VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MATHEMATICAL MODELING (CO2011)

Assignment

Stochastic Programming and Applications

Advisor:	Nguyen An Khuong
Students:	Ha Kien Hoa - 2252225
	Pham Le Huu Hiep - 2252223
	Nguyen Duc Quoc Ky - 2252415
	Le Huu Anh Quan - 1952942

HO CHI MINH CITY, SEPTEMBER 2023

Contents

1 Member list & Workload

No.	Fullname	Student ID	Problems	Work
1	Pham Le Huu Hiep	2252223	- Problem 1 - Problem 2 - Edit the implementation for Problem 2	30%
2	Le Huu Anh Quan	1952942	- Collect references for Problem 1 - Implement Python code for Problem 1	30%
3	Ha Kien Hoa	2252225	- Write theory for Problem 2	20%
4	Nguyen Duc Quoc Ky	2252415	- Write theory for Problem 2	20%

2 Two-Stage Stochastic linear programming (2-SLP)

2.1 Two-stage SLP Recourse model - (simple form)

The Two-stage Stochastic linear program With Recourse (2-SLPWR) or precisely with penalize corrective action generally described as

$$2-SLP: \quad \min_{\boldsymbol{x}\in\mathbf{X}} \boldsymbol{c}^T \cdot \boldsymbol{x} + \min_{\boldsymbol{y}(\boldsymbol{\omega})\in\boldsymbol{Y}} \mathbf{E}_{\boldsymbol{\omega}}[\boldsymbol{q}\cdot\boldsymbol{y}]$$

or in general

$$2-SLP: \quad \min_{\boldsymbol{x}\in\mathbf{X},\boldsymbol{y}(\boldsymbol{\omega})\in\boldsymbol{Y}} \mathbf{E}_{\boldsymbol{\omega}}\left[\boldsymbol{c}^T\cdot\boldsymbol{x}+v(\mathbf{x},\boldsymbol{\omega})\right]$$
$$\text{with } v(\mathrm{x},\omega) := q\cdot y$$

subject to

$$A\boldsymbol{x}=\boldsymbol{b} \quad \text{First Stage Constraints ,}$$

$$T(\boldsymbol{\omega})\cdot\boldsymbol{x}+W\cdot\boldsymbol{y}(\boldsymbol{\omega})=h(\boldsymbol{\omega}) \text{ Second Stage Constraints or shortly} \quad W\cdot\boldsymbol{y}=h(\boldsymbol{\omega})-T(\boldsymbol{\omega})\cdot\boldsymbol{x}$$

This SLP program specifies a random grand objective function g(x) which has:
(1) the deterministic $f(x)$ - being linear function, while accounting
(2) for a probability function $v(x,\omega)$ associated with various scenarios ω. $y=y(x,\omega)\in\mathbb{R}^p_+$ is named recourse action variable for decision x and realization of ω. Recourse actions are viewed as Penalize corrective actions in SLP.
The Penalize correction is expressed via the mean $Q(x)=E_\omega[v(x,\omega)]$

To solve system (4-4.1) numerically, approaches are based on a random vector α having a finite number of possible realizations, called ***scenarios.***

Expected value $Q(\mathrm{x})$ obviously for a discrete distribution of ω ! So we take $\Omega=\{\omega_k\}$ be a finite set of size S (there are a finite number of scenarios $\omega_1,\ldots,\omega_S\in\Omega$, with respective probability masses p_k). Since $\boldsymbol{y}=\boldsymbol{y}(\boldsymbol{x},\omega)$ so the expectation of $v(\boldsymbol{y})=v(\mathbf{x},\boldsymbol{\omega}):=q\cdot\boldsymbol{y}$ (one cost q for all y_k) is

$$Q(\mathbf{x})=\mathbf{E}_\omega[v(\mathbf{x},\boldsymbol{\omega})]=\sum_{k=1}^{S}p_k q y_k=\sum_{k=1}^{S}p_k v\left(\mathbf{x},\omega_k\right)$$

where

p_k is the density of scenario ω_k, q is single unit penalty cost

$qy_k=v\left(\mathbf{x},\omega_k\right)$ is the penalty cost of using y_k units in correction phase, depends on both the first-stage decision x and random scenarios ω_k.

2.2 Two-stage SLP Recourse model - (canonical form)

Next, we need to characterize the system (4-4.1) in the linear field. The canonical 2-stage stochastic linear program with Recourse can be formulated as

$$2 - SLP: \quad \min_{\boldsymbol{x}} g(\boldsymbol{x}) \text{ with}$$

$$g(\boldsymbol{x}) := \ \boldsymbol{c}^T \cdot \boldsymbol{x} + v(\boldsymbol{y})$$

$$\text{subject to (} s.\ t.) \quad A\boldsymbol{x} = \boldsymbol{b} \text{ where } \boldsymbol{x} \in \boldsymbol{X} \subset \mathbb{R}^n,\ \boldsymbol{x} \geq \ 0$$

$$v(\boldsymbol{z}) := \min_{\boldsymbol{y} \in \mathbb{R}^p_+} \quad \boldsymbol{qy} \quad \text{subject to} \quad W \cdot \boldsymbol{y} = h(\boldsymbol{\omega}) \ - \ T(\boldsymbol{\omega}) \cdot \boldsymbol{x} =: \ \boldsymbol{z}$$

where $v(\boldsymbol{y}) := v(\mathbf{x}, \boldsymbol{\omega})$ is the second-stage value function, and

$\boldsymbol{y} = \boldsymbol{y}(\boldsymbol{x}, \boldsymbol{\omega}) \in \mathbb{R}^p_+$ is a recourse action for decision $\boldsymbol{x}$ and realization of $\boldsymbol{\omega}$.

