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

MATHEMATICAL MODELLING (CO2011)

Assignment

Stochastic Programming and Applications

Adivsors: Nguyen An Khuong & Nguyen Van Minh Man
Students: Van Duy Anh – 2252045
Nguyen Doan Hai Bang – 2252078
Nguyen Quang Duy – 2252120
Huynh Mai Quoc Khang – 2252293
Tran Anh Khoa – 2252362

Ho Chi Minh City, December 2023

Member List & Workload

No	Full name	Student ID	Problems	Percentage of work
1	Van Duy Anh	2252045	Theory Problem 2	20%
2	Nguyen Doan Hai Bang	2252078	Theory Problem 2	20%
3	Nguyen Quang Duy	2252120	Theory Problem 1, Implement Problem 1	20%
4	Huynh Mai Quoc Khang	2252293	Implement Problem 2	20%
5	Tran Anh Khoa	2252362	Theory Problem 1, Latex File	20%

Contents

1 Introduction

Deterministic linear programming relies on inflexible assumptions, often unsuitable for real-world workflows involving uncertainties. In response, stochastic programming (SP) proves invaluable, providing a modeling approach that considers variability. This assignment empowers students by applying SP, comparing it with simpler linear programming, and addressing complex problems through systematic analysis and GAMS programming, with a focus on quantifying tradeoffs in the presence of variability.

The first task involves formulating a two-stage SP model for production planning, accounting for uncertain demand. The team breaks down the problem, implementing the optimization algorithm in GAMS to gain insights into system dynamics.

The second task revisits Li Wang's two-stage SP model for evacuation routing, utilizing past disaster data for pre-event planning and adaptive post-event rerouting. GAMS is employed for implementation, providing hands-on experience in understanding tradeoffs.

Both analyses use cutting-edge SP techniques tailored to domain data, showcasing practical skills in planning optimization under uncertainty. Acquired skills highlight the superiority of SP over conventional methods, demonstrating improvements in cost, timing, robustness, and computational efficiency.

Stochastic programming, a mathematical optimization approach, excels in addressing complex decision-making with uncertain parameters. By incorporating probabilistic modeling, it considers variability and associated probability distributions. The goal is to determine optimal decisions considering both expected values and associated risks. This involves formulating problems as mathematical programs with explicit recourse, accounting for additional decisions after observing uncertain parameters. Given the prevalence of uncertainty, stochastic programming finds applications in finance, transportation, and energy optimization.

This collaborative assignment requires teams to apply stochastic programming, presenting theoretical insights and computational outcomes. The first problem involves constructing numerical models using simulated data and software like GAMSPy. The second problem explores Li Wang's two-stage stochastic programming framework for disaster response evacuation planning. By definition, a Stochastic Linear Program (SLP) is

$$\min Z = g(x) = c^T x$$

Such that

$$Ax = b$$

$$Tx \geq h$$

With $x = (x_1, x_2, \ldots, x_n)$ as decision variables, a certain real matrix A, vector b for deterministic constraints, and with random parameters T, h in $Tx \geq h$ defining chance or probabilistic constraints.

2 Industry - Manufacturing Problem

In the first problem, we have an industrial firm, referred to as F, which engages in the manufacturing of $n = 8$ products. The production process requires the acquisition of various

parts or sub-assemblies from external suppliers, specifically from $m = 5$ different sites. To illustrate this process, we have a transportation plan that visualizes the movement of goods between F and 5 suppliers, as well as between F and 8 production locations (warehouses) where the products are manufactured. In this problem, a unit of product i requires a_{ij} units of part

