NTNU | Norwegian University of Science and Technology

FLOW AND NETWORK MODELLING

Jørn Vatn/September-2024

Motivation

- Our LP framework can be used in many “classical” situations described in the literature
- Sometimes it is more efficient with other approaches.....

Transportation problems

- We often deal with the problem of distributing products from $m > 1$ sources to $n > 1$ locations
- In the retail industry the sources could be warehouses and the locations could be markets or specific stores
- In the problem formulation we will use the term warehouse, $W_i, i = 1, \ldots, m$ for the sources and marked, $M_j, j = 1, \ldots, n$ for the locations receiving the products
- In the model we assume only one product type and deterministic supply and demand

Transportation problems, cont.

- The supply provided by warehouse W_i is a_i
- The demand by marked M_j is b_j
- c_{ij} is the unit cost of shipping products from W_i to M_j
- The decision variables are x_{ij}, i.e., the shipments from W_i to M_j
- Problem formulation

$$\text{Minimize: } Z = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$\text{Subject to: } \sum_{j=1}^{n} x_{ij} \leq a_i, i = 1, \ldots, m$$

$$\sum_{i=1}^{m} x_{ij} \geq b_j, j = 1, \ldots, n$$

$$x_{ij} \geq 0$$

Example

- Consider a situation with $m = 3$ warehouses and $n = 4$ markets with shipping costs given in the table below
- Further assume supply parameters given by $a_1 = 3$, $a_2 = 8$ and $a_3 = 6$ and demand parameters are given by $b_1 = 4$, $b_2 = 3$, $b_3 = 2$ and $b_4 = 5$

	M_1	M_2	M_3	M_4
W_1	3	3	3	2
W_2	12	8	4	3
W_3	9	7	7	10

▸ Solution in Python

Example, Python considerations

In Python we specify the problem and import libraries:

```
import numpy as np
import lpLib as lp
a = [3,8,6]
b = [5,3,2,7]
C = \
[[3,  3,  3,  2], \
[12,  8,  4,  3], \
[9, 7,  7,  10]]
```

Example, Python considerations, cont

- The `lpLib` library assumes that we don't have double indexing in our x-variable
- That is, we would like to have an index function returning one index representing $x_{i,j}$
- The following code provides such an index function:

```
10 def ndx(i,j): # One-dimensional index to represent warehouse i and
       marked j
11     return i*n + j
```

Example, Python considerations, cont

To specify the objective function with `coefficients` from `lpLib` we use:

```
13 m = len(a)
14 n = len(b)
15 c = np.zeros([m*n])
16 for i in range(m):
17     for j in range(n):
18         c[ndx(i,j)] = C[i][j] # Copy cost coefficients from matrix to
       vector
19 lp.coefficients(c)
```

Constraints specification in Python

```
21 for i in range(m): # Supply constraints, i.e., sum out of ware house i <=
       a_i
22     A_row = np.zeros([m*n])
23     for j in range(n):
24         A_row[ndx(i,j)] = 1
25     lp.upperBound(a[i],A_row)
26
27 for j in range(n): # Demand constraints, i.e., sum in to marked j >= b_j
28     A_row = np.zeros([m*n])
29     for i in range(m):
30         A_row[ndx(i,j)] = 1
31     lp.lowerBound(b[j],A_row)
```

And we use `lp.lpSolve()` to obtain the solution

Job assignment problems

In a manufacturing shop machining is required for several jobs

- The jobs are denoted $J_j, j = 1, \ldots, n$
- There are $M_i, i = 1, \ldots, n$ machines available for the machining
- We assume that each job only needs one machine, the machines can work in parallel, but one machine can only perform one job
- Let c_{ij} denote the cost of doing job J_j on machine M_i
- Binary decision variables, i.e., x_{ij} equals one if job J_j is done on machine M_i, zero otherwise

Objective function

$$\text{Minimize: } Z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$\text{Subject to: } \sum_{j=1}^{n} x_{ij} = 1, i = 1, \ldots, n$$

$$\sum_{i=1}^{n} x_{ij} = 1, j = 1, \ldots, n$$

x_{ij} are all binary

Example

The model is similar to the transportation problem

```
import numpy as np
import lpLib as lp
c = \
[[8,  9,  8,  10],\
[3, 2,  5,  6],\
[2, 1,  4,  2],\
[4, 3,  5,  6]]
n = len(c)
```

▸ Solution in Python

Constraints

We use the same index function as in the transportation problem, and the objective function is also specified as in the transportation problem. The constraints are specified by:

```
for i in range(n): # For each machine i, only one job can be done
    A_row = np.zeros([n*n])
    for j in range(n):
        A_row[ndx(i,j)] = 1
    lp.equalityConstraint(1,A_row)

for j in range(n): # For each job j, only one machine can do the job
    A_row = np.zeros([n*n])
    for i in range(n):
        A_row[ndx(i,j)] = 1
    lp.equalityConstraint(1,A_row)
```

Binary variables

Since all variables are binary we need to test all possible combinations, and we use the `all_binary_combinations` from `lpLib`:

```
23 best_Z = 1e30
24 for x in lp.all_binary_combinations(n*n):
25     if lp.isFeasible(x):
26         Z = lp.Z(x)
27         if Z < best_Z:
28             best_Z = Z
29             best_x = x
```

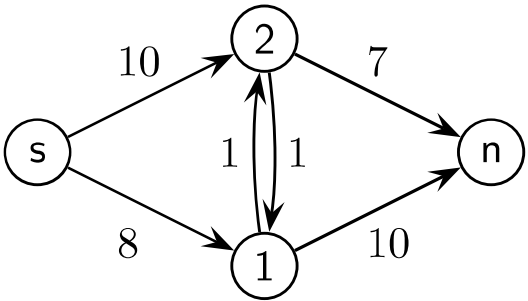
Note that the `isFeasible(x)` function from `lpLib` tests if a specific *x* vector is fulfilling all constraints specified, and the `Z(x)` function from `lpLib` calculates the objective function for a given *x* vector.

