n):
                    y[i][j] <= sum_((u(k)-d(i))*x[k][i][j] for k in range(K))
```

Listing 5.5: MIPCL-PY implementation of the VRP: constructor and `model`

The constructor, `__init__()`, stores a reference to a structure, named `instance`, that describes a VRP-instance. This structure has the following fields:

- `instance['Name']`: instance name;
- `instance['VehicleTypes']`: list of dictionaries that describes the depot fleet of vehicles, where `vehicleType = instance['VehicleTypes'][t]` describes the vehicles of type `t`:
  - `vehicleType['Name']`: vehicle identifier (or model name);
  - `vehicleType['Quantity']`: number of vehicles of this type in the fleet;
  - `vehicleType['Capacity']`: vehicle capacity;
  - `vehicleType['FixedCost']`: fixed cost of using one vehicle;
  - `vehicleType['UnitDistCost']`: cost of traveling unit distance;
- `instance['Customers']`: list of customers, where `customer = instance['Customers'][i]` describes customer `i`:
  - `cuscomer['Name']`: customer name;
  - `cuscomer['Coord']`: coordinates of the customer place;
  - `cuscomer['Demand']`: demand for goods.

The function `model`, first, defines a number of local functions that maps some input parameters onto the coefficients of IP (5.2), then a straightforward implementation of this IP follows. Let us notice that, when implementing the function, `c(k,i,j)`, that computes transportation expenses, the fixed costs of hiring vehicles are summed together with the transportation costs of traveling along the legs leaving the depot.

Listing 5.6 presents two auxiliary procedures.

```
    def setSolution(self):
        x = self.x
        K, n = len(x), len(x[0])
        self.first = first = []
        self.next = next = [0 for i in range(n)]

        for k in range(K):
            for j in range(1,n):
                if x[k][0][j].val > 0.5:
                    first.append((k,j))
                for i in range(1,n):
                    if x[k][i][j].val > 0.5:
                        next[i] = j
                        break

    def printSolution(self):
        vehicleType = self.instance['VehicleTypes']
        customer = self.instance['Customers']
        first, next = self.first, self.next

        print('Objective value: {0:.2f}\n'.format(self.getObjVal()))

        for r,(k,i) in enumerate(first):
            str = '\t0'
            load = 0
            while True:
                str += ' -> {:d}'.format(i)
                if i > 0:
                    load += customer[i]['Demand']
                    i = next[i]
                else: break
            print('Route {:d}, vehicle of type {!s}, cargo weight - {:d}'.\
                          format(r+1,vehicleType[first[r][0]]['Name'],load))
            print(str)
```

Listing 5.6: MIPCL-PY implementation of VRP: auxiliary procedures

The function `setSolution` extracts the vehicles routes from the values of `x`-variables. The routes are stored in two lists:

- `first`: list of pairs of integers, where `first[r][0]` is the type of a vehicle that serves route `r`, and `first[r][1]` is the first customer on route `r`;
- `next`: list of integers, where `next[i]` is the customer that is on the same route with customer `i` and is visited just after customer `i`.

The other auxiliary procedure, `printSolution`, prints the routes stored in the lists `first` and `next`. In addition, this procedure counts the weights of cargoes on all the routes.

5.2.3 Example of Usage

The program presented below will show you how to use our class `VRP`.