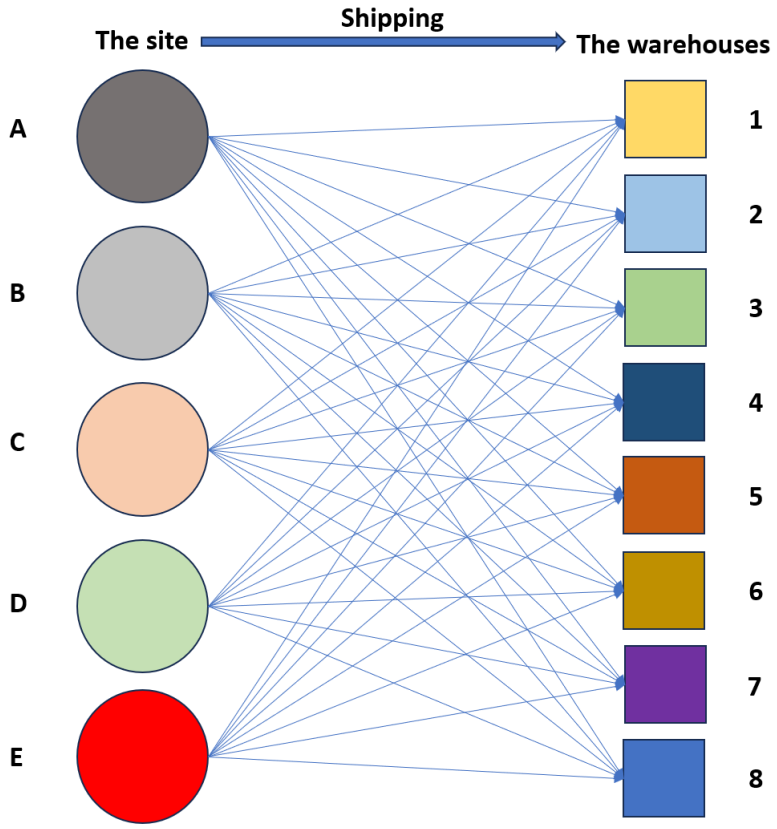

Figure 1: *Transportation plan of Firm F*

j, where i varies from 1 to 8 and j varies from 1 to 5. The demand for the products is unknown and modeled as a random vector $\omega = D = (D_1, D_2, \ldots, D_8)$. The request is to minimize the cost of the manufacturing process. In order to solve this problem, we need to consider every constraint and variable to build up a SLP (Stochastic Linear Programming) model and use a suitable software to solve the said model.

2.1 The second-stage model

The second-stage model deals with determining the optimal production plan when an observed value (a realization of) $d = (d_1, d_2, \ldots, d_n)$ of the above random demand vector D is known. The decision variables $z = (z_1, z_2, \ldots, z_n)$ represent the quantity produced of each product, and $y = (y_1, y_2, \ldots, y_m)$ represent the excessive parts from each supplier. The goal is to minimize production costs. Constraints ensure that inventory is sufficient to meet demand and that production levels do not exceed demand. We declare some more necessary vectors for this problem:

- $s \in \mathbb{R}^m = (s_1, s_2, ..., s_n)$ representing the selling price of leftover parts in the inventory.

- $A \in \mathbb{R}^{m \times n} = [a_{ij}]$ is the relationship matrix between the number of products created and the number of parts each product requires. Specifically, a_{ij} denotes the number of parts that z_i needs.
- $l \in \mathbb{R}^n$, $l = (l_1, l_2, ..., l_n)$ denotes the production cost of the product z_i.
- $q \in \mathbb{R}^n$, $q = (q_1, q_2, ..., q_n)$ serves as another vector, providing the selling price for products. Each q_i designates the selling price of product z_i.
- $c \in \mathbb{R}^n$, $c = (c_1, c_2, ..., c_n)$ representing the cost coefficients associated with each product.

We formulate our problem:

$$\min_{(z,y)} Z = c^T z - s^T y \quad (7)$$

Such that

$$y = x - A^T z$$
$$0 \leq z \leq d$$
$$y \geq 0$$

The second-stage model can be expressed in more detail, knowing that there are $n = 8$ products and the number of parts to be ordered before production $m = 5$, we obtain the followings:

- $\mathbf{s} = (s_1, s_2, \ldots, s_8)$
- $\mathbf{l} = (l_1, l_2, \ldots, l_8)$
- $\mathbf{q} = (q_1, q_2, \ldots, q_8)$
- $\mathbf{c} = (c_i l_i - q_i) = (c_1, c_2, \ldots, c_8)$

$$\min_{(z,y)} Z = \sum_{i=1}^{8} (l_i - q_i) z_i - \sum_{j=1}^{5} s_j y_j$$

Such that

$$\begin{cases} y_j = x_j - \sum_{i=1}^{8} a_{ij} z_i \\ 0 \leq z_i \leq D_i \\ y_j \geq 0 \end{cases}$$

for $i = 1, \ldots, 8$ and $j = 1, \ldots, 5$.