(1) The expected recourse costs of the decision $\mathbf{x}$ is $Q(\mathbf{x}) \ := \boldsymbol{E}_\omega[v(\mathbf{x}, \boldsymbol{\omega})]$ by Equation (5). [precisely expected costs of the recourse $\boldsymbol{y}(\boldsymbol{\alpha})$, for any policy $\mathbf{x} \in \mathbb{R}^n$.] Hence overally we minimize total expected costs:

$$\min_{\boldsymbol{x} \in \mathbb{R}^n,\ \boldsymbol{y} \in \mathbb{R}^p_+} \quad \boldsymbol{C}^T \cdot \boldsymbol{x} + Q(\boldsymbol{x})$$

(2) We design the 2nd decision variables $\boldsymbol{y}(\boldsymbol{\omega})$ so that we can (tune, modify, or) react to our original constraints (4.2) in an intelligent (or optimal) way: we call it recourse action!

$$\mathbf{x} - - - - - - - - - - T, h, \boldsymbol{\omega} - - - - - - - \longrightarrow \boldsymbol{y}$$

(3) The optimal value of the 2nd-stage LP is $v_* = v(\boldsymbol{y}^*)$, with $\boldsymbol{y}^* = \boldsymbol{y}^*(\boldsymbol{x}, \omega)$ is its optimal solution, here $\boldsymbol{y}^* \in \mathbb{R}^p_+$. The total optimal value is $\boldsymbol{c}^T \cdot \boldsymbol{x}^* + v(\boldsymbol{y}^*)$.

The second-stage problem:
For an observed value (a realization) $\boldsymbol{d} = (d_1, d_2, \ldots, d_n)$ of the above random demand vector D, we can find the best production plan by solving the following stochastic linear program (SLP) with decision variables $\boldsymbol{z} = (z_1, z_2, \ldots, z_n)$ - the number of units produced, and other decision variables $\boldsymbol{y} = (y_1, y_2, \ldots, y_m)$ - the number of parts left in inventory

$$\text{LSP} : \min_{\boldsymbol{z}, \boldsymbol{y}} Z = \sum_{i=1}^{n} (l_i - q_i) z_i - \sum_{j=1}^{m} s_j y_j, \tag{6}$$

where $s_j < b_j$ (defined as pre-order cost per unit of part j), and
$x_j, j = 1, \ldots, m$ are <u>the numbers of parts to be ordered before production</u>.

$$\text{subject to} = \begin{cases} y_j = x_j - \displaystyle\sum_{i=1}^{n} a_{i_j} z_i, & j = 1, \ldots, m \\ 0 \leq z_i \leq d_i, \quad i = 1, \ldots, n; & y_j \geq 0, j = 1, \ldots, m. \end{cases}$$

The whole model (of the second-stage) can be equivalently expressed as

$$\text{MODEL} = \begin{cases} \min_{\boldsymbol{z},\boldsymbol{y}} Z = \boldsymbol{c}^T \cdot \boldsymbol{z} - \boldsymbol{s}^T \cdot \boldsymbol{y} \\ \text{with } \boldsymbol{c} = (c_i := l_i - q_i) \text{ are cost coefficients} \\ \boldsymbol{y} = \boldsymbol{x} - A^T\ \boldsymbol{z}, \text{ where } A = [a_{ij}] \text{ is matrix of dimension } n \times m, \\ 0 \leq \boldsymbol{z} \leq \boldsymbol{d}, \quad \boldsymbol{y} \geq 0. \end{cases} \tag{7}$$

Observe that the solution of this problem, that is, the vectors $\boldsymbol{z}$, $\boldsymbol{y}$ depend on realization $\boldsymbol{d}$ of the random demand $\boldsymbol{\omega} = \boldsymbol{D}$ as well as on the 1st-stage decision $\boldsymbol{x} = (x_1, x_2, \dots, x_m)$.

The first-stage problem:
The whole 2-SLPWR model is based on a popular rule that **production** $\geq$ **demand**.
Now follow distribution-based approach, we let $Q(\boldsymbol{x}) := \boldsymbol{E}[Z(\boldsymbol{z}, \boldsymbol{y})] = \boldsymbol{E}_{\boldsymbol{\omega}}[\boldsymbol{x}, \boldsymbol{\omega}]$ denote the optimal value of problem (6). Denote

$\underline{\boldsymbol{b} = (b_1, b_2, \dots, b_m) \text{ built by preorder cost } b_j \text{ per unit of part } j}$ (before the demand is known).

The quantities x_j are determined from the following optimization problem

$$\min g(\mathbf{x}, \boldsymbol{y}, \boldsymbol{z}) = \boldsymbol{b}^T \cdot x + Q(\boldsymbol{x}) = \boldsymbol{b}^T \cdot \boldsymbol{x} + \boldsymbol{E}[Z(\boldsymbol{z})] \tag{8}$$

where $Q(\boldsymbol{x}) = \boldsymbol{E}_{\boldsymbol{\omega}}[Z] = \sum_{i=1}^{n} p_i\ c_i\ z_i$ is taken w. r. t. the probability distribution of $\boldsymbol{\omega} = \boldsymbol{D}$.

The first part of the objective function represents the pre-ordering cost and $\boldsymbol{x}$. In contrast, the second part represents the expected cost of the optimal production plan (7), given by the updated ordered quantities $\boldsymbol{z}$, already employing random demand $\boldsymbol{D} = \boldsymbol{d}$ with their densities.

3 Min-cost Flow Problem

3.1 Network Flows

Many problems in computer science can be represented by a graph consisting of nodes and links between them. Examples are **network flow** problems, which involve transporting goods or material across a network, such as a railway system.

You can represent a network flow by a graph whose **nodes** are cities and whose **arcs** are rail lines between them. (They're called **flows** because their properties are similar to those of water flowing through a network of pipes.)

A key constraint in network flows is that each arc has a **capacity** - the maximum amount that can be transported across the arc in a fixed period of time.