```
#!/usr/bin/python

from mipcl_py.models.VRP import VRP

vrp1 = {
  'NAME': "vrp1",
  'VehicleTypes': (
     {'Name': "1", 'Quantity': 3, 'Capacity': 100, 'FixedCost': 0, 'UnitDistCost':
         1},
  ),
  'Customers': (
     {'Name': "Depot", 'Coord': (82,76), 'Demand': 0},
     {'Name':  "1", 'Coord': (96,44), 'Demand': 19},
     {'Name':  "2", 'Coord': (50, 5), 'Demand': 21},
     {'Name':  "3", 'Coord': (49, 8), 'Demand':  6},
     {'Name':  "4", 'Coord': (13, 7), 'Demand': 19},
     {'Name':  "5", 'Coord': (29,89), 'Demand':  7},
     {'Name':  "6", 'Coord': (58,30), 'Demand': 12},
     {'Name':  "7", 'Coord': (84,39), 'Demand': 16},
     {'Name':  "8", 'Coord': (14,24), 'Demand':  6},
     {'Name':  "9", 'Coord': ( 2,39), 'Demand': 16},
     {'Name': "10", 'Coord': ( 3,82), 'Demand':  8},
     {'Name': "11", 'Coord': ( 5,10), 'Demand': 14},
     {'Name': "12", 'Coord': (98,52), 'Demand': 21},
     {'Name': "13", 'Coord': (84,25), 'Demand': 16},
     {'Name': "14", 'Coord': (61,59), 'Demand':  3},
     {'Name': "15", 'Coord': ( 1,65), 'Demand': 22},
     {'Name': "16", 'Coord': (88,51), 'Demand': 18},
     {'Name': "17", 'Coord': (91, 2), 'Demand': 19},
     {'Name': "18", 'Coord': (19,32), 'Demand':  1},
     {'Name': "19", 'Coord': (93, 3), 'Demand': 24},
     {'Name': "20", 'Coord': (50,93), 'Demand':  8},
     {'Name': "21", 'Coord': (98,14), 'Demand': 12}
  )
}

prob = VRP("VRP1",vrp1)
prob.model()
prob.optimize(False,3600)
prob.setSolution()
prob.printSolution()
```

Listing 5.7: Example of usage of class `VRP`

If we run the above program, we get the following result.