2.2 The first-stage model

In this first-stage section, we will only consider the case production is always less or equal to demand. The first stage is the initial phase of decision-making in a stochastic programming problem. The decision variable

$$x \in \mathbb{R}^m, \quad x = (x_1, x_2, \ldots, x_m)$$

representing the number of parts to be ordered before production. Some additional vectors have to be declared:

$$b \in \mathbb{R}^m, \quad b = (b_1, b_2, \ldots, b_m)$$

acts as the pre-order cost of each part

$$p \in \mathbb{R}^n, \quad p = (p_1, p_2, \ldots, p_n)$$

is a vector of the probability of each scenario.
We formulate our model:

$$\min g(x, z) = b^T x + Q(x) \quad (8)$$

The first part of the objective function captures the cost associated with making initial orders and decisions represented by the variable x. In contrast, the second part $Q(x)$ accounts for the expected cost of implementing the optimal production plan (as determined by Equation 7) while considering the updated order quantities x. This expected cost calculation takes into consideration the randomness in the demand and its associated probability distributions. Therefore, we can formulate our model more detailed:

$$\min g(x, z) = b^T x + \sum_{j=1}^{5} p_j c_j z_j$$

We can turn this into its deterministic equivalent form.

2.3 Utilizing Python for Problem Solving

After having constructed the 2-SLPWR model above, we utilize suitable software to solve. Specifically, we use the Pyomo library to solve this 2-SLPWR model. First, we build up a numerical model for the problem, where $n = 8$, $m = 5$, and the number of scenarios $S = 2$ with density $p_s = \frac{1}{2}$.
Constants:

- $n = 8$ – the number of products.
- $m = 5$ – the number of parts to be ordered before production.
- $S = 2$ – the number of scenarios .
- $p_s = \frac{1}{2}$ – density of each scenario.

Randomly generated values:

- Vector $b = (b_1, b_2, \ldots, b_5)$ – pre-ordered cost of each part.
- Vector $l = (l_1, l_2, \ldots, l_8)$ – the production cost of the product.
- Vector $q = (q_1, q_2, \ldots, q_8)$ – the selling price of the product.
- Vector $s = (s_1, s_2, \ldots, s_5)$ – the salvage value of leftover parts.
- Matrix $A = [a_{ij}]$ – the matrix of requirements.
- Vector D – a random demand vector follows the binomial distribution $\text{Bin}(10, \frac{1}{2})$.

Decision variables:

- $x = (x_1, x_2, \ldots, x_5)$ – the number of parts to be ordered before production.

- $z_1 = (z_{11}, z_{12}, \ldots, z_{18})$ – the number of units produced in the first scenario.
- $z_2 = (z_{21}, z_{22}, \ldots, z_{28})$ – the number of units produced in the second scenario.
- $y_1 = (y_{11}, y_{12}, \ldots, y_{15})$ – the number of excessive parts of each supplier in the first. scenario.
- $y_2 = (y_{21}, y_{22}, \ldots, y_{25})$ – the number of excessive parts of each supplier in the second scenario.

Objective function:

Stage 1:

$$\min g(x, z) = b^T x + \sum_{j=1}^{5} p_j c_j z_j$$

Stage 2:

$$\min_{z,y} Z = c^T z - s^T y = \sum_{i=1}^{8} (l_i - q_i) z_i - \sum_{j=1}^{5} s_j y_j$$

Constraints:

$$\begin{aligned}
&0 \leq z_{1i} \leq d_{1i} \quad \text{for } i = 1, \ldots, 8 \\
&0 \leq z_{2i} \leq d_{2i} \quad \text{for } i = 1, \ldots, 8 \\
&y_{1j} = x_j - \sum_{i=1}^{8} a_{ij} z_{1i} \quad \text{for } j = 1, \ldots, 5 \\
&y_{2j} = x_j - \sum_{i=1}^{8} a_{ij} z_{2i} \quad \text{for } j = 1, \ldots, 5 \\
&x, y_1, y_2, z_1, z_2 \geq 0
\end{aligned}$$

2.3.1 Implementation of Problem 1

We implement the code for the numerical model using Python (the code has been written in a separate file).

First of all, we randomly generate all of our data vectors and matrix using a uniform random distribution with the exception of the random demand vector D using a binomial distribution:

```
Preorder cost b:
[25.48813504 27.15189366 26.02763376 25.44883183 24.23654799]
Production cost l:
[164.58941131 143.75872113 189.17730008 196.36627605 138.34415188
 179.17250381 152.88949198 156.80445611]
Selling price q:
[962.79831915 535.5180291  543.56464985 510.10919872 916.30992277
 889.07837547 935.00607412 989.30917112]
Salvage value s:
[13.99579282 12.30739681 13.90264588 10.59137213 13.19960511]
Requirement matrix A:
[[1.43353287 9.44668917 5.21848322 4.1466194  2.64555612]
 [7.74233689 4.56150332 5.68433949 0.187898   6.17635497]
 [6.12095723 6.16933997 9.43748079 6.81820299 3.59507901]
 [4.37031954 6.97631196 0.60225472 6.66766715 6.7063787 ]
 [2.10382561 1.28926298 3.15428351 3.63710771 5.7019677 ]
 [4.38601513 9.88373838 1.02044811 2.08876756 1.61309518]
 [6.53108325 2.53291603 4.66310773 2.44425592 1.58969584]
 [1.10375141 6.56329589 1.38182951 1.96582362 3.68725171]]
Demand Vector D:
[[6 3 7 3 8 5 8 5]
 [6 2 4 3 4 3 4 5]]
```