The **maximum flow problem** is to determine the maximum total amount that can be transported across all arcs in the network, subject to the capacity constraints.

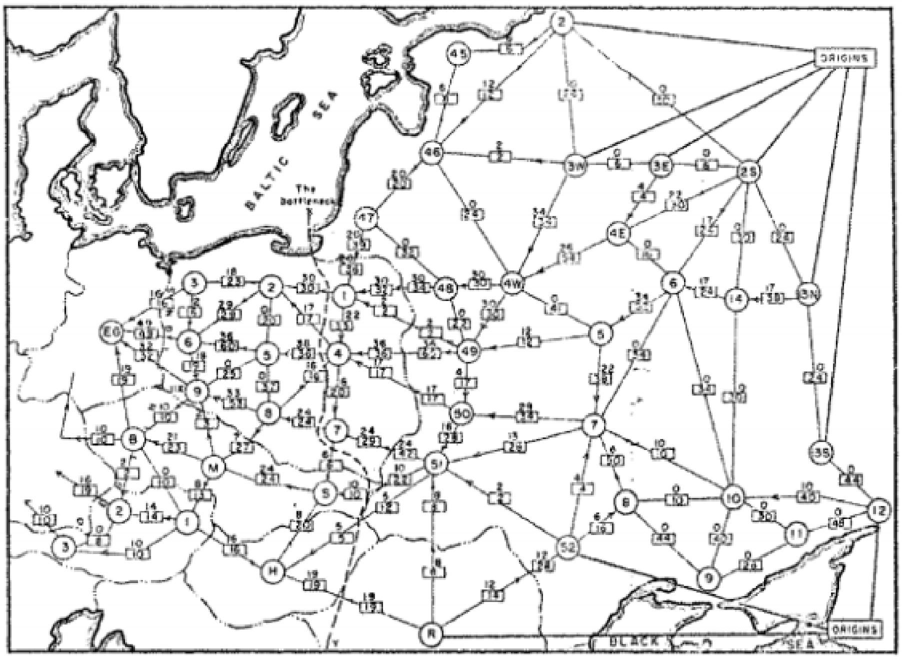


Figure 1: The first person to study this problem was the Russian mathematician A.N. Tolstoi, in the 1930s. The map above shows the actual railway network for which he wanted to find a maximum flow.

3.2 Model decomposition

From the original model it can be observed that the coupling constraint is a hard constraint. This constraint characterizes the relationship between selection of a physical link and corresponding scenario-based time-dependent arcs. Hence, we introduce the Lagrangian multiplier $\alpha^s_{i,j}(t), (i,j) \in A, s = 1, 2, ..., S, t$ for the coupling constraint, and then this constraint can be relaxed into the objective function in the following form: $\sum\limits_{s=1}^{S} \sum\limits_{t} \sum\limits_{(i,j)\in A} \alpha^S_{i,j}(\boldsymbol{y}^s_{i,j}(t) - \boldsymbol{x}_{i,j}$

$$\begin{cases} min \sum_{(i,j)} \in A\boldsymbol{p}_{ij}\boldsymbol{x}_{ij} + \sum_{s=1}^{S}(\mu_s \sum_{(i,j)} \in A_s c^s_{i,j}(t)\boldsymbol{y}^s_{i,j}(t))+ \\ \sum_{s=1}^{S} \sum_t \leq T \sum_{i,j} \in A\alpha^s_{i,j}(t) - \boldsymbol{x}_{i,j} \\ s.t \\ \sum_{(i,j)} \in A\boldsymbol{x}_{i,j} - \sum_{ji\in A} \boldsymbol{x}_{ij} = \boldsymbol{d}_i, \forall_i \in V \\ 0 \leq \boldsymbol{x}_{ij} \leq u_{ij}, \forall(i,j) \in A \\ \sum_{(i_t,j_{t'}\in A_s} \boldsymbol{y}_{ij}{}_s(t) - \sum (j_{t'}, i_t) \in A_s\boldsymbol{y}^s_{ij}(t') = \boldsymbol{d}^s_i(t), \forall i \in V, \\ t \in \{0,1,...,T\}, s = 1,2,...,S \\ 0 \leq \boldsymbol{y}^s_{ij}(t) \leq \boldsymbol{u}^s_{ij}(t), \forall(i,j) \in A, t \in \{0,1,...,T\}, s = 1,2,...,S \end{cases} \tag{13}$$

It is worthwhile to note that the variables X and Y can be separated from each other in above relaxed model (13). That is, by combining similar terms, the relaxed model is decomposed into two subproblems as follows:

3.3 SubProblem 1: Min-cost Flow Problem

Obviously, the first sub-problem can be regarded as a min-cost flow problem, and its form is given as follows:

$$\begin{cases} minSP1() = \sum_{(i,j)\in A}(P_{(i,j)} - \sum_{s=1}^{S}\sum_{t\leq T}\alpha_{(i,j)}^{S}(t))\chi_{(i,j)} \\ s.t \\ sum_{(i,j)} \in A\chi_{(i,j)} - \sum_{(j,i)} \in A\chi_{(i,j)} = d \in i, \forall_i \in V \\ 0 \leq \chi_{i,j} \leq u_{i,j}, \forall_{(i,j)} \in A \end{cases}$$

The objective function of subProblem 1 can be defined as $g_{i,j} := p_{i,j} - \sum_{s=1}^{S}\sum_{t\leq T}\alpha_{i,j}^{s}(t)$ to represent the generalized cost of each link. Therefore, the sub-Problem 1 can be solved by the successive shortest path algorithm. For convenience, the optimal objective value of sub-Problem 1 is abbreviated as $Z_{SP1}^{*}(\alpha)$ The successive shortest path algorithm was proposed by Jewell (1962), Iri (1960), Busacker and Gowen (1961), et al. The goal of this algorithm is to calculate the minimum cost flow in the network $G = (V, A, C, U, D)$ (Xie, Xing, & Wang, 2009). Eventually, the successive shortest path algorithm for subProblem 1 is developed in Algorithm 1.

