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

MATHEMATICAL MODELING (CO2011)

Assignment (Semester: 231, Duration: 06 weeks)

"Stochastic Programming and Applications"

Advisor: Nguyễn Văn Minh Mẫn, Mahidol University
Nguyễn An Khương, CSE - HCMUT
Mai Xuân Toàn, CSE - HCMUT
Trần Hồng Tài, CSE - HCMUT
Nguyễn Tiến Thịnh, CSE - HCMUT

Students: Nguyễn Văn Thành Đạt - 2152055.
Nguyễn Khánh Nam - 2153599.
Lê Văn Phúc - 2152241.
Nguyễn Quang Thiện - 2152994.
Trần Minh Tuấn - 2152336.

HO CHI MINH CITY, DECEMBER 2023

Contents

1 Member list & Workload

No.	Fullname	Student ID	Problems	Percentage of work
1	Nguyễn Văn Thành Đạt	2152055	Problem 1	20%
2	Nguyễn Khánh Nam	2153599	Problem 2	20%
3	Lê Văn Phúc	2152241	Problem 2	20%
4	Nguyễn Quang Thiện	2152994	Problem 1	20%
5	Trần Minh Tuấn	2152336	Problem 2	20%

2 Introduction to Two-Stage Stochastic Linear Programming

Two-Stage Stochastic Linear Programming (SLP) is a mathematical modeling and optimization framework that addresses decision-making problems under uncertainty. When faced with uncertain future scenarios, it is a potent instrument used in management, operations research, and decision sciences to make strategic judgments.

With two-stage SLP, uncertainty is incorporated into the decision-making process, in contrast to typical linear programming, where all parameters are assumed to be known exactly. This acknowledges that a range of uncontrolled factors may affect key criteria, such demand, costs, or resource availability.

The decision-making process happens in two separate phases or time periods, which is referred to as the "two-stage" feature of the model. The **first stage** entails choices made in the face of ambiguity; they are commonly known as "first-stage decisions." In the **second stage**, choices are determined based on actual realization of unknown parameters; these are referred to as *"second-stage decisions"* or *"recourse decisions"*. First-stage decision variables are often specified prior to uncertainty resolution in a two-stage SLP model, and second-stage decision variables are updated depending on observed realization of uncertain parameters.

The goal of two-stage SLP is to **reduce expenses** or **increase earnings** over both phases in a way that is weighted. Taking into account every situation, this objective function includes the costs of first-stage decisions as well as the anticipated costs or benefits of second-stage decisions.

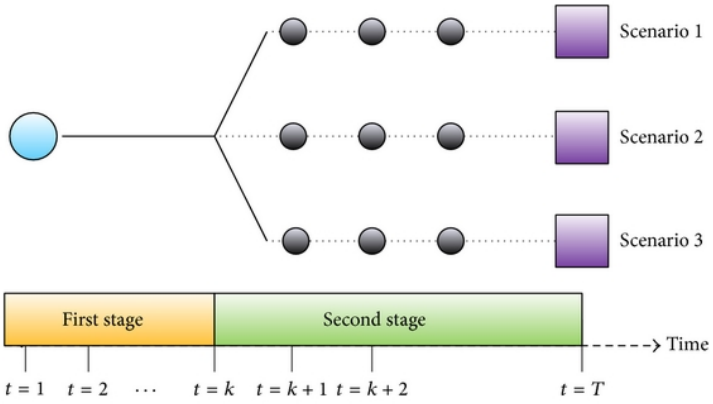


Figure 1: Two-stage Stochastic Linear Programming

All in all, advanced optimization approaches, such as stochastic programming algorithms, recourse methods that dynamically modify decisions depending on observed results, and linear programming solvers, are usually required to solve a two-stage SLP model. Hence, Two-Stage Stochastic Linear Programming offers a strong framework for making decisions when faced with uncertainty, enabling businesses to decide wisely and adapt to a range of contingencies. It is an effective tool for resource allocation and strategic planning because it combines the mathe-

matical precision of linear programming with the adaptability to change course in reaction to unanticipated events.

3 Tools and environment

3.1 Google Colaboratory

Colaboratory, or "Colab" for short, is a product from Google Research. Colab assists users in writing and executing arbitrary Python code through the browser, and is eminently suited to machine learning, data analysis and education. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, whereas offering access free of charge to computing resources including Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), enabling users to accelerate computations without the need for dedicated hardware.

Moreover, Colab acquires some pros which may greatly assist us in the process of dealing with the assigned problems:

- **Pre-installed Libraries:** To make data analysis, machine learning, and scientific computing activities easier, Colab comes pre-installed with popular Python libraries including NumPy, Pandas, TensorFlow, and Matplotlib.

- **Google Drive integration:** Colab and Google Drive have been connected so that users may view, save, and share their Colab notebooks straight from their Google Drive accounts.

- **Real-Time Collaboration:** Colab notebooks enable several users to work together in real-time, promoting cooperation and enabling adjustments and comments to be made in real time. It is such a great tool for group projects and learning environments since it has collaboration capabilities that are comparable to those of Google Docs.

Therefore, we have decided to use this supportive environment so as to associate, perform live code and run it altogether.

Figure 2: Google Colaboratory

3.2 Python

Python is a helpful and straightforward programming language that allows developers to write programs with fewer lines than some other programming languages with its simple syntax and can be treated in a procedural way, an object-oriented way or a functional way. Python runs on an interpreter system, meaning that code can be executed as soon as it is written. On top of that, this kind of language also comes with a mixed diversity of tools for developing models and analyzing data. The tools are referred known as "packages" which pave the way for us to analyze, visualize and manipulate sets of data in this project.