Figure 2: *Randomly generated values*

Figure 2: Randomly generated values

Then, we calculate the optimal solution as well as the value for the 2-SLPWR model:

```
Optimal Solution for the First Stage:
x: [105.129300899013, 169.492738414771, 105.861417312276, 93.8035813237139, 100.708379478648]
z in the First Scenario: [6.0, 0.0, 0.0, 0.0, 8.0, 5.0, 8.0, 5.0]
z in the Second Scenario: [6.0, 2.0, 2.21260529860633, 1.76065403792022, 4.0, 3.0, 4.0, 5.0]
Cost of the First Stage: -7346.97410468548

Optimal Solution for the Second Stage in the First Scenario:
y in the First Scenario: [0.0, 2.55795384873636e-13, 5.43565192856477e-13, 4.08562073062058e-14, 5.6843418860808e-13]
z in the First Scenario: [6.0, -2.55819128379529e-14, 0.0, 0.0, 8.0, 5.0, 8.0, 5.0]
Cost of the Second Stage in the First Scenario: -24981.965204722317

Optimal Solution for the Second Stage in the Second Scenario:
y in the Second Scenario: [27.8269919421314, 25.9331861246391, 21.9417821888076, 28.1271936309386, 20.0401345787188]
z in the Second Scenario: [6.0, 2.0, 0.0, 0.0, 4.0, 3.0, 4.0, 5.0]
Cost of the Second Stage in the Second Scenario: -19681.449756649527
```

Figure 3: *Optimal values and solution*

Figure 3: Optimal values and solution

Based on the above data, we can conclude the optimal solution $x, y \in \mathbb{R}^m$, and $z \in \mathbb{R}^n$ requested by the problem.

```
Optimal Solution:
x: [105.129300899013, 169.492738414771, 105.861417312276, 93.8035813237139, 100.708379478648]
y in the First Scenario: [0.0, 2.55795384873636e-13, 5.43565192856477e-13, 4.08562073062058e-14, 5.6843418860808e-13]
y in the Second Scenario: [27.8269919421314, 25.9331861246391, 21.9417821888076, 28.1271936309386, 20.0401345787188]
z in the First Scenario: [6.0, -2.55819128379529e-14, 0.0, 0.0, 8.0, 5.0, 8.0, 5.0]
z in the Second Scenario: [6.0, 2.0, 0.0, 0.0, 4.0, 3.0, 4.0, 5.0]
```

Figure 4: *Optimal values and solution*

Figure 4: Optimal solution $x, y \in \mathbb{R}^m$, and $z \in \mathbb{R}^n$

2.4 Code Analysis

The implemented code showcases a stochastic programming approach to solve the Industry – Manufacturing Problem. The code starts by importing the necessary libraries, including Pyomo for modeling and optimization and NumPy for generating random data. Then, the code defines several constants such as the number of scenarios (S), the density of each scenario (p_s), the number of products (n), and the number of parts to be ordered before production (m). A function `get_data()` is defined to generate random data for the problem, including pre-ordered costs of each part (`vector_b`), production costs of the product (`vector_l`), selling price of the product (`vector_q`), salvage value of leftover parts (`vector_s`), requirement matrix (`matrix_A`), and demand vector (`vector_D`). The code prints the randomly generated data to provide an overview of the problem instance.

The code initializes a Pyomo `ConcreteModel` object `model` to represent the first-stage problem. Decision variables `x` and `z` are defined using the `Var()` function. The objective function is created as an expression (`Obj_function`) and added to the model as the objective to minimize. The model constraints are added using the `ConstraintList()` object. The code uses the GLPK solver to solve the first-stage problem by calling `solver.solve(model)`. The optimal solution values for decision variables `x` and `z` are extracted from the model. The code prints the optimal solution values for `x` and `z` variables. By solving the first stage, optimal production and pre-order decisions are obtained. The code then proceeds to solve separate second-stage problems for each scenario.

Similar to the previous step, the code initializes new object models to represent the second-stage problem in the first and second scenarios. These problems optimize production and salvage decisions based on observed demand in each scenario. The objective is to minimize the difference between production costs and selling prices, while considering salvage values of leftover parts. The code solves the second-stage problem for both scenarios. Then, the optimal solution values for decision variables `y` and `z` are extracted from the models. Finally, the code prints the optimal solution values for `y` and `z` variables in both scenarios.