3.4 SubProblem 2: Time-Dependent Min-cost Flow Problem

The second subproblem of the relaxed model (13) is associated with the decision variables $\mathbf{Y}$, and its optimal objective value of the problem is abbreviated as $Z_{SP2}^{*}(\alpha)$, shown as follows:

$$\begin{cases} \min\ SP2(\alpha) = \sum_{s=1}^{S}\sum_{(i,j)\in A}\left(\sum_{t\in\{0,1,\ldots,T\}}\mu_s \cdot c_{ij}^{s}(t) + \sum_{t\leq\tilde{T}}\alpha_{ij}^{s}(t)\right)\mathbf{y}_{ij}^{s}(t) \\ s.\ t. \\ \sum_{(i_t j_{t'})\in A_s}\mathbf{y}_{ij}^{s}(t) - \sum_{(i_t, j_{t'})\in A_s}\mathbf{y}_{ij}^{s}(t') = d_i^{s}(t), \forall i \in V, \\ t \in \{0, 1, \ldots, T\}, s = 1, 2, \ldots, S \\ 0 \leq \mathbf{y}_{ij}^{s}(t) \leq \mathbf{u}_{ij}^{s}(t), \forall (i,j) \in A, t \in \{0, 1, \ldots, T\}, s = 1, 2, \ldots, S \end{cases} \tag{15}$$

3.5 Algorithm 1: Successive shortest path algorithm for min-cost flow problem.

Step 1: Take variable x as a feasible flow between any OD and it has the minimum delivery cost in the feasible flows with the same flow value.

Step 2: The algorithm will terminate if the flow value of x reaches v or there is no minimum cost path in the residual network $(V, A(x), C(x), U(x), D)$; otherwise, the shortest path with the maximum flow is calculated by label-correcting algorithm, and then go to Step 3. The functions $A(x), C(x), U(x)$ in the residual network can be defined as:

$$A(x) = \{(i,j)|(i,j) \in A, x_{ij} < u_{ij}\} \cup A(x) = \{(i,j)|(i,j) \in A, x_{ij} > 0\}$$

$$C(x) = \begin{cases} c_{ij}, i, j \in A, x_{ij} < u_{ij} \\ -c_{ji}, j, i \in A, x_{ji} > 0 \end{cases}$$

$$U_{ij}(x) = \begin{cases} u_{ij}, i, j \in A, x_{ij} < u_{ij} \\ x_{ji}, j, i \in A, x_{ji} > 0 \end{cases}$$

Step 3: Increase the flow along the minimum cost path. If the increased flow value does not exceed v, go to Step 2.

1. **A(x):** Flow function - the remaining flow on the edges in the residual network.
2. **C(x):** Cost function - the cost on the edges of the residual network.
3. **U(x):** Capacity function - the remaining capacity on the edges of the residual network.
4. **D:** Set of vertices in the residual network.
5. **V:** A specific value, potentially an upper limit for the flow value or another value depending on the specific problem.

SubProblem 2 can be further decomposed into a total of S sub problems, each of which can be referred to as the min-cost flow problem with time-dependent link travel times and capacities, namely:

$$\begin{cases} \min\ SP2(\alpha, s) = \sum_{(i,j)\in A} \left(\sum_{t\in\{0,1,\dots,T\}} \mu_s \cdot c_{ij}^s(t) + \sum_{t\leq\tilde{T}} \alpha_{ij}^s(t) \right) \mathbf{y}_{ij}^s(t) \\ s.\ t. \\ \sum_{(i_t j_{t'})\in A_s} \mathbf{y}_{ij}^s(t) - \sum_{(i_t, j_{t'})\in A_s} \mathbf{y}_{ij}^s(t') = d_t^s(t), \forall i \in V, t \in \{0, 1, \dots, T\} \\ 0 \leq \mathbf{y}_{ij}^s(t) \leq \mathbf{u}_{ij}^s(t), \forall (i,j) \in A, t \in \{0, 1, \dots, T\} \end{cases} \tag{16}$$

For each scenario $s \in \{1, 2, \dots, \}$ subproblem (16) has a similar structure as subproblem (14) with time-dependent link cost $C_{ij}^s(t)$ and link capacity $u_{ij}^s(t)$ and the generalized cost $g_{ij}^s(t)$. Since the considered time period T is divided into two time stages, the generalized cost is defined as a piecewise function:

$$g_{ij}^s(t) = \begin{cases} \mu_s \cdot c_{ij}^s(t) + \alpha_{ij}^s(t), t \leq \tilde{T} \\ \mu_s \cdot c_{ij}^s(t), \qquad \tilde{T} < t \leq T \end{cases}$$

Since subproblem (16) is a time-dependent min-cost flow problem, and thus the algorithm 1 should be modified in Step 2. Firstly, parameters $A(y(t)), C(y(t))$ and $U(y(t))$ in the residual network $N(y(t))$ are defined as follows:

$$A_s(y(t)) = \{(i_t, j_{t'}) | (i_t, j_{t'}) \in A_s, y_{ij}^s < u_{ij}^s\} \cup \{(j_{t'}, i_t) | (j_{t'}, i_t) \in A_s, y_{ij}^s(t) > 0\}, s = 1, 2, \dots, S$$

$$c_{ij}^s(y(t)) = \begin{cases} c_{ij}^s(t), (i_t, j_{t'}) \in A_s, y_{ij}^s(t) < u_{ij}^s(t), t \in \{0, 1, \dots, T\} \\ -c_{ji}^s(t'), (j_{t'}, i_t) \in A_s, \forall \{t' \in \{0, 1, \dots, T\} | y_{ji}^s(t') > 0\}, \\ s = 1, 2, \dots, S \\ T, (j_{t'}, i_t) \in A_s, \forall \{t' \in \{0, 1, \dots, T\} | y_{ji}^s(t') = 0\} \end{cases}$$

$$u_{ij}^s(y(t)) = \begin{cases} u_{ij}^s(t) - y_{ij}^s(t), (i_t, j_{t'}) \in A_s, y_{ij}^s(t) < u_{ij}^s(t), t \in \{1, 2, \dots, T\} \\ y_{ji}^s(t), (j_{t'}, i_t) \in A_s, \forall \{t' \in \{0, 1, \dots, T\} | y_{ji}^s(t') > 0\}, \\ s = 1, 2, \dots, S \\ 0, (j_{t'}, i_t) \in A_s, \forall \{t' \in \{0, 1, \dots, T\} | y_{ji}^s(t') = 0\} \end{cases}$$