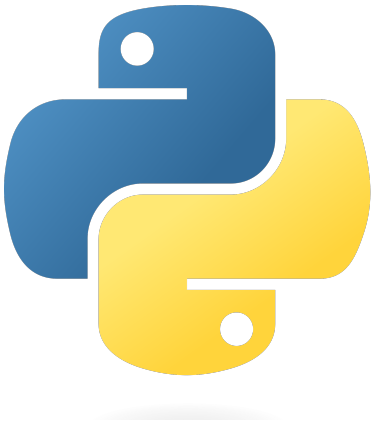


Figure 3: Python programming language

3.2.1 Gurobi and GurobiPy

Gurobi, a top optimization solver for challenging mathematical programming problems, has a Python interface called GurobiPy. GurobiPy is a user-friendly and adaptable tool for creating, refining, and evaluating a variety of mathematical models. It enables users to effortlessly incorporate the capabilities of Gurobi into Python contexts.

In the project, we opt for employing Gurobi to assist us in Problem 1 with Model Formulation, Solver, Optimization, Result Analysis, Iterative Refinement, and also based on several useful characteristics:

- **Excellence in Optimization:** With its cutting-edge optimization methods, Gurobi is well-known for being a top option for solving challenging mathematical programming issues. Gurobi's optimization skills may be easily included into Python programs with GurobiPy, providing Gurobi's functionality to a wider range of users.

- **Comfort in Utilization:** The task of creating optimization models in Python is made easier by GurobiPy's user-friendly interface, which offers an expressive and straightforward modeling language. The syntax of Python improves readability and makes it easier to convert mathematical models into executable code.

- **High Performance:** Gurobi is engineered for maximum efficiency and can effectively tackle complex, large-scale optimization challenges. GurobiPy ensures that optimization models are performed efficiently and rapidly by extending this performance to Python users.

- **Adaptability:** Gurobi is compatible with several optimization techniques, such as mixed-integer programming (MIP), quadratic programming (QP), and linear programming (LP). GurobiPy carries on this adaptability, enabling users to tackle optimization challenges across a wide range of industries, from manufacturing and operations research to finance and logistics.

- **Availability of Enhanced Functionalities:** Access to Gurobi's advanced features and capabilities, including warm starts, custom constraints, and solution pool management, is made possible via GurobiPy. GurobiPy allows users to use Python programs to fully use the optimization engine of Gurobi.

Figure 4: Gurobi optimization

3.2.2 NetworkX

Python network analysis is made easy with NetworkX, a flexible and intuitive framework. Because of its extensive tool and algorithm set, it is a priceless instrument for investigating and comprehending the complicated dynamics and structures of complex networks in a variety of domains, especially in Problem 2. The key features of NetworkX such as Graph Data Structures, Graph Algorithms, Graph Generation and Analysis, Graph Drawing and Visualization, and Dynamics and Evolution would play a vital role for us to figure out the root of the assigned problem.

Figure 5: NetworkX

3.2.3 NumPy

NumPy, standing for Numerical Python, is a Python library used for working with mathematical problems, such as linear algebra, fourier transform, multi-dimensional arrays and matrices. This tool would pave the way for us to facilitate an abundance of issues related to maths throughout the time we build the models. We may also consider some of its primary features:

- **N-dimensional Arrays:** The n-dimensional array is the core data structure, which enables us to conduct actions on complete arrays simultaneously based on these arrays' efficiency in storing and handling enormous datasets of numerical values.
- **Mathematical Functions:** NumPy includes a rich set of mathematical functions for performing operations on arrays without the need for explicit looping. This allows for concise and efficient code when working with numerical data.
- **Linear Algebra Operations:** NumPy includes a comprehensive set of functions for linear algebra operations, such as matrix multiplication, eigenvalue decomposition, and singular value decomposition.

Figure 6: NumPy

3.2.4 Pandas

Pandas is built on top of the Python programming language that is widely renowned as a fast, powerful, flexible and easy to use open source data analysis and manipulation tool. It is a go-to tool for activities ranging from cleaning and examining data to preparing it for analysis and machine learning applications because of its easy data structures and broad capability.

What is more, we can utilize its fundamental characteristic which is integration with NumPy and Matplotlib inside Problem 2. Pandas easily works with Matplotlib for data display and NumPy for effective numerical computations. Thanks to this connection, it helps us to analyze and explore data in the most effective way.

Figure 7: Pandas

3.2.5 Matplotlib

Matplotlib is a comprehensive library for serving as the creation of static, animated, and interactive visualizations in Python. In order to make a graphic that clearly illustrates the ideal cost and the average value in both problems, especially Problem 1, we did utilize this useful library, specifically the submodule matplotlib.pyplot, focused on providing a convenient interface for creating plots. Also, Pyplot follows a declarative syntax that allows users to create plots by specifying the desired visual elements and properties, supports a wide range of plot types, including line plots, scatter plots, bar plots, histograms, pie charts, and Users may generate, edit, and view data right within the notebook environment thanks to its seamless integration with Jupyter Notebooks.

Figure 8: Matplotlib

4 QUESTIONS for ASSIGNMENT 2023

4.1 PROBLEM 1

To PROBLEM 1 [produce n products satisfying **production** $\geq$ **demand**]. (4 points)

Use the 2-SLPWR model given in Equations **7** and **8** when $\mathbf{n = 8}$ products, the number of scenarios $\mathbf{S = 2}$ with density $\boldsymbol{p_s = 1/2}$, the number of parts to be ordered before production $\mathbf{m = 5}$, we randomly simulate data vector **b, l, q, s** and matrix **A** of size $\mathbf{n \times m}$.