3 Problem 2

3.1 Min-cost Flow Problem

The minimum cost flow problem is a classic optimization problem in network flow theory. It involves finding the cheapest way to send a certain amount of flow through a directed graph from a source node to a sink node, while respecting capacity constraints on the edges.

Formally, let $G = (V, E)$ be a directed graph with a source node s, a sink node t, and a set of edges E. Each edge $(u, v) \in E$ has a capacity $c(u, v)$ and a cost per unit of flow $f(u, v)$. The goal is to find the minimum cost flow that sends a specified amount of flow from the source node to the sink node, subject to the capacity constraints.

Mathematically, the minimum cost flow problem can be defined as follows:

$$\begin{cases} \text{Minimize} & \sum_{(u,v)\in E} f(u,v)\cdot c(u,v) \\ \text{Subject to:} & \\ & \text{Flow conservation constraint: } \sum_{(u,v)\in E} f(u,v) - \sum_{(v,u)\in E} f(v,u) = 0 \\ & \text{for all nodes } v \text{ except the source and sink nodes.} \\ & \text{Capacity constraint:} 0 \le f(u,v) \le c(u,v) \text{for all edges } (u,v)\in E. \\ & \text{Flow requirement constraint: } \sum_{(s,v)\in E} f(s,v) = D, \text{where } D \text{ is the desired amount of flow} \\ & \text{from the source node } s \text{ to the sink node } t. \end{cases}$$

There are several algorithms to solve the minimum cost flow problem, such as the successive shortest path algorithm, the cycle-canceling algorithm, and the cost scaling algorithm. These algorithms iteratively find augmenting paths in the residual graph to improve the flow and reduce the cost until an optimal solution is reached.

It's worth mentioning that the minimum cost flow problem is a generalization of other flow problems, such as the maximum flow problem and the minimum cost circulation problem. It has various applications in transportation, logistics, network design, and resource allocation, among others. The objective function of the minimum cost flow problem is to minimize the total cost of the flow. It can be represented as:

$$\text{Minimize} \sum_{(u,v)\in E} f(u,v)\cdot c(u,v),$$

where $f(u,v)$ represents the flow on edge (u,v), and $c(u,v)$ represents the cost per unit of flow on edge (u,v). The objective is to minimize the sum of the costs for all edges.

Subject to:

$$\begin{cases} \sum_{(u,v)\in E} f(u,v) - \sum_{(v,u)\in E} f(v,u) = 0 & \text{for all nodes } v \text{ except the source and sink nodes.} \\ 0 \le f(u,v) \le c(u,v) & \text{for all edges } (u,v)\in E. \\ \sum_{(s,v)\in E} f(s,v) = D. \end{cases}$$

3.2 Time-dependent Min-cost Flow Problem

With the growing interest in dynamically managing transportation systems, there is a need to identify the shortest paths in scenarios where the weights or delays associated with arcs change dynamically over time. Real-time traffic information, along with well-known standard traffic patterns, enables the enhancement of user services, such as providing optimal travel routes considering rush hour congestion, for instance, "How to travel from one city to another city as fast as possible." Time-dependent min-cost flow problems are commonly encountered in real-world applications such as traffic management, logistics, and supply chain optimization.

The Time-dependent Minimum Cost Flow Problem is a variation of the traditional Minimum Cost Flow Problem in network optimization. The goal is to determine the optimal flow of goods or resources through a network from a source to a sink, minimizing the total cost incurred. In this variant, the capacities and costs associated with arcs in the network are time-dependent. This means that these parameters change over time, introducing a temporal dimension to the problem.

Let $y_{ij}(t)$ be a positive integer decision variable representing the number of people starting from node i at time t and reaching node j. The capacity of edge $(i,j) \in E$ at time t is denoted as $u_{ij}(t)$, imposing the capacity constraint $0 \leq y_{ij}(t) \leq u_{ij}(t)$. The flow on node i at time t, denoted as $d_i(t)$, equals the difference between the number of people starting from node i at time t and those arriving at node i at the same time. Mathematically, this can be expressed as: $d_i(t) =$ (number of people starting from node i at time t) $-$ (number of people arriving at node i at time t).