Secondly, the modified label-correcting algorithm (Ziliaskopoulos & Mahmassani, 1992) will be adopted to find the time-dependent min-cost path in the residual network.

By solving the subproblem (14) and (15) with the relaxation solution $\boldsymbol{X}$ and $\boldsymbol{Y}$, the optimal objective value Z^*_{LR} for the relaxed model (13) with a set of given Lagrangian multiplier vector can be expressed as follows:

$$Z^*_{LR}(\alpha) = Z^*_{SP1}\alpha + Z^*_{SP2}(\alpha) \qquad (17)$$

Obviously, the optimal objective value of the relaxed model (13) is the lower bound of the optimal objective value of the original model (10). In order to obtain a high-quality solution, it is needed to obtain a lower bound which is close to the optimal objective value of the original model. That is, the greatest possible lower bound should be obtained, and the expression is given as follows:

$$Z_{LD}(\alpha^*) = max_{\alpha \geq 0} Z_{LR}(\alpha) \qquad (18)$$

3.6 Label-correcting algorithm

Let us begin with some notation. Let $G = (V, E)$ be a graph (which could be undirected or directed) and $a_{ij} : (i, j) \in E$ be a set of non–negative weights on the edges. We distinguish two nodes in the graph, namely, the origin $s \in V$ and the destination $t \in V$. A node $j \in V$ is called a child of node $i \in V$ if there is an edge from i to j. Note that if the graph is undirected and (i, j) is an edge, then i is a child of j and j is a child of i. Our goal is to find a shortest path from s to t.

The idea behind the label correcting algorithms is to progressively discover shorter paths from the origin to every other node i. This is achieved by maintaining a label d_i for each node i, which represents an upper bound on the shortest distance from the origin s to node i. The label d_t of the destination t is maintained in a variable called Upper. As we shall see, the variable Upper plays a special role in the algorithm.

As the algorithm progresses, if a path from the origin to node i is discovered and whose distance is smaller than d_i , then the label of node i will get corrected. To facilitate bookkeeping, the algorithm maintains a list of nodes called Open. The list Open contains nodes that will be examined by the algorithm and are possible candidates to be included in the shortest path. When the list Open becomes empty, the algorithm terminates.

To illustrate the label correcting algorithm, consider again the following problem:

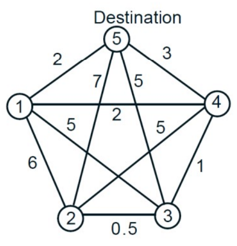


Figure 2: A shortest path problem with $s = 2$ and $t = 5$

Suppose that we want to find the shortest path from $s = 2$ to $t = 5$ using the label correcting algorithm with best–first search. We trace the iterations as follows:

Iteration	Open Node	Removed	Label Updates
1	{2}	2	$d_1 = 6$ $(2 \to 1)$, $d_3 = 0.5$ $(2 \to 3)$, $d_4 = 5$ $(2 \to 4)$, Upper $= 7$ $(2 \to 5)$
2	{1, 3, 4}	3	$d_4 = 1.5$ $(3 \to 4)$, Upper $= 5.5$ $(3 \to 5)$
3	{1, 4}	4	$d_1 = 3.5$ $(4 \to 1)$, Upper $= 4.5$ $(4 \to 5)$
4	{1}	1	
5	∅		

The shortest path distance is given by $Upper = 4.5$, and the shortest path can be constructed by tracing the parents:

$$5 \leftarrow 4 \leftarrow 3 \leftarrow 2$$

Simplest case

First we only consider the simplest case, where the graph is oriented, and there is at most one edge between any pair of vertices (e.g. if (i, j) is an edge in the graph, then (j, i) cannot be part in it as well).

Let U_{ij} be the capacity of an edge (i, j)if this edge exists. And let C_{ij}be the cost per unit of flow along this edge (i, j). And finally let $F_{i,j}$ be the flow along the edge (i, j). Initially all flow values are zero.

We modify the network as follows: for each edge (i, j) we add the reverse edge (j, i) to the network with the capacity $U_{ji} = 0$ and the cost $C_{ji} = -C_{ij}$. Since, according to our restrictions, the edge (j, i) was not in the network before, we still have a network that is not a multigraph (graph with multiple edges). In addition we will always keep the condition $F_{ji} = -F_{ij}$ true during the steps of the algorithm.

We define the residual network for some fixed flow F as follow (just like in the Ford-Fulkerson algorithm): the residual network contains only unsaturated edges (i.e. edges in which $F_{ij} < U_{ij}$), and the residual capacity of each such edge is$R_{ij} = U_{ij} - F_{ij}$

Undirected graphs / multigraphs

In the context of an undirected graph or a multigraph, the underlying conceptual framework of the algorithm remains unchanged from the description above. The algorithm is applicable to these types of graphs as well, albeit with some additional implementation complexities.

For undirected edges, such as (i, j), they are treated as two directed edges, (i, j) and (j, i), each possessing identical capacities and values. Consequently, the algorithm, designed for directed graphs, transforms an undirected edge into four directed edges, resulting in the creation of a multigraph.