We also assume that the random demand vector $\boldsymbol{\omega} = \mathbf{D} = (D_1, D_2, ..., D_n)$ where each $\boldsymbol{\omega_i}$ with density $\boldsymbol{p_i}$ follows the binomial distribution **Bin(10, 1/2)**.

REQUEST: build up the numerical models of Equations **7** and **8** with simulated data. Find the optimal solution $\boldsymbol{x, y} \in \mathbb{R}^{\boldsymbol{m}}, \boldsymbol{and z} \in \mathbb{R}^{\boldsymbol{n}}$ by suitable soft (as GAMSPy).

4.1.1 Theoretical basis and variables definition

4.1.1.a The Second-Stage Problem

$\boldsymbol{d} = (d_1, d_2, \cdots, d_8)$ of the above random demand vector D.

Finding the best production plan by solving the following stochastic linear program (SLP):

$\boldsymbol{z} = (z_1, z_2, \cdots, z_8)$: number of units produced.

$\boldsymbol{y} = (y_1, y_2, \cdots, y_5)$: number of parts left in inventory.

$\boldsymbol{l} = (l_1, l_2, \cdots, y_8)$: extra cost to complete each product.

$\boldsymbol{q} = (q_1, q_2, \cdots, q_8)$: selling cost each product.

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

where $s_j < b_j$ (defined as pre-order cost per unit of part j), and $x_j, j = 1, \ldots, 5$ are the numbers of parts to be ordered before production.

$$\text{subject to} \begin{cases} y_j = x_j - \sum\limits_{i=1}^{8} a_{ij} z_i, \quad j = 1, \ldots, 5 \\ 0 \leq z_i \leq d_i, \quad i = 1, \ldots, 8; \quad y_j \geq 0, j = 1, \ldots, 5. \\ (\boldsymbol{a_{ij}} \times z_i\text{: Each } z_i \text{ i-th product needs } a_{ij} \text{ i-th part.}) \\ (\boldsymbol{x_j}\text{: part type j.}) \end{cases}$$

$$\Longrightarrow MODEL = \begin{cases} \min\limits_{\boldsymbol{z},\boldsymbol{y}} Z = \boldsymbol{c}^T \cdot \boldsymbol{z} - \boldsymbol{s}^T \cdot \boldsymbol{y} \\ \text{with } \boldsymbol{c} = (c_i := l_i - q_i) \text{ are cost coefficients} \\ \boldsymbol{y} = \boldsymbol{x} - A^T \boldsymbol{z}, \quad \text{where } A = [a_{ij}] \text{ is matrix of dimension } 8 \times 5, \\ 0 \leq \boldsymbol{z} \leq \boldsymbol{d}, \quad \boldsymbol{y} \geq 0 \end{cases}$$

The vectors z, y depend on realization d of the random demand $\boldsymbol{\omega} = \boldsymbol{D}$ as well as on the 1st-stage decision $x = (x_1, x_2, \cdots, x_5)$.

4.1.1.b The First-Stage Problem (Production $\geq$ Demand)

$Q(\boldsymbol{x}) := \mathrm{E}[Z(\boldsymbol{z}, \boldsymbol{y})] = \mathbf{E}_{\boldsymbol{\omega}}[\boldsymbol{x}, \boldsymbol{\omega}]$: expected value.

$\boldsymbol{x} = (x_1, x_2, \cdots, x_5)$: number of part types that need to be ordered to assemble products.

$\boldsymbol{b} = (b_1, b_2, \cdots, b_5)$: preorder cost b_j per unit of part j (before the demand is known).

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

where $Q(\boldsymbol{x}) = \mathrm{E}_{\boldsymbol{\omega}}[Z] = \sum_{i=1}^{8} p_i c_i z_i$ is taken w. r. t. the probability distribution of $\boldsymbol{\omega} = \boldsymbol{D}$.

$\boldsymbol{z_i}$: product type i.

$\boldsymbol{c_i}(< 0)$: which is the difference between the buying cost and the selling cost.

$\boldsymbol{p_i}$: probability of the random variable of the random demand vector $\boldsymbol{\omega} = \boldsymbol{D} = (D_1, D_2, \cdots, D_8)$.

The Second-Stage Problem and First-Stage Problem are essentially finding the maximum units produced when $c_i = l_i - q_i$ always < 0 due to the fact that the extra cost to complete each product is always lower than the selling cost. ($\Longrightarrow$ Production $\leq$ Demand)

4.1.1.c One Large Scale Linear Programming Model

In the special case of finitely many demand scenarios $d^1, \ldots, d^K$ occurring with positive probabilities $p_1, \ldots, p_K$, with $\sum_{k=1}^{K} p_k = 1$, the two stage problem can be written as **one large scale linear programming problem**:

$$\begin{aligned}
&\text{Min } c^\top x + \sum_{k=1}^{K} p_k \left[(l-q)^\top z^k - s^\top y^k\right] \\
&\text{s.t. } y^k = x - A^\top z^k, \quad k = 1, \ldots, K, \\
&\quad 0 \leq z^k \leq d^k, \quad y^k \geq 0, \quad k = 1, \ldots, K, \\
&\quad x \geq 0,
\end{aligned}$$

Where the minimization is performed over vector variables x and $z^k, y^k, k = 1, \ldots, K$. We have integrated the second stage problem into this formulation, but we had to allow for its solution $\left(z^k, y^k\right)$ to depend on the scenario k, because the demand realization d^k is different in each scenario. Hence, this has the numbers of variables and constraints roughly proportional to K.