The second sub problem within the relaxed model (13) is associated with the decision variables denoted as Y, and the optimal objective value for this problem is represented as $Z^*_{SP2}(\alpha)$, shown as follows:

$$\begin{cases} \min\limits_{P} \sum\limits_{S=1}^{S} \left(\mu_S \cdot \sum\limits_{(i,j)\in A_S} C^s_{ij}(t) \cdot y^s_{ij}(t) \right) \\ \text{s.t.} \\ \quad \sum\limits_{(i_t, j_{t'})\in A_S} y^s_{ij}(t) - \sum\limits_{(j_{t'}, i_t)\in A_S} y^s_{ij}(t') = d^s_i(t), \forall i \in V, \forall t \in \{0,1,\ldots,T\}, \forall s = 1,2,\ldots,S \\ \quad 0 \leq y^s_{ij}(t) \leq u^s_{ij}(t), \forall (i,j) \in A, \forall t \in \{0,1,\ldots,T\}, \forall s = 1,2,\ldots,S \end{cases}$$

If remaining within a node between intervals is not permitted, the variable $d_i(t)$ will be zero for every time t and node i, excluding the source and sink. The objective function is represented by z, which is the summation over all arcs $(i,j) \in E$ of the product $c_{ij}(t) \cdot y_{ij}(t)$, where $c_{ij}(t)$ is the cost associated with traversing link (i,j) at time t.

3.3 Successive Shortest Path Problem

The Successive Shortest Path algorithm is a well-known optimization technique used in network optimization to solve the minimum-cost flow problem. Its objective is to determine the most cost-effective way to transport goods or resources from a source to a sink within a network while adhering to capacity constraints and demand requirements.

The algorithm operates by iteratively identifying the shortest path in the residual graph, representing the remaining capacity of edges in the network. It begins with an initial flow of zero and gradually increases the flow along these paths to enhance the overall solution.

Given a network G consisting of n vertices and m edges, the capacity (a non-negative integer) and the cost per unit of flow along each edge (some integer) are given. Also, the source S and the sink T are marked.

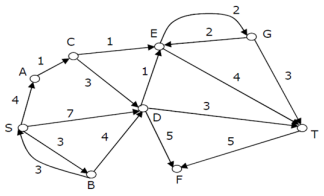

Figure 6: Shortest path from A -> T

For a given value K, we have to find a flow of this quantity, and among all flows of this quantity, we have to choose the flow with the lowest cost. This task is called the minimum-cost flow problem.

For example, in the illustration, the shortest path from node A to node T is $A \to C \to E \to T$ with the minimum cost 6.

3.3.1 Successive Shortest Path Algorithm for Solving Min-Cost Flow Problem

Let's consider a min-cost flow problem model:

$$\begin{cases} \min_P \sum_{(i,j)\in E} c_{ij} \cdot x_{ij} \\ \text{s.t.} \\ 0 \le x_{ij} \le u_{ij} \quad \forall (i,j) \in E \\ \sum_{(i,j)\in E} x_{ij} - \sum_{(j,i)\in E} x_{ji} = d_i \quad \forall i \in V \end{cases}$$

With the condition:

$$d_i = \begin{cases} v, & \text{if } i = s \text{ (source)} \\ -v, & \text{if } i = t \text{ (sink)} \\ 0, & \text{otherwise} \end{cases}$$

The min-cost flow problem is a more advanced version of the shortest path problem. In this problem, we are required to find a path from a source node to a sink node. However, there are some key differences. In the min-cost flow problem, the number of people that need to be transferred is more than one, and each link now has a maximum capacity. This means that we can't simply solve the problem by allowing all people to go through the shortest path.
In the min-cost flow problem, the shortest path is defined as the path with the lowest cost for transfer. The main idea is to maximize the number of people that can be transferred along the shortest path. It's important to note that flow transportation is parallel, so we need to update the available capacity of the edges that are part of the shortest path. Then, we continue finding the shortest path in the remaining graph.
The algorithm will terminate under two conditions: when the number of people being transferred reaches the total number of people (denoted as v), or when there are no paths left with positive capacity. The termination condition is based on our understanding and the best explanation found in the research on the represented algorithm.
To summarize, the min-cost flow problem extends the concept of the shortest path problem by considering multiple people to be transferred and maximum capacities on each link. The algorithm aims to maximize the number of people transferred along the shortest path, updating capacities and finding new paths until the termination conditions are met.
Algorithm 1: Successive Shortest Path Algorithm for Solving Min-Cost Flow Problem
Step 1: Define $G(V, A(x), U(x), C(x))$ as the residual network.

- V: Set of vertices.
- $A(x) = \{(i,j) \mid ((i,j) \in A) \cap (u_{ij} > x_{ij})\}$: Set of edges where the flow does not reach its capacity.
- $U(x) = \{u_{ij} \mid (i,j) \in A\}$: Set of edge capacities.
- $C(x) = \{c_{ij} \mid (i,j) \in A\}$: Set of edge costs.
- $R(x) = \{u_{ij} - x_{ij} \mid (i,j) \in A\}$: Set of residual capacities of edges. $R(x)$ also represents the ability of an edge to have more flow.