Dealing with multiple edges involves maintaining separate flow values for each of them. When searching for the shortest path, it becomes crucial to consider which specific multiple edge is utilized in the path. Therefore, instead of just storing an ancestor array, it becomes necessary to include the edge number through which the path was reached in addition to the ancestor information.

Moreover, as the flow increases along a particular edge, it is essential to simultaneously reduce the flow along the corresponding back edge. Due to the existence of multiple edges, it becomes imperative to store the edge number associated with the reversed edge for each individual edge.

Apart from these considerations related to undirected graphs or multigraphs, there are no other significant obstacles in applying the algorithm to these scenarios.

Complexity

The algorithm exhibits an exponential complexity with respect to the input size. Specifically, in the worst-case scenario, it may only augment the flow by 1 unit per iteration, requiring $O(F)$ iterations to discover a minimum-cost flow of magnitude F. Consequently, the overall runtime is $O(F \cdot T)$, where T denotes the time needed to determine the shortest path from the source to the sink.

4 Problem

4.1 Problem 1

Write a linear program using the GAMSPy library to make a concrete model:
1. Use only a single (.py) file.
2. Constants
$n = 8$ - products
$S = 2$ - the number of scenarios
$p_s = 1/2$ - density
$m = 5$ - the number of parts to be ordered before production

3. Randomly simulated values
vector b - previous order cost per unit of part
vector s - salvage value per unit of part
vector l - additional cost per unit of product
vector q - unit selling price of product
vector c = difference between vector l and q - the cost coefficients
matrix of requirements $A = [a_{ij}]$ is matrix of dimension n × m
vector D - a random demand vector follows the binomial distribution Bin(10, 1/2)

4. Sets and Parameters
Set i has records from 0 to 7
Set j has records from 0 to 4
Parameter b, s, l, q, a, d, d2

5. Decision Variables
$x = (x_1, x_2, ..., x_m)$ - the numbers of parts to be ordered before production
$y = (y_1, y_2, ..., y_m)$ - the number of parts left in inventory in day 1
$y2 = (y2_1, y2_2, ..., y2_m)$ - the number of parts left in inventory in day 2
$z = (z_1, z_2, ..., z_n)$ - the number of units produced in day 1
$z2 = (z2_1, z2_2, ..., z2_n)$ - the number of units produced in day 2

6. Demands and Constraints
a. Demands:
First Demand: $z_i \leq d_i$
Second Demand: $z2_i \leq d2_i$
b. Constraints:
First Constraint: $y_j = x_j - a_{ij} * z_i$
Second Constraint: $y2_j = x_j - a_{ij} * z2_i$

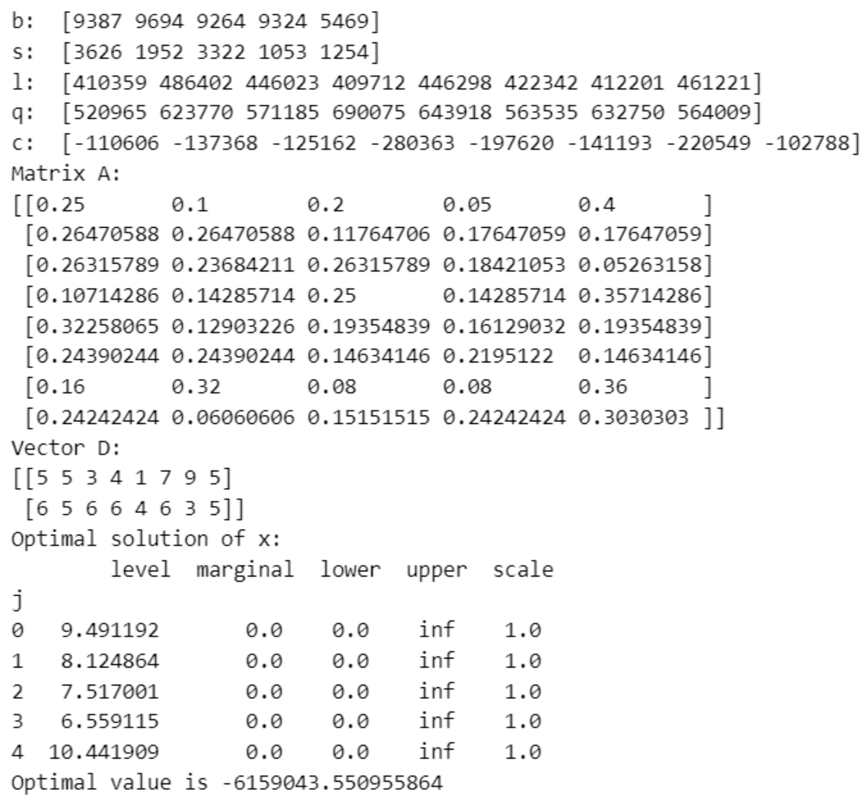
7. Model Objective

$$\text{Objective Function} = \sum_{j=0}^{4} b_j * x_j + \sum_{i=0}^{7} p_s * c_i * z_i - \sum_{j=0}^{4} s_j * y_j + \sum_{i=0}^{7} p_s * c_i * z2_i - \sum_{j=0}^{4} s_j * y2_j$$