4.1.2 Implementation

```python
# Install matplot to draw a graph
import matplotlib.pyplot as plt
import gurobipy as gp
import numpy as np
from gurobipy import GRB
from numpy import random
from array import*

# Define parameters
n = 8  # Number of products
m = 5  # Number of parts to be ordered before production

# Function to generate non-zero random matrix
def generate_non_zero_matrix(shape, max_value):
    matrix = random.randint(0, max_value + 1, size=shape)
    while np.all(matrix == 0):
        matrix = random.randint(0, max_value + 1, size=shape)
    return matrix

# Simulate data for two-stage stochastic
b_r = random.randint(10, 25, m)  # Random data for pre-order cost per unit
l_r = random.randint(50, 100, n)   # Random data for extra cost to complete each product
q_r = random.randint(500, 1000, n)    # Random data for selling cost
s_r = random.randint(5, 15, m)    # Random data for pre-order resale costs per unit
A_r = generate_non_zero_matrix((n, m), 3)  # Random data for the matrix A

# Ensure that b[j] is always greater than s[j]
for j in range(m):
    while b_r[j] <= s_r[j]:
        b_r[j] = random.randint(10, 25)

print(f"Pre-order cost per unit (b): {b_r}")
print(f"Extra cost to complete each product (l): {l_r}")
print(f"Selling cost (q): {q_r}")
print(f"Extra cost - Selling cost: {l_r - q_r}")
print(f"Pre-order resale costs per unit (s): {s_r}")
print(f"Matrix A:\n{A_r}\n")

# Simulate random demand vectors for each scenario
S = 2  # Number of scenarios
p_s = 1 / 2  # Probability for each scenario

result = np.zeros(100)
result_x = np.zeros((100, 5))

for index in range(100):
  b = b_r
  l = l_r
  q = q_r
  s = s_r
  A = A_r

  p_k = random.binomial(10, 0.5, size=(S, n))

  # Normalize the demand vectors for each scenario
  p_k_normalized = np.apply_along_axis(lambda x: x / np.sum(x), axis=1, arr=p_k)

  print(f"Random demand matrix (p_k):\n{p_k}")
```

```
59
60  M = gp.Model("2Stage-SPL")
61  x = M.addVars(m, name="x", vtype=GRB.INTEGER)
62  # Scenario 1
63  y = M.addVars(m, name="y", vtype=GRB.INTEGER)
64  z = M.addVars(n, name="z", vtype=GRB.INTEGER)
65  # Scenario 2
66  y1 = M.addVars(m, name="1", vtype=GRB.INTEGER)
67  z1 = M.addVars(n, name="2", vtype=GRB.INTEGER)
68
69  M.Params.OutputFlag = 0
70  M.setParam('OutputFlag', 0)  # Telling gurobi to not be verbose
71  M.params.logtoconsole = 0
72
73  # Constraints
74  for i in range(n):
75        M.addConstr(z[i] <= p_k[0, i])
76        M.addConstr(z1[i] <= p_k[1, i])
77        M.addConstr(z[i] >= 0)
78        M.addConstr(z1[i] >= 0)
79
80  for j in range(m):
81        M.addConstr(y[j] == x[j] - gp.quicksum(A[i, j] * z[i] for i in range(n)))
82        M.addConstr(y[j] >= 0)
83        M.addConstr(y1[j] == x[j] - gp.quicksum(A[i, j] * z1[i] for i in range(n)))
84        M.addConstr(y1[j] >= 0)
85        M.addConstr(x[j] >= 0)
86
87  # Objective function
88  M.setObjective(
89      gp.quicksum(b[j] * x[j] for j in range(m))
90      + p_s * ( gp.quicksum( (l[i] - q[i]) * z[i] for i in range(n) ) - gp.quicksum( s[j] *
        ↪  y[j] for j in range(m) ) )    # Scenario 1
91      + p_s * ( gp.quicksum( (l[i] - q[i]) * z1[i] for i in range(n) ) - gp.quicksum( s[j] *
        ↪  y1[j] for j in range(m) ) )  # Scenario 2
92      , GRB.MINIMIZE)
93
94  # Solve the model
95  M.optimize()
96
97  # Get the optimal solution
98  optimal_x = [var.x for var in M.getVars() if var.varName.startswith('x')]
99  optimal_y = [var.x for var in M.getVars() if var.varName.startswith('y')]
100 optimal_z = [var.x for var in M.getVars() if var.varName.startswith('z')]
101 optimal_y1 = [var.x for var in M.getVars() if var.varName.startswith('1')]
102 optimal_z1 = [var.x for var in M.getVars() if var.varName.startswith('2')]
103
104 # Initialize E_Z
105 E_Z = 0
106
107 # Calculate the expected value, the average value of products in both scenarios
108 for i in range(n):
109     E_Z += p_s * p_k_normalized[0, i] * (l[i] - q[i]) * optimal_z[i]
110     E_Z += p_s * p_k_normalized[1, i] * (l[i] - q[i]) * optimal_z1[i]
111
112 # Print the results
113 result_x[index] = optimal_x # Save optimal_x values
114 print(f"Optimal solution for x: {optimal_x}")
115
116 print(f"Optimal solution for y (Scenario 1): {optimal_y}")
117 print(f"Optimal solution for z (Scenario 1): {optimal_z}")
```

```
  print(f"Optimal solution for y1 (Scenario 2): {optimal_y1}")
  print(f"Optimal solution for z1 (Scenario 2): {optimal_z1}")
  print(f"Expected value: {round(E_Z, 4)}")

  result[index] = M.objVal # Save optimal value of the model
  print(f"Optimal value of the model: {result[index]} \n")

Average_optimal_x = np.zeros(5);
for index in range(100):
    Average_optimal_x += result_x[index]

Average_optimal_x = Average_optimal_x / 100
Total_cost_order = 0
for index in range(m):
    print(f"\nAverage cost to order parts type {index} is: {round(Average_optimal_x[index]) *
    ↪  b[index]}, and the number of pre-ordered parts is: {round(Average_optimal_x[index])}")
    Total_cost_order += round(Average_optimal_x[index]) * b[index]

print(f"\nTotal cost required to order: {Total_cost_order}")

Average_optimal_value = 0;
for index in range(100):
    Average_optimal_value += result[index]

Average_optimal_value = Average_optimal_value / 100
print(f"\nAverage value of the objective function: {Average_optimal_value}")

# Draw chart to show clearly the optimal cost
result = np.insert(result, 0, np.nan) # Start with 1
plt.plot(result, marker='o', linestyle='-')
plt.axhline(y=Average_optimal_value, color='red', linestyle='--', label='Average Value')
plt.xlim(0, len(result) + 10)

# Show the average value
plt.annotate(f'{Average_optimal_value:.2f}',
             xy=(len(result) + 3, Average_optimal_value),
             xytext=(len(result) + 1, Average_optimal_value + 260),
             ha='center', va='bottom', color='red', fontsize=10,
             bbox=dict(boxstyle='round,pad=0.3', edgecolor='red', facecolor='white'))

plt.legend() # Display caption

plt.xlabel('Iteration')
plt.ylabel('Value')
plt.title('Optimization Result')

plt.show()
```