Step 2: Find the shortest path in the graph $G = (V, A(x))$ from the source to the sink. If no path is found, the problem is infeasible.

Step 3: Increase the flow of all edges in the shortest path as much as possible. To do this, choose the edges with the smallest residual capacity, and increase all flows x_{ij} in the shortest path by this value. Note that edges with zero residual capacity are removed from the set $A(x)$.

Step 4: If the flow in the sink node, denoted as x, is less than v, go to Step 2. Otherwise, the problem terminates. Note that in the last iteration, it is necessary to increase the flow such that $x = v$.

3.3.2 Applications

3.3.2.a Mathematics Application

The Successive Shortest Path algorithm belongs to the broader field of mathematical optimization techniques. It utilizes principles from graph theory, linear programming, and network flow theory. The algorithm's mathematical basis involves formulating the problem as a linear program and employing methods for solving linear programming problems, such as the simplex method or interior point methods. The algorithm's correctness and optimal properties are established through mathematical analysis and proofs.

3.3.2.b Coding Application

The Successive Shortest Path algorithm can be implemented in different programming languages to solve network optimization problems. The typical implementation involves representing the network as a graph and utilizing appropriate data structures and algorithms for efficient graph traversal and shortest path calculations, such as Dijkstra's algorithm or the Bellman-Ford algorithm. Programming concepts like arrays, graphs, loops, and conditional statements can be employed. Additionally, there are libraries and frameworks available that offer built-in functions or classes for network flow optimization, simplifying the implementation.

In coding, the Successive Shortest Path algorithm finds applications in various domains, including transportation and logistics systems, supply chain management, network routing, and resource allocation problems. It aids in optimizing resource allocation, minimizing costs, and enhancing efficiency in various real-world scenarios.

3.3.3 Example 1: Optimization of Transportation Network

Consider a transportation network comprising factories, warehouses, and customers. The objective is to minimize the cost of transporting goods from factories to customers while fulfilling demand requirements. Each edge in the network bears a specific cost, representing the transportation cost per unit of flow. By utilizing the Successive Shortest Path algorithm, we can determine the optimal configuration of flow that minimizes the overall transportation cost while ensuring that each customer's demand is satisfied.

3.3.4 Example 2: Maximum Flow with Minimum Cost

In the realm of network flow problems, the Successive Shortest Path algorithm proves useful for solving the maximum flow with minimum cost problem. Here, the aim is to maximize the flow from a source to a sink while minimizing the total cost associated with transporting the flow. The algorithm achieves this goal by iteratively augmenting the flow along the shortest paths in the residual graph, considering both capacity constraints and the minimization of costs.

3.3.5 Benefits and Drawbacks

3.3.5.a Benefits

- **Efficiency:** The SSP algorithm has a relatively efficient time complexity compared to other flow algorithms. It typically performs well on sparse networks and can find the optimal solution efficiently.
- **Flexibility:** The algorithm can handle various network flow problems, including minimum-cost flow, maximum flow, and multi-commodity flow problems. It can accommodate different supply and demand constraints, making it versatile for a wide range of applications.
- **Convergence:** The SSP algorithm is guaranteed to converge to an optimal solution under certain conditions, such as non-negative costs and the absence of negative cycles. It provides a systematic approach to iteratively improve the flow configuration until the optimal solution is reached.
- **Scalability:** The algorithm can handle large-scale networks with a large number of nodes and edges. It is designed to solve flow problems in complex networks efficiently.

3.3.5.b Drawbacks

- **Non-Optimality on Negative Cycles:** The SSP algorithm assumes non-negative costs in the network. If the network contains negative-cost cycles, the algorithm may not converge to an optimal solution or may encounter infinite loops. Preprocessing steps, such as cycle detection and elimination, may be required to handle negative cycles.
- **Sensitivity to Network Structure:** The efficiency of the SSP algorithm can be affected by the network's structure and properties. In some cases, the algorithm may experience slow convergence or encounter degenerate situations, leading to suboptimal solutions. Preprocessing or post-processing techniques may be needed to improve performance.
- **Limited Applicability to Dynamic Networks:** The SSP algorithm is designed for static networks where the network topology and parameters remain constant. It may not perform optimally in dynamic networks with changing edge capacities or costs. Additional techniques, such as dynamic graph algorithms or incremental updates, may be required to handle dynamic scenarios.
- **Dependency on Shortest Path Algorithm:** The efficiency of the SSP algorithm relies on the performance of the chosen shortest path algorithm. The choice of the shortest path algorithm affects the overall runtime and convergence speed. If the shortest path algorithm is not well-optimized, it may impact the efficiency of the SSP algorithm.