Implementing the simulation for the numerical model:

```
import random
import sys
import gamspy as gp
from gamspy import Container, Set, Parameter, Variable, Equation, Model, Sense, Sum
import pandas as pd
import numpy as np

# Set some specific values
S = 2 # quantity of scenarios
p_s = 1/2 # probability of occurrence of each scenario
n = 8 # quantity of products
m = 5 # quantity of parts to be ordered before production

# Function to generate random b, s, l, q, A
vector_b = np.random.randint(5000, 9999, size=(m)) # pre-order cost per unit of part
vector_s = np.random.randint(1000, 4999, size=(m)) # salvage value per unit of part
vector_l = np.random.randint(400000, 499999, size=(n)) # additional cost per unit of
    product
vector_q = np.random.randint(500000, 699999, size=(n)) # unit selling price of product
vector_c = vector_l - vector_q # the cost coefficients

matrix_A = np.zeros([n, m])
sum_i = [0, 0, 0, 0, 0, 0, 0, 0]
for i in range(n):
    for j in range(m):
        matrix_A[i][j] = random.randint(1, 10)
for i in range(n):
    for j in range(m):
            sum_i[i] = sum_i[i] + matrix_A[i][j]
for i in range(n):
    for j in range(m):
        matrix_A[i][j] = matrix_A[i][j] / sum_i[i]

print('b: ', vector_b)
print('s: ', vector_s)
print('l: ', vector_l)
print('q: ', vector_q)
print('c: ', vector_c)
print('Matrix A: ')
print(matrix_A)

# The random demand vector following Binomial distribution
vector_D = np.random.binomial(10, p_s, (S, n))
print('Vector D:')
print(vector_D)

model_m = Container()
# Create sets and parameters in the model
i = Set(container=model_m, name="i", description="A unit of product", records=["0","1",
    "2", "3", "4", "5", "6", "7"])
j = Set(container=model_m, name="j", description="A unit of part to be ordered",
    records=["0","1", "2", "3", "4"])

b = Parameter(container=model_m, name="b", description="pre-order cost per unit of
    part", domain=j, records=np.array(vector_b))
s = Parameter(container=model_m, name="s", description="salvage value per unit of
    part", domain=j, records=np.array(vector_s))
l = Parameter(container=model_m, name="l", description="additional cost per unit of
    product", domain=i, records=np.array(vector_l))
q = Parameter(container=model_m, name="q", description="unit selling price of product",
    domain=i, records=np.array(vector_q))
a = Parameter(container=model_m, name="a", description="matrix of requirements",
    domain=[i, j], records=np.array(matrix_A))
d = Parameter(container=model_m, name="d", description="the first demand for the
    products", domain=i, records=np.array(vector_D[0]))
d2 = Parameter(container=model_m, name="d2", description="the second demand for the
    products", domain=i, records=np.array(vector_D[1]))

# Create some variables in the model
x = Variable(container=model_m, name="x", domain=j, type="Positive",
    description="numbers of parts to be ordered before production")
y = Variable(container=model_m, name="y1", domain=j, type="Positive",
    description="numbers of parts left in inventory in day 1")
y2 = Variable(container=model_m, name="y2", domain=j, type="Positive",
    description="numbers of parts left in inventory in day 2")
z = Variable(container=model_m, name="z1", domain=i, type="Positive",
    description="number of units produced in day 1")
z2 = Variable(container=model_m, name="z2", domain=i, type="Positive",
    description="number of units produced in day 2")

# Add model demands and constraints
first_demand = Equation(container=model_m, name="The_first_demand",
    domain=i,definition= z[i] <= d[i])
second_demand = Equation(container=model_m, name="The_second_demand",
    domain=i,definition = z2[i] <= d2[i])
first_constraint = Equation(container=model_m, name="The_first_constraint",
    domain=j,definition = y[j] == x[j] - Sum(i,a[i,j]*z[i]))
second_constraint = Equation(container=model_m, name="The_second_constraint",
    domain=j,definition = y2[j] == x[j] - Sum(i,a[i,j]*z2[i]))

# Add a model objective
the_object_func = Sum(j, b[j]*x[j]) + p_s*(Sum(i, (l[i]-q[i])*z[i]) - Sum(j,s[j]*y[j]))
    + p_s*(Sum(i, (l[i]-q[i])*z2[i]) - Sum(j,s[j]*y2[j]))

# Create a model instance for First-Stage Problem and solve
first_stage = Model(container=model_m, name="first_stage",
    equations=model_m.getEquations(), problem="LP", sense=Sense.MIN,
    objective=the_object_func)
first_stage.solve()

# Print solutions
print("Optimal solution of x:\n",x.records.set_index(["j"]))
print("Optimal value is", first_stage.objective_value)
```

Eventually, we obtain the optimal value as follow:



```
b:  [9387 9694 9264 9324 5469]
s:  [3626 1952 3322 1053 1254]
l:  [410359 486402 446023 409712 446298 422342 412201 461221]
q:  [520965 623770 571185 690075 643918 563535 632750 564009]
c:  [-110606 -137368 -125162 -280363 -197620 -141193 -220549 -102788]
Matrix A:
[[0.25       0.1        0.2        0.05       0.4       ]
 [0.26470588 0.26470588 0.11764706 0.17647059 0.17647059]
 [0.26315789 0.23684211 0.26315789 0.18421053 0.05263158]
 [0.10714286 0.14285714 0.25       0.14285714 0.35714286]
 [0.32258065 0.12903226 0.19354839 0.16129032 0.19354839]
 [0.24390244 0.24390244 0.14634146 0.2195122  0.14634146]
 [0.16       0.32       0.08       0.08       0.36      ]
 [0.24242424 0.06060606 0.15151515 0.24242424 0.3030303 ]]
Vector D:
[[5 5 3 4 1 7 9 5]
 [6 5 6 6 4 6 3 5]]
Optimal solution of x:
        level  marginal  lower  upper  scale
j
0   9.491192       0.0    0.0    inf    1.0
1   8.124864       0.0    0.0    inf    1.0
2   7.517001       0.0    0.0    inf    1.0
3   6.559115       0.0    0.0    inf    1.0
4  10.441909       0.0    0.0    inf    1.0
Optimal value is -6159043.550955864
```