4.1.3 Outputs and Explanations

Demand vector D is not deterministic. The chance that each situation where the received value $D = d$ will occur is $P(D = d)$. Scenarios are cases which having a probability $P(D = d)$ for each cost. In order to minimize risk, using the expectation formula's average to create the one linear programming that uses such as the model displayed above.

Additionally, to identify the most accurate approximation in practice, it is necessary to build the model multiple times using the different data's random demand vector D. And eliminate some unrealistic cases to build up a model closest to reality in 2 the diagrams below.

```
Pre-order cost per unit (b): [19 16 18 15 18]
Extra cost to complete each product (l): [81 56 63 68 94 92 50 75]
Selling cost (q): [831 991 835 664 766 527 744 795]
Extra cost - Selling cost: [-750 -935 -772 -596 -672 -435 -694 -720]
Pre-order resale costs per unit (s): [14  6 11  6  5]
Matrix A:
[[3 0 0 2 3]
 [3 1 0 3 2]
 [0 1 0 2 2]
 [1 1 3 0 0]
 [2 1 0 2 3]
 [2 1 1 3 3]
 [2 0 2 0 2]
 [2 1 0 3 3]]
```

Figure 9: Random data vectors

```
Random demand matrix (p_k):
[[7 8 8 9 5 6 6 6]
 [7 6 6 3 3 5 7 5]]
Optimal solution for x: [100.0, 42.0, 45.0, 100.0, 116.0]
Optimal solution for y (Scenario 1): [0.0, 0.0, 0.0, 0.0, 0.0]
Optimal solution for z (Scenario 1): [7.0, 8.0, 8.0, 9.0, 5.0, 6.0, 6.0, 6.0]
Optimal solution for y1 (Scenario 2): [18.0, 14.0, 17.0, 20.0, 18.0]
Optimal solution for z1 (Scenario 2): [7.0, 6.0, 6.0, 3.0, 3.0, 5.0, 7.0, 5.0]
Expected value: -4577.4937
Optimal value of the model: -27723.0

Random demand matrix (p_k):
[[5 9 3 6 2 2 6 4]
 [5 2 3 5 6 5 4 8]]
Optimal solution for x: [76.0, 29.0, 32.0, 73.0, 90.0]
Optimal solution for y (Scenario 1): [0.0, 3.0, 0.0, 8.0, 15.0]
Optimal solution for z (Scenario 1): [5.0, 9.0, 3.0, 6.0, 2.0, 2.0, 6.0, 4.0]
Optimal solution for y1 (Scenario 2): [4.0, 0.0, 4.0, 0.0, 0.0]
Optimal solution for z1 (Scenario 2): [5.0, 2.0, 3.0, 5.0, 6.0, 5.0, 4.0, 8.0]
Expected value: -4011.021
Optimal value of the model: -21408.5
```

Figure 10: Here are our implementations (Implementing total 100 times)

```
Average cost to order parts type 0 is: 1539, and the number of pre-ordered parts is: 81
Average cost to order parts type 1 is: 512, and the number of pre-ordered parts is: 32
Average cost to order parts type 2 is: 612, and the number of pre-ordered parts is: 34
Average cost to order parts type 3 is: 1215, and the number of pre-ordered parts is: 81
Average cost to order parts type 4 is: 1746, and the number of pre-ordered parts is: 97
Total cost required to order: 5624
```

Figure 11: Average Cost to Order Parts and Total Cost Required to Order

```
Average value of the objective function: -22529.42
```