```
Objective value: 635.00

Route 1, vehicle of type 1, cargo weight - 100
    0 -> 8 -> 4 -> 11 -> 9 -> 15 -> 10 -> 5 -> 20 -> 0
Route 2, vehicle of type 1, cargo weight - 90
    0 -> 16 -> 7 -> 13 -> 1 -> 12 -> 0
Route 3, vehicle of type 1, cargo weight - 98
    0 -> 21 -> 19 -> 17 -> 2 -> 3 -> 18 -> 6 -> 14 -> 0
```

Listing 5.8: Output of program from Listing 5.7

5.3 Multi-Dimensional Packing

In a m-*dimensional* (*orthogonal*) *packing problem* the main requirement is to pack a number of small m-dimensional rectangular boxes (*items*) into a large m-dimensional rectangular box (*container*) so that no two items overlap, and item edges are parallel to the container edges.

Two-dimensional ($m = 2$) packing problems arise in different industries, where steel, wood, glass, or textile materials are cut. In such cases these packing problems are also known as the *two-dimensional cutting stock problems*. The problem to optimize the layout of advertisements in a newspaper is also formulated as a two-dimensional packing problem. Three-dimensional ($m = 3$) packing problems — also known as the *container loading problems* — appear as important subproblems in logistics and supply chain applications.

Many scheduling problems can be formulated as multi-dimensional packing problems: we represent a job as a box one of which sizes is its processing time, and the other ones represent amounts of some renewable resources needed to fulfill this job. Then the container sizes give the limit on the processing time of all jobs, and the available amounts of all resources.

We are given a large m-dimensional rectangular box (*container*) which sizes are given by a vector $L = (L_1, \ldots, L_m)^T \in \mathbb{Z}^m$ and a set of n small m-dimensional rectangular boxes (*items*), item r sizes are given by a vector $l^r = (l_1^r, \ldots, l_m^r)^T \in \mathbb{Z}^m$.

In the (*orthogonal*) *m-dimensional knapsack problem* (m-KP), for each item r, we also know its *cost* c_r. The goal is to pack into the container — which is also called the *knapsack* — a subset of items of maximum total cost so that no two items overlap, and item edges are parallel to knapsack edges. If the items cannot be rotated (say, when cutting decorated materials, or scheduling machines), then every edge i of each item is parallel to knapsack edge i — we have an m-KP without rotation.

In the *m-dimensional strip packing problem* (m-SP), it is assumed that one edge (let us call it the *height*) of the container — which is called the *strip* — is sufficiently big so that all the items can be packed into the strip, and the objective is to minimize the height to which the strip is used.

In the *m-dimensional bin packing problem* (m-BP) there are many large boxes of equal sizes — which are called *bins* — and the objective is to pack all n items into a minimum number of bins. We can translate any m-BP instance into an instance of $(m+1)$-SP, where the additional $(m+1)$-st direction (the height of the strip) is used to count bins: L_{m+1} is an upper bound on the number of needed bins, and $l_{m+1}^r = 1$ for all items r.

In what follows we will develop an integer programming (IP) formulation of m-KP. With minor modifications that formulation is also valid for m-SP (we just have to change the objective).

5.3.1 Integer Programming Formulation of m-KP

The first integer programming (IP) formulation of a two-dimensional packing problem (namely, the cutting stock problem) based on the discrete representation of the geometrical space was given by Beasley[1]. Similar formulation was given by Hadjiconstantinou and Christofides[2]. Both formulations have a great deal of variables, and their constraint matrices are dense (each constraint contains a great deal of non zero entries) what results in IPs that are difficult to solve. Although both formulations can be easily extended to model packing problems in higher (than 2) dimensions, the number of variables in each extended formulation would increase exponentially as the dimension increases. Here we present a slightly different formulation of m-KP, that is substantially more compact — it has fewer variables and its matrix is sparse — compared to the above mentioned formulations. Moreover, the number of variables in our formulation grows only linearly as the dimension grows.

First, let us define two families of decision binary variables:

[1] J.E. Beasley. *An exact two-dimensional non-guillotine cutting tree search procedure*, Operational Research **33** (1985) 49–64.

[2] E. Hadjiconstantinou, N. Christofides. *An exact algorithm for the orthogonal, 2-D cutting problems using guillotine cuts*, European Journal of Operational Research **83** (1995) 21–38.

- for $r = 1, \dots, n$, $z_r = 1$ if item r is placed into the knapsack, and $z_r = 0$ otherwise;
- for $r = 1, \dots, n$, $i = 1, \dots, m$, and $j = 0, \dots, L_i - l_i^r$, $y_{rij} = 1$ if j is the value of coordinate i of the corner of item r that is nearest to the origin, and $y_{rij} = 0$ otherwise.

For modeling purposes, we also need two families of auxiliary variables:

- for $r = 1, \dots, n$, $i = 1, \dots, m$, and $j = 1, \dots, L_i$, $x_{rij} = 1$ if the open unit strip $U_j^i \overset{\text{def}}{=} \{w \in \mathbb{R}^m : j - 1 < w_i < j\}$ intersects item r, and $x_{rij} = 0$ otherwise;
- for $r_1 = 1, \dots, n-1$, $r_2 = r_1 + 1, \dots, n$, and $i = 1, \dots, m$, $s_{r_1,r_2,i} = 1$ if items r_1 and r_2 are separated by a hyperplane that is ortogonal to axis i, and $s_{r_1,r_2,i} = 0$ otherwise.

In these variables the model is written as follows:

$$\sum_{r=1}^{n} c_r z_r \to \max, \tag{5.3a}$$

$$\sum_{j=0}^{L_i - l_i^r} y_{rij} = z_r, \quad i = 1, \dots, m,\ r = 1, \dots, n, \tag{5.3b}$$

$$x_{rij} = \sum_{j_1 = \max\{0, j - l_i^r\}}^{\min\{j-1, L_i - l_i^r\}} y_{r,i,j_1}, \quad j = 1, \dots, L_i,\ i = 1, \dots, m,\ r = 1, \dots, n, \tag{5.3c}$$

$$\sum_{j=1}^{L_i} x_{rij} = l_i^r z_r, \quad i = 1, \dots, m,\ r = 1, \dots, n, \tag{5.3d}$$

$$\sum_{i=1}^{m} s_{r_1,r_2,i} \ge z_{r_1} + z_{r_2} - 1, \quad r_2 = r_1 + 1, \dots, n,\ r_1 = 1, \dots, n-1, \tag{5.3e}$$

$$x_{r_1,i,j} + x_{r_2,i,j} + s_{r_1,r_2,i} \le z_{r_1} + z_{r_2}, \quad j = 1, \dots, L_i,\ i = 1, \dots, m,\ r_2 = r_1 + 1, \dots, n,\ r_1 = 1, \dots, n-1, \tag{5.3f}$$

$$z_r \in \{0, 1\}, \quad r = 1, \dots, n, \tag{5.3g}$$

$$y_{rij} \in \{0, 1\}, \quad j = 0, \dots, L_i - l_i^r,\ i = 1, \dots, m,\ r = 1, \dots, n, \tag{5.3h}$$

$$x_{rij} \in \{0, 1\}, \quad j = 1, \dots, L_i,\ i = 1, \dots, m,\ r = 1, \dots, n, \tag{5.3i}$$

$$s_{r_1,r_2,i} \in \{0, 1\}, \quad i = 1, \dots, m,\ r_2 = r_1 + 1, \dots, n,\ r_1 = 1, \dots, n-1. \tag{5.3j}$$

The objective (5.3a) is to maximize the total cost of items placed into the knapsack. The inequalities (5.3b) set to zero the values of those y-variables that correspond to the items not placed into the knapsack. The equations (5.3c) establish the relationship between x and y-variables. The equalities (5.3d) require that each item in the knapsack must be of given sizes. Two families of inequalities, (5.3e) and (5.3f) imply that each pair of items placed into the knapsack are separated by at least one hyperplane that are ortogonal to a coordinate axis.

The other relations (5.3g)–(5.3j) declare that the all variables are binary.

5.3.2 Tightening Basic Model

Computational experiments showed that our *basic model* (5.3a)–(5.3j) is not tight, and, therefore, there is no hope to apply it for solving problems of practical importance. Next we will add to (5.3a)–(5.3j) new constrains to make the formulation tighter.

First, we introduce a family of knapsack inequalities that can significantly strengthen our IP (5.3a)–(5.3j). Let us denote by Vol the volume, $\prod_{i=1}^m L_i$, of the knapsack, and by vol_r the volume, $\prod_{i=1}^m l_i^r$, of item r. With this notations, we introduce the following knapsack inequalities:

$$\sum_{r=1}^n vol_r z_r \le Vol, \tag{5.4a}$$

$$\sum_{r=1}^n \frac{vol_r}{l_i^r} x_{rij} \le \frac{Vol}{L_i}, \quad j = 1, \dots, L_i, \ i = 1, \dots, m. \tag{5.4b}$$

The inequality (5.4a) imposes the following natural restriction: the sum of item volumes cannot exceed the knapsack volume. The inequalities (5.4b) express the fact that the sum of volumes of the intersections of the items with any unit strip ortogonal to some axis cannon exceed the volume of the intersection of the knapsack with that strip.

Computational experiments showed that these knapsack inequalities may greatly tighten the IP formulation.

The purpose of the next two families of constraints is to remove symmetry. For two items r_1, r_2, we write $r_1 \preceq r_2$ if $l_i^{r_1} \le l_i^{r_2}$ for all i, $c_{r_1} \ge c_{r_2}$, and either at least one of these inequalities are strict or $r_1 < r_2$. Two items r_1, r_2 are identical if they have the same sizes ($l^{r_1} = l^{r_2}$) and costs ($c_{r_1} = c_{r_2}$), in such a case we write $r_1 \equiv r_2$. With these notations we can add to (5.3a)–(5.3j) the following inequalities:

$$z_{r_1} \ge z_{r_2}, \quad r_1, r_2 = 1, \dots, n, \ r_1 \preceq r_2, \tag{5.5a}$$

$$o_{ri} = \sum_{j=0}^{L_i - l_i^r} j y_{rij}, \quad i = 1, \dots, m, \ r = 1, \dots, n, \tag{5.5b}$$

$$\sum_{i=1}^m o_{r_1,i} \le \sum_{i=1}^m o_{r_2,i}, \quad \forall\, r_1 \equiv r_2, \ r_1 < r_2, \tag{5.5c}$$

$$o_{ri} \ge 0, \quad i = 1, \dots, m, \ r = 1, \dots, n. \tag{5.5d}$$

The inequalities (5.5a) imply that out of two related items r_1, r_2, $r_1 \preceq r_2$, item r_2 will be in the knapsack only if item r_1 is already there. The equations (5.5b) are to compute, for each item r placed into the knapsack, the item corner nearest to the origin, $(o_{r1}, \dots, o_{rm})$. The inequalities (5.5c) are to prevent exchanging the positions of two identical items when branching by a y_{rij} variable. Each of these inequalities requires that of two identical items the item which index is smaller must be closer to the origin.

When we rotate the knapsack with some items inside along the line passing through the knapsack center and parallel to some coordinate axis, we get a new symmetric placement of the same items. These symmetries may significantly increase the solution time of any modern IP solver. To avoid two central symmetries, we can add another system of inequalities. For $i = 1,\dots,m$, let us define two integers: $k_1^i = L_i \mod 2$, $k_2^i = k_1^i$ if k_1^i is even, or $k_2^i = k_1^i + 1$ if k_1^i is odd. The following inequalities remove all central symmetries:

$$\sum_{r=1}^{n}\sum_{j=1}^{k_1^i}(L_i - j)x_{rij} \ge \sum_{r=1}^{n}\sum_{j=k_2^i}^{L_i}(j+1)x_{rij}, \quad i = 1,\dots,m. \tag{5.6}$$

Rotations and Complex Loading Patterns

In this section we show how to extend our IP formulation of m-KP to cover the cases when rotations of items are allowed, as well as when the knapsack and the items are the unions of rectangular boxes.

If rotations of items are allowed, let us assume that all possible rotations of item r result in the boxes of the following sizes: $l^{r1},\dots,l^{r,Q_r} \in \mathbb{Z}^m$. Then we must substitute (5.3d) for the following system:

$$\begin{aligned}
&\sum_{j=1}^{L_i} x_{rij} = \sum_{q=1}^{Q_r} l_i^{rq} f_{rq}, \quad i = 1,\dots,m,\ r = 1,\dots,n,\\
&\sum_{q=1}^{Q_r} f_{rq} = z_r, \quad r = 1,\dots,n,\\
&f_{r,q} \in \{0,1\}, \quad q = 1,\dots,Q_r,\ r = 1,\dots,n.
\end{aligned} \tag{5.7}$$

Let the knapsack be a rectangular box from which a number of non-overlapping rectangular boxes have been removed. Alternatively, we can think that a number of dummy items should be placed into the knapsack at predefined positions. It is convenient to think that each item r is either dummy or real. Let $\mathcal{D} \subseteq \{1,\dots,n\}$ denote the set of dummy items and let $j^r \in \mathbb{R}^m$ be the closest to the origin corner of dummy item r. To guaranty that all dummy items are put into the knapsack at their predefined positions, we just fix the values of some x-variables:

$$y_{rij^r} = 1, \quad i = 1,\dots,m,\ r \in \mathcal{D}. \tag{5.8}$$

To model the case when some items are unions of two or more rectangular boxes, let us assume that our input n items are divided into groups of one or more items, and all items from a group are together put or together not put into the knapsack. In each group of items we choose one item, $\bar{r}$, called the *base*; any non-base item r in this group is assigned a reference, $ref(r) = \bar{r}$, to the base item, $ref(\bar{r}) = -1$. Each non-base item r is also assigned a vector $v^r \in \mathbb{R}^m$: if $o_{\bar{r}}$ is the nearest to the origin corner of the base item $\bar{r} = ref(r)$, then $o_{\bar{r}} + v^r$ is the nearest to the origin corner of item r. In other words, the vectors v^r assigned to all non-base items r in a group uniquely determine the shape of this group.

The following equations model the above defined group restrictions;

$$\begin{aligned}
z_r &= z_{ref(r)}, \quad r = 1,\dots,n,\ ref(r) \ge 0,\\
o_{ri} &= 0_{ref(r),i} + v_i^r z_r, \quad i = 1,\dots,m,\ r = 1,\dots,n,\ ref(r) \ge 0.
\end{aligned} \tag{5.9}$$

5.3.3 MIPCL-PY Implementation

Our implementation of IP (5.3) and some of its extensions (5.5), (5.6), and (5.4), is given in listings 5.9, 5.10, and 5.11.

Listing 5.9 presents the definition of class `Packing` that is a subclass of another class, `Problem`, which implements an interface to the `mipcl-py` module.