In summary, there are both advantages and disadvantages to the Successive Shortest Path (SSP) method that should be taken into account when using it to solve network flow issues. The algorithm offers benefits in terms of scalability, adaptability, convergence, and efficiency. It can effectively manage a variety of flow issues and, in some cases, converges to the best answer. It is also capable of efficiently managing large-scale networks.

However, there are drawbacks as well. If the network has negative-cost cycles, the SSP method might not yield the best results. The network's structure may impact its performance, and for better results, pre- or post-processing methods might be needed. The approach is dependent on

the effectiveness of the selected shortest path algorithm and has limited application to dynamic networks with changing characteristics.

In the end, the needs of the particular problem, the properties of the network, and the trade-offs between optimality and efficiency should be taken into account while choosing the SSP algorithm. It's critical to evaluate whether the algorithm's advantages in a particular situation outweigh its disadvantages, and if not, consider alternative algorithms.

3.3.6 Implementation of problem 2

The code has been written in a separate file.

3.4 Code Analysis

The provided Python code implements the Successive Shortest Paths Min-Cost Max-Flow algorithm to determine the minimum-cost flow in a directed graph. The code defines a graph using the `Edge` and `Vertex` classes, representing directed edges and vertices, respectively. The `Edge` class encapsulates crucial attributes such as source, destination, flow, capacity, and cost, while the `Vertex` class keeps track of connected arcs, forming the underlying structure of the graph.

The code begins by constructing a graph, specifying vertices, and associating them with edges and their respective attributes using the `addEdge()` method. Subsequently, the user can designate a source and sink vertex. In this specific example, a graph with 7 vertices is created, and various edges are added with their respective attributes. The goal is to find the shortest paths with maximum flow using the `min_costflow()` method.

The Successive Shortest Paths Min-Cost Max-Flow algorithm consists of two main steps: potential calculation and Dijkstra's algorithm. Initially, the Bellman-Ford algorithm is applied to calculate potentials, ensuring non-negative edge costs. Following that, Dijkstra's algorithm is employed to find the shortest paths in the modified graph. The algorithm iteratively augments the flow along the shortest paths until no more augmenting paths are available. The priority queue `frontier[]` is implemented as a list and sorted in descending order, ensuring efficient selection of the next vertex with the minimum potential (cumulative cost) during Dijkstra's algorithm execution.

The `printPathDetails()` method is used for printing the details of a given path. These details are printed in reverse order to illustrate the path from source to sink and their associated cost and flow values. Finally, when no path can be found, the algorithm terminates, the method returns the result, representing the maximum flow value from the source to the sink.

4 Conclusion

Under the expert guidance of Mr.Khuong and Mr.Man during our classes and forums, we've adeptly navigated through the intricate realm of mathematical modeling, with a specific focus on the challenging domain of stochastic programming. This note serves as an expression of our deep gratitude for the insightful assignment that delved into the nuances of this complex field.

The assignment, centered around stochastic programming, has been instrumental in broadening our understanding of mathematical modeling and its practical applications. The thought-provoking nature of the tasks assigned has not only strengthened our grasp on stochastic programming but has also played a pivotal role in the development of critical skills crucial for

our academic and professional journey.

The exposure to the complexities of stochastic programming has proven invaluable, providing us with a profound comprehension of advanced mathematical modeling techniques. Your commitment to presenting stimulating assignments has been a catalyst for our academic and professional growth. We genuinely appreciate the effort you invest in guiding us through challenging yet rewarding tasks. Once again, thank you for this enriching and enlightening experience!

4.1 References

References

[1] School of Modern Post, Beijing University of Posts and Telecommunications, China, *Computers & Industrial Engineering*, chapter: A two-stage stochastic programming framework for evacuation planning in disaster responses, 2020.

[2] Donald E. Knuth. *The TEX Book*. Addison-Wesley Professional, 1986.

[3] Leslie Lamport. *LATEX: a Document Preparation System*. Addison Wesley, Massachusetts, 2nd edition, 1994.

[4] Frank Mittelbach, Michel Gossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LATEX Companion*. Addison-Wesley Professional, 2nd edition, 2004.

[5] Helmut Kopka, Patrick W. Daly. *A Guide to LATEX and Electronic Publishing*. Addison-Wesley Long Man Limited, 4th edition, 2004.