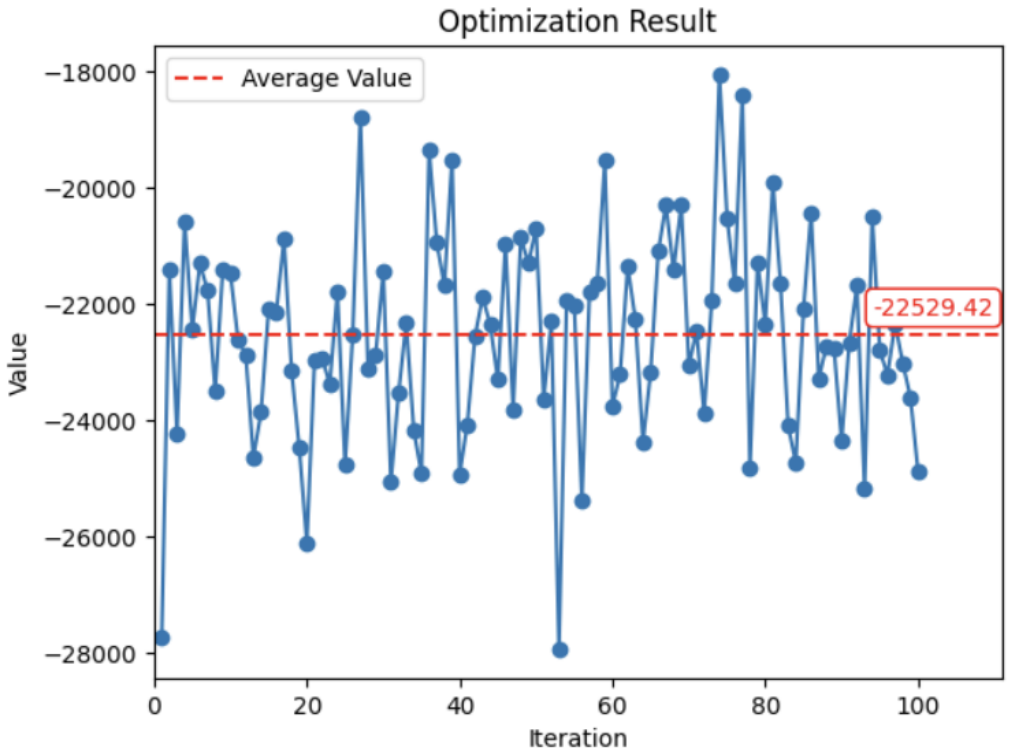


Figure 12: Average value of the objective function after several implementations

4.2 PROBLEM 2

4.2.1 Introduction to Minimum-cost flow problem

The minimum-cost flow problem (MCFP) is an optimization and decision problem to find the cheapest possible way of sending a certain amount of flow through a flow network. This problem is applied in many fields and the typical one is the distribution problem which involves finding the best delivery route from a factory to a warehouse where the road network has some capacity and cost-associated. Moreover, it can be used to reconstruct the Left Ventricle from X-ray Projections, Optimal Loading of a Hopping Airplane, Scheduling with Deferral Costs, etc.

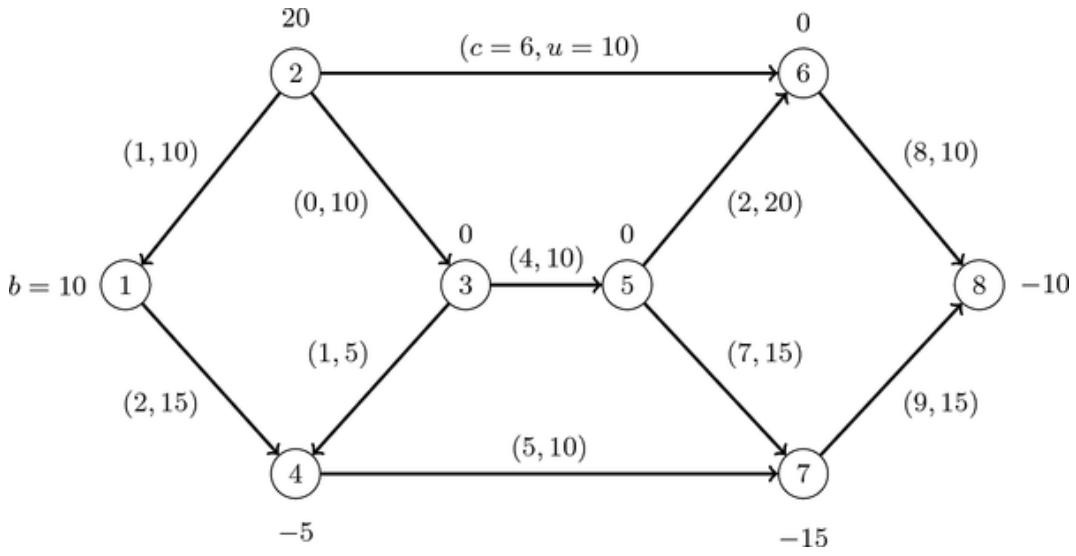


Figure 13: Minimum-cost flow problem.

There are several well-known fundamental algorithms used to solve this problem, such as Cycle canceling, Successive shortest path, Network simplex algorithm and Out-of-kilter algorithm, etc. In this assignment, we are going to delve into the theory and the implementation of Successive shortest path algorithm as well as use Network simplex algorithm to verify the efficiency of it.