Figure 3: Randomly Generated Vectors, Matrix and Optimal Solution

4.2 Problem 2

Implementation

```python
from sys import maxsize
from typing import List

# Stores the found edges
found = []

# Stores the number of nodes
N = 0

# Stores the capacity
# of each edge
cap = []

flow = []

# Stores the cost per
# unit flow of each edge
cost = []

# Stores the distance from each node
# and picked edges for each node
dad = []
dist = []
pi = []

INF = maxsize // 2 - 1

# Function to check if it is possible to
# have a flow from the src to sink
def search(src: int, sink: int) -> bool:

   # Initialise found[] to false
   found = [False for _ in range(N)]

   # Initialise the dist[] to INF
   dist = [INF for _ in range(N + 1)]

   # Distance from the source node
   dist[src] = 0

   # Iterate until src reaches N
   while (src != N):
      best = N
      found[src] = True

      for k in range(N):

         # If already found
         if (found[k]):
          continue

        # Evaluate while flow
        # is still in supply
        if (flow[k][src] != 0):

          # Obtain the total value
          val = (dist[src] + pi[src] -
              pi[k] - cost[k][src])

          # If dist[k] is > minimum value
          if (dist[k] > val):

            # Update
            dist[k] = val
            dad[k] = src

        if (flow[src][k] < cap[src][k]):
          val = (dist[src] + pi[src] -
              pi[k] + cost[src][k])

          # If dist[k] is > minimum value
          if (dist[k] > val):

            # Update
            dist[k] = val
            dad[k] = src

        if (dist[k] < dist[best]):
          best = k

      # Update src to best for
      # next iteration
      src = best

    for k in range(N):
      pi[k] = min(pi[k] + dist[k], INF)

    # Return the value obtained at sink
    return found[sink]

# Function to obtain the maximum Flow
def getMaxFlow(capi: List[List[int]],
        costi: List[List[int]],
        src: int, sink: int) -> List[int]:

    global cap, cost, found, dist, pi, N, flow, dad
    cap = capi
    cost = costi

    N = len(capi)
    found = [False for _ in range(N)]
    flow = [[0 for _ in range(N)]
```

```
        for _ in range(N)]
  dist = [INF for _ in range(N + 1)]
  dad = [0 for _ in range(N)]
  pi = [0 for _ in range(N)]

  totflow = 0
  totcost = 0

  # If a path exist from src to sink
  while (search(src, sink)):

    # Set the default amount
    amt = INF
    x = sink

    while x != src:
      amt = min(
        amt, flow[x][dad[x]] if
        (flow[x][dad[x]] != 0) else
        cap[dad[x]][x] - flow[dad[x]][x])
      x = dad[x]

    x = sink

    while x != src:
      if (flow[x][dad[x]] != 0):
        flow[x][dad[x]] -= amt
        totcost -= amt * cost[x][dad[x]]

      else:
        flow[dad[x]][x] += amt
        totcost += amt * cost[dad[x]][x]

      x = dad[x]

    totflow += amt

  # Return pair total cost and sink
  return [totflow, totcost]


if __name__ == "__main__":

  s = 0
  t = 4

  cap = [ [ 0, 3, 1, 0, 3 ],
      [ 0, 0, 2, 0, 0 ],
      [ 0, 0, 0, 1, 6 ],
      [ 0, 0, 0, 0, 2 ],
      [ 0, 0, 0, 0, 0 ] ]

  cost = [ [ 0, 1, 0, 0, 2 ],
        [ 0, 0, 0, 3, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 1 ],
        [ 0, 0, 0, 0, 0 ] ]

ret = getMaxFlow(cap, cost, s, t)

print("Maximum flow:", ret[0])
print("Total cost:", ret[1])
```

Effectiveness

Test case 0:
Max flow: 6
Cost: 8

```
s = 0
t = 4
cap = [ [ 0, 3, 1, 0, 3 ],
      [ 0, 0, 2, 0, 0 ],
      [ 0, 0, 0, 1, 6 ],
      [ 0, 0, 0, 0, 2 ],
      [ 0, 0, 0, 0, 0 ] ]
cost = [ [ 0, 1, 0, 0, 2 ],
      [ 0, 0, 0, 3, 0 ],
      [ 0, 0, 0, 0, 0 ],
      [ 0, 0, 0, 0, 1 ],
      [ 0, 0, 0, 0, 0 ] ]
ret = getMaxFlow(cap, cost, s, t)
print("Maximum flow:", ret[0])
print("Total cost:", ret[1])
```

Test case 1:
Max flow: 0
Cost: 0

```
s = 0
t = 3

cap = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]

cost = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]

ret = getMaxFlow(cap, cost, s, t)

print("Maximum flow:", ret[0])
print("Total cost:", ret[1])
```

Test case 2:
Max flow: 3
Cost: 12

```
s = 0
t = 4
```

```
cap = [
    [0, 2, 1, 0, 1],
    [0, 0, 0, 3, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 2],
    [0, 0, 0, 0, 0]
]

cost = [
    [0, 1, 2, 0, 1],
    [0, 0, 0, 3, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 2],
    [0, 0, 0, 0, 0]
]

ret = getMaxFlow(cap, cost, s, t)

print("Maximum flow:", ret[0])
print("Total cost:", ret[1])
```

Test case 3:
Max flow: 5
Cost: 59

```
s = 0
t = 9

cap = [
    [0, 5, 3, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 2, 4, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 7, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 5, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 4, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 3, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 6],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 4],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]

cost = [
    [0, 2, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 3, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 4, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 5, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 3, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 2, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 2],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]

ret = getMaxFlow(cap, cost, s, t)

print("Maximum flow:", ret[0])
print("Total cost:", ret[1])
```

References

[1] L. Wang, “A two-stage stochastic programming framework for evacuation planning in disaster responses,” Computers & Industrial Engineering, vol. 145, p. 106458, 2020.

[2] https://gamspy.readthedocs.io/en/latest/user/index.html

[3] https://developers.google.com/optimization/flow/maxflow?hl=vi

[4] https://cp-algorithms.com/graph/min_cost_flow.html