Maximal-flow problems and flow networks

- A flow network is a directed graph with arcs (edges) and nodes (vertices)
- An arc defines the maximum flow that can be transferred from one node to another node
- The capacities on flow between node *i* and node *j* is denoted c_{ij}
- The source node ***s*** is the location from where the flow starts, and the sink node *n* is the location receiving the flow
- The objective of the analysis is to determine the maximum flow *f* that can be sent from the source node to the sink node

The *conservation rule* has to be fulfilled at all nodes, this means that total flow out of a node must equal total flow in to that node

Example



Solution in Python

The LP problem is now formulated as follows

$$\text{Maximize: } Z = \sum_{j=1}^{n} x_{0,j}$$

$$\text{Subject to: } \sum_{j=1}^{n} x_{lj} = \sum_{i=0}^{n-1} x_{il}, l = 1, \ldots, n-1$$

$$\sum_{j=1}^{n} x_{0,j} = \sum_{i=0}^{n-1} x_{in}$$

$$c_{ij} \geq x_{ij}, i = 0, \ldots, n-1, j = 1, \ldots, n$$

where index $i = 0$ is used for the source node, and $c_{ii} = 0$ for all nodes per definition

Specifying the model in Python

We specify the flow capacities in a C-matrix, and load the required libraries:

```
1 import lpLib as lp
2 import numpy as np
3 #                        | 1  2 n=3|
4 #                   -----+---------|
5 C = [[8, 10,  0],\ # s=0 | 8 10  0 |
6      [0,  1, 10],\ #   1 | 0  1 10 |
7      [1,  0,  7]]  #   2 | 1  0  7 |
```

- Here a row corresponds to “from” and a column corresponds to “to”
- The Python numbering for the rows are OK, if we let $s = 0$ by definition
- However, the Python column number is shifted by one, meaning that Python column j corresponds to node number $j + 1$
- We therefore introduce an offset:

```
9 jOffset=-1
```

Specifying the objective function

The total transportation in the model is the sum of what goes out of the source node. This is the same as what goes in to the destination node *n*, but we only need to specify what goes out of the source:

```
s = 0
n = len(C[0]) # Number of the destination node
c = []
for j in range(1,n+1):
    c.append(1) # First row in x-matrix, i.e., sum transportation out of
    the sink s
for i in range(s+1,n):  # Remaining rows, profit coefficients = 0
    for j in range(1,n+1):
        c.append(0)
lp.coefficients(c, False)  # Maximization problem
n_x = len(c)
```

Index function

We need an index function to map “from” → “to”:

```
22 def ndx(i,j): # From state s <= i <= n-1, to state 1 <= j <= n
23     return jOffset + j + i*n # i,j = "logical indexes" ==> indexing to
       use in A-row
```

Specify slack constraints, $x_{ij} <= c_{ij}$

From all nodes i, $0 \leq i < n \rightarrow$ all nodes j, $0 < j \leq n$

```
k = 0 # Keep track of corresponding variable in the decision vector
for i in range(s,n):
    for j in range(1,n+1):
        A_i = np.zeros([n_x])
        A_i[k] = 1 # coefficient in front of x_ij
        lp.upperBound(C[i][jOffset + j],A_i)
        k += 1
```

Mass conservation

Specify “ mass conservation”, i.e., Σ out of node s = Σ in to node n

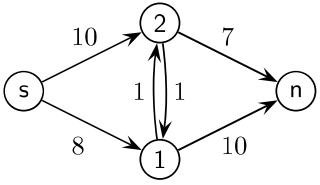
```
33 A_i = np.zeros([n_x])
34 for j in range(1,n+1): # out of node s to node j
35     A_i[ndx(s,j)] = 1
36 for i in range(s,n): # from node i to node n
37     A_i[ndx(i,n)] = -1
38 lp.equalityConstraint(0,A_i)
```

Mass conservation, continued

Specify more “ mass conservation”, i.e., Σ out of node k = Σ in to node k for all nodes except s and n

```
40 for k in range(s+1,n):
41     A_i = np.zeros([n_x])
42     for j in range(1,n+1):
43         A_i[ndx(k,j)] = 1
44     for i in range(s,n):
45         A_i[ndx(i,k)] = -1
46     lp.equalityConstraint(0,A_i)
```

Result




```
x_ij| 1  2  3
----+----------
 0  | 8  8  0 |
 1  | 0  0  9 |
 2  | 1  0  7 |
Max flow 16.0
```

Thank you for your attention