4.2.2 Theoretical Basis

4.2.2.a Mathematical modeling problem

Let $G = (V,E)$ be a directed network defined by a set V of vertexes (nodes) and set E of edges (arcs). For each edge $(i, j) \in E$, it has two associated functions: the positive valued function **capacity** $u(i, j)$ (u_{ij} denotes the maximum amount of flow on the edge (i, j)) and the positive valued function **cost** $c(i, j)$ (c_{ij} denotes the cost per unit of flow on the edge (i, j)). For each vertex $i \in V$ a integer-valued function $v(i)$ is assigned, representing the **balance** of that vertex. If $v(i) > 0$, vertex i is a **supply node** and if $v(i) < 0$, vertex i is a **demand node**. Otherwise, the vertex i with $v(i) = 0$ is called **transshipment**. If every parameter needs to be clearly specified, the directed graph G is referred to as a transportation network and is denoted as

$G = (V, E, u, c, v)$. Using x_{ij} to represent the flow on arc $(i, j) \in E$, the optimization model for the minimum cost flow problem can be expressed as follows:

$$min\ z(x) = \sum_{(i,j)\in E} c_{ij}x_{ij}$$

subject to

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = v(i)\ \forall i \in V$$

$$0 \leq x_{ij} \leq u_{ij}\ \forall (i, j) \in E$$

The first constraint states that the total outflow of a node subtract the total inflow of the node must be equal to mass balance (supply/demand value) of this node. This is known as the **mass balance constraints**. The second constraint states that the flow of each edge must be less than or equal the capacity of that edge, called **flow bound constraints**. This optimization model illustrates a common connection between warehouses and shops, such as in a scenario involving a single type of product. The goal is to meet the demand of each shop by transferring goods from a subset of warehouses, while minimizing transportation costs.

4.2.2.b Finding a solution

If $\delta = \sum_{i\in V} v(i) \neq 0$ the problem has no solution, because it does not satisfy the mass balance constraints anymore. It can be avoided by adding a special node, but our implementation do not consider it. We just conclude the problem has no solution, if $\delta \neq 0$.

However, even if $\delta = 0$ it is not sure that the edges have enough capacities to allow transfer flow from supply nodes to demand ones. To determine the feasibility of the network's flow, it is essential to identify a transfer path that satisfies all the constraints of the problem. While this feasible solution may not necessarily be optimal, its absence would lead to the problem unsolvable.

In our implementation, we always add a source node s and a sink node t are created, called *"dummy source"* and *"dummy sink"*, respectively. For each node $i \in V$ with $v(i) > 0$ (supply node), a source arc (s, i) with capacity $v(i)$ and cost 0 is added to G, $v(s) = \sum_{i\in supply\ node} v(i)$. While each node $i \in V$ with $v(i) < 0$ (demand node), a sink arc (i, t) with capacity $-v(i)$ and cost 0 is added to G, $v(t) = \sum_{i\in demand\ node} v(i)$. For a maximum flow problem from s to t, if the maximum flow saturates all the source and sink arcs, then the problem has a feasible solution; otherwise, it is infeasible. In conclusion, for any min-cost flow problem some crucial conditions for tackling the problem should be met:

- the supply/demand balance
- the existence of a feasible solution
- the non-existence of incapacitated negative cycles (If the network contains a negative cycle with infinite capacity, supposing that the network has a feasible solution, the objective function will be unbounded).

4.2.2.c Assumptions

Without these following asumptions we also can solve this problem, but we add these to make our problem easier.

Assumption 1: *All data* $(u_{ij},\ c_{ij},\ v(i))$ *are integers.*

As the computer works with rational numbers, this assumption is not restrictive in practice while rational numbers can be converted to integers by multiplying by a suitable large number.

Assumption 2: *The network is directed.*

If the network is undirected, we also have way to transfrom a undirected network into a directed one. Nevertheless, I will not discuss here, because supposing that our input network is directed network.

Assumption 3: *All costs associated with edges are nonnegative.*

This assumption imposes a loss of generality, but it makes us more simple control the residual network.

Assumption 4: *The supply/demand function satisfies the condition* $\sum_{i \in V} v(i) = 0$ *and the minimum cost flow problem has a feasible solution.*

If the network does not satisfy the first part of this assumption, we will conclude the problem has no solution (while we can use corresponding transformation to solve). If the second part of the assumption is not met then surely the solution does not exist.

4.2.3 Algorithms

4.2.3.a Dijkstra Algorithm

Dijkstra's algorithm, conceived by computer scientist Edsger Dijkstra in 1956, stands as a fundamental and widely-used algorithm in the realm of graph theory and optimization.

It serves the purpose of finding the shortest path between two nodes in a weighted graph, where each edge possesses a non-negative weight. The algorithm operates by maintaining a set of tentative distances from a designated start node to all other nodes in the graph, continually updating these distances as it explores neighboring nodes.

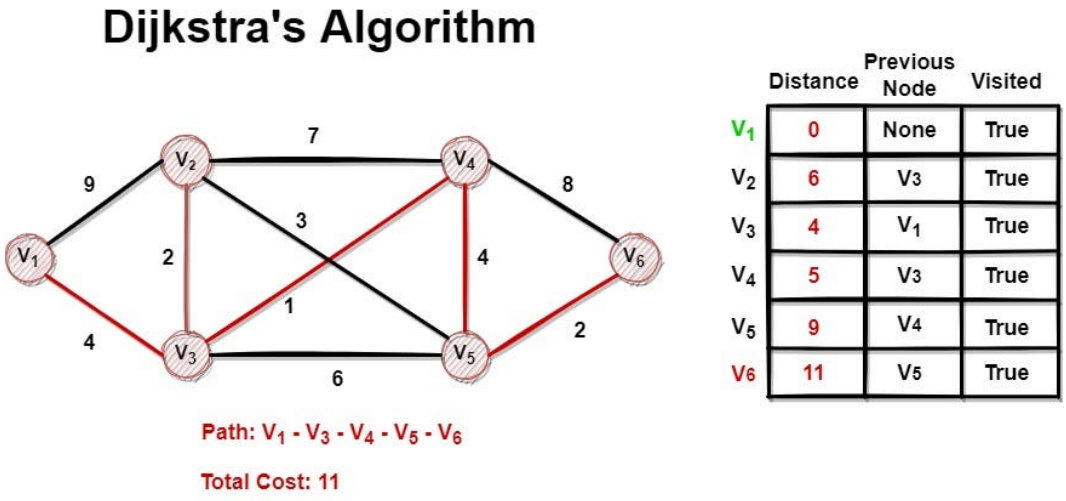


Figure 14: Dijkstra algorithm.