```
from mipcl_py.mipshell.mipshell import *

class Packing(Problem):
    def prepare(self, factor, up):
        up = self.up
        LR, Vol = [], 1.0
        for s in self.instance['Cont. Sizes']:
            sr = (s * factor[0]) // factor[1]
            Vol = Vol * sr
            LR.append(sr)
        self.instance['Volume'] = Vol
        self.instance['Cont. rSizes'] = LR

        for item in self.instance['Items']:
            lr, v = [], 1.0
            for s in item['Sizes']:
                sr = (s * factor[0]) // factor[1]
                if up:
                    if (sr * factor[1]) < s * factor[0]: sr += 1
                v = v * sr
                lr.append(sr)
            item['Volume'] = v
            item['rSizes'] = lr

    def __init__(self, name, instance, factor=(1,1), up=True):
        Problem.__init__(self,name)
        self.instance = instance
        self.factor = factor
        self.up = up
        self.prepare(factor,up)
```

Listing 5.9: MIPCL-PY implementation of m-KP: `prepare` and the constructor

The constructor, `__init__()`, stores a reference to a structure, named `instance`, that describes an instance of m-KP. This structure has the following fields:

- `instance['Cont. Sizes']`: tuple of container sizes;
- `instance['Items']`: list of dictionaries, where `item = instance['Items'][i]` describes item `i`;
  - `item['Type']`: type of item; usually, identical items are of the same type;
  - `item['Sizes']`: tuple of item sizes;
  - `item['Cost']`: item cost.

The last two arguments passed to `__init__()`, `factor` and `up`, are useful when packing items into a big container. If the value of `up` is `True`, `prepare()`

1) divides the container sizes by the value of `factor` and then rounds down the results;

2) divides the sizes of all items by the value of `factor` and then rounds up the results.

As a result, we will get the packing problem which is easier to solve and which solution is feasible for the original problem.

If the value of `up` is `False`, `prepare()`

1) divides the container sizes by the value of `factor` and then rounds up the results;

2) divides the sizes of all items by the value of `factor` and then rounds down the results.

Again we will get the packing problem which is easier to solve, and the optimal objective value for this problem is an upper bound for the optimal objective value of the original problem. We can use this upper bound to estimate the quality of any approximate solution.

Our implementation of IP (5.3) is given in the function named `model` which is shown in Listing 5.10.

```
    def model(self):
        def c(r): return items[r]['Cost']
        def vol(r): return items[r]['Volume']
        def l(r,i): return items[r]['rSizes'][i]

        instance = self.instance
        L = instance['Cont. rSizes']
        items = instance['Items']
        Vol = instance['Volume']
        m, n = len(L), len(items)

        self.z = z = VarVector([n],'z',BIN,priority=10)
        self.x = x = [[VarVector([L[i]],'x[{:d}][{:d}]'.format(r,i),BIN) \
                       for i in range(m)] for r in range(n)]
        y = [[VarVector([L[i]-l(r,i)+1],'y[{:d}][{:d}]'.format(r,i),BIN) \
                for i in range(m)] for r in range(n)]
        s = [{r1: [Var('s({:d},{:d},{:d})'.format(r,r1,i),BIN) for i in range(m)] \
                for r1 in range(r+1,n)} for r in range(n)]
        o = VarVector((n,m),'o')

        maximize(sum_(c(r)*z[r] for r in range(n)))

        for r in range(n):
            for i in range(m):
                o[r][i] == sum_(j*y[r][i][j] for j in range(L[i]-l(r,i)+1))
                for j in range(L[i]):
                    x[r][i][j] == sum_(y[r][i][j1] for j1 in \
                                       range(max(0,j-l(r,i)+1),min(j,L[i]-l(r,i))+1))
                sum_(y[r][i][j] for j in range(L[i]-l(r,i)+1)) == z[r]
# restrictions on the item sizes
                sum_(x[r][i][j] for j in range(L[i])) == l(r,i)*z[r]

# each pair of items, (r,r1), must be separated by a hyperplane
# which is orthogonal to one of m axes
            for r1 in range(r+1,n):
                sum_(s[r][r1][i] for i in range(m)) >= z[r] + z[r1] - 1
            q = 0
            for i in range(m):
                if l(r,i) + l(r1,i) <= L[i]:
                    for j in range(L[i]):
                        x[r][i][j] + x[r1][i][j] + s[r][r1][i] <= z[r] + z[r1]
                else:
                    s[r][r1][i] == 0
                    q += 1
            if q == m:
                z[r] + z[r1] <= 1

# Less size and bigger cost => allocate first
            q, q1 = 0, 0
            if c(r) >= c(r1): q += 1
            if c(r) <= c(r1): q1 += 1
            for i in range(m):
                if l(r,i) <= l(r1,i): q += 1
                if l(r,i) >= l(r1,i): q1 += 1
                if q == m+1:
                    z[r] >= z[r1]
                    if q1 == m+1:
# if r and r1 are identical, allocate r closer to the origin
                        sum_(o[r][i] for i in range(m)) <= \
                        sum_(o[r1][i] for i in range(m))
                elif q1 == m+1:
                    z[r] <= z[r1]
# volume knapsack
        sum_(vol(r)*z[r] for r in range(n)) <= Vol
# strip knapsacks
        for i in range(m):
            for j in range(0,L[i],10):
                sum_((vol(r) // l(r,i))*x[r][i][j] for r in range(n) if l(r,i) > 0)
                    <= Vol // L[i]