How the algorithm work: Dijkstra's algorithm operates by systematically exploring nodes in a graph, updating tentative distances from a chosen start node to all other nodes.

- **Step 1:** The algorithm maintains a priority queue to keep track of nodes with their current tentative distances. It begins by initializing the distance to the start node as zero and the distances to all other nodes as infinity.
- **Step 2:** It then iteratively selects the node with the smallest tentative distance from the priority queue, exploring its neighbors and updating their tentative distances if a shorter path is found.
- **Step 3:** The process continues until the destination node is reached or all reachable nodes have been explored.

Dijkstra's algorithm ensures that once a node's tentative distance is finalized, it is the shortest path from the start node to that specific node. The priority queue ensures that the algorithm explores nodes with the shortest tentative distances first, making it an efficient and reliable method for finding the shortest path in weighted graphs.

4.2.3.b Successive Shortest Path Algorithm

Working with Residual Networks

Let G be a network and x be a feasible solution of the minimum cost flow problem. Suppose that an edge (i, j) in E carries x_{ij} units of flow. The residual capacity of the edge (i, j) is defined as $r_{ij} = u_{ij} - x_{ij}$. This means that additional r_{ij} units of flow can be sent from vertex i to vertex j. The existing flow x_{ij} on the arc (i, j) can also be canceled by sending up x_{ij} units of flow from j to i over the arc (i, j).

Sending a unit of flow from i to j along the arc (i, j) increases the objective function by c_{ij}, while sending a unit of flow from j to i on the same arc decreases the flow cost by c_{ij} . Based on these ideas, for a transportation network G = (V, E) and a feasible solution x, the residual network with respect to the given flow x is denoted by G_x = (V, E_x), where E_x is the set of residual edges corresponding to the feasible solution x.

Each arc (i, j) in E is replaced by two arcs (i, j), (j, i): the arc (i, j) has cost c_{ij} and (residual) capacity $r_{ij} = u_{ij} - x_{ij}$, and the arc (j, i) has cost $-c_{ij}$ and (residual) capacity $r_{ji} = x_{ij}$.

Shortest path optimality conditions

We denote the distance labels $d(i)$ defines shortest path distances from source node s to a node i $\forall i \in V$. Then, $d(i)$ satisfy the following shortest path optimality conditions:

$$d(j) \leq d(i) + c_{ij} \ \forall (i, j) \in E$$

Equivalently,

$$c_{ij}^d = c_{ij} + d(i) - d(j) \geq 0 \ \forall (i, j) \in E$$

c_{ij}^d is an optimal **"reduced cost"** for arc (i, j) in the sense that it measures the cost of this arc relative to the shortest distances $d(i)$ and $d(j)$. Moreover, since $d(j) = d(i) + c_{ij}$, if arc (i, j) is on the shortest path connecting the source node s to any other node, the shortest path uses only zero reduced cost arcs. Consequently, we simply find a path from source node s to every other node that uses only arcs with zero reduced costs.

From that idea, for each vertex i $\forall i \in V$ we refer to $\pi(i) = -d(i)$. Thus, we define the *"reduced cost"* of an arc (i, j) as $c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)$. These reduced costs are applicable to the residual network as well as the original network. We define the reduced costs in the residual network just as we did the costs, but now using c_{ij}^π in place of c_{ij}. It leads to some following properties:

- *For any directed path P from node k to l,* $\sum_{(i,j)\in P} c_{ij}^\pi = \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l)$
- *For any directed cycle W,* $\sum_{(i,j)\in W} c_{ij}^\pi = \sum_{(i,j)\in W} c_{ij}$

This property implies that the node potentials do not change the shortest path between any pair of nodes k and l, since the potentials increase the length of every path by a constant amount $\pi(k) - \pi(l)$. This property also implies that if W is a negative cycle with respect to c_{ij} as arc costs, it is also a negative cycle with respect to c_{ij}^π as arc costs. In summary, the reduced cost do not change our solution.

For each arc $(i, j) \in E$ the following shortest path optimality condition is satisfied:

$$-\pi(j) \leq -\pi(i) + c_{ij} \ (\pi(i) = -d(i))$$

$$\implies 0 \leq c_{ij} - \pi(i) + \pi(j) \text{ and } c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)$$

Then: $c_{ij}^\pi \geq 0 \ \forall (i, j) \in E_x$ **(Optimality condition)**

We can interpret this optimality condition by this example. We have c_{ij} as the cost of transporting 1 unit of a commodity from node i to node j through the arc (i, j), and $d(i) = -\pi(i)$ as the cost of obtaining a unit of this commodity at node i. Then $c_{ij} + d(i)$ is the cost of the commodity at node j if we obtain it at node i and transport it at at node i. The optimality condition, $c_{ij} - \pi(i) + \pi(j) \geq 0$ or $d(j) \leq c_{ij} + d(i)$, states that the cost of obtaining the commodity at node j is no more than the cost of the commodity if we obtain it at node i and incur the transportation cost in sending it from node i to j. In conclusion, there might be a more cost-effective way to transport the commodity to node j via other nodes.