# breaking central symmentries
        for i in range(m):
            L0 = L[i]
            k1 = L0 // 2
            if 2*k1 < L0: k2 = k1 + 1
            else: k2 = k1
            sum_((L0-j)*x[r][i][j] for r in range(n) for j in range(k1)) >= \
            sum_((j+1)*x[r][i][j] for r in range(n) for j in range(k2,L0))
```

Listing 5.10: MIPCL-PY implementation of m-KP: `model`

When a solution has been found, one may want to print it in a readable format. Listing 3.6 presents a procedure — which is a member of `Packig` — that does this job.

```
    def getLayout(self):
        L = self.instance['Cont.␣rSizes']
        factor = self.factor
        m, n = len(L), len(self.instance['Items'])
        x, z = self.x, self.z
        layout = {'Cost': int(self.getObjVal() + 0.5), 'Positions': {}}
        positions = layout['Positions']

        for r in range(n):
            if z[r].val > 0.5:
                origin = positions[r] = []
                for i in range(m):
                    for j in range(L[i]):
                        if x[r][i][j].val > 0.5:
                            origin.append((j * factor[1]) // factor[0])
                            break
        return layout

    def printLayout(self):
        if self.is_solution is not None:
            if self.is_solution:
                items = self.instance['Items']
                m = len(self.instance['Cont.␣Sizes'])
                factor = self.factor
                layout = self.getLayout()
                positions = layout['Positions']
                print('Layout␣cost␣=␣{:d}'.format(layout['Cost']))
                for r in sorted(positions.keys()):
                    str0 = repr(items[r]['Type']).rjust(3) + '({:d})␣:␣'.format(
                        items[r]['Cost']).rjust(8)
                    str1, str2 = '', ''
                    for i,X in enumerate(positions[r]):
                        if i > 0:
                            str1 += ','
                            str2 += ','
                        str1 += repr(X)
                        str2 += repr(X + items[r]['Sizes'][i])
                    print(str0 + '[(' + str1 + '),␣(' + str2 + ')]')
            else:
                print('No␣feasible␣solution␣has␣been␣found')
        else:
            print('Please␣call␣optimize()␣first')
```

Listing 5.11: MIPCL-PY implementation of m-KP: auxiliary procedures

First, the procedure `printLayout` calls `getLayout` to extract the layouts of items from the values of decision variables `z[r]` and `x[r][i][j]`. Then this layout is printed.

5.3.4 Example of Usage

The program presented below will show you how to use our class `Packing`.

```
#!/usr/bin/python
from multiPacking import Packing

instance = {
  'Cont.␣Sizes': (10, 10),
  'Items': (
     {'Type': 1, 'Sizes': (3, 7), 'Cost': 35},
     {'Type': 1, 'Sizes': (3, 7), 'Cost': 35},
     {'Type': 2, 'Sizes': (8, 2), 'Cost': 40},
     {'Type': 2, 'Sizes': (8, 2), 'Cost': 40},
     {'Type': 3, 'Sizes': (10, 2), 'Cost': 27},
     {'Type': 4, 'Sizes': (5, 4), 'Cost': 23},
     {'Type': 4, 'Sizes': (5, 4), 'Cost': 23},
     {'Type': 4, 'Sizes': (5, 4), 'Cost': 23},
     {'Type': 5, 'Sizes': (2, 9), 'Cost': 43},
     {'Type': 5, 'Sizes': (2, 9), 'Cost': 43}
  )
}

prob = Packing("test0",instance)
```

```
prob.model()
prob.optimize(False,3600)
prob.printLayout()
```

Listing 5.12: Example of usage of class `Packing`

If we run the above program, we get the following result.

```
Layout cost = 164
  1 (35) : [(5,1), (8,8)]
  2 (40) : [(0,8), (8,10)]
  4 (23) : [(0,4), (5,8)]
  4 (23) : [(0,0), (5,4)]
  5 (43) : [(8,0), (10,9)]
```

Listing 5.13: Output of program from Listing 5.12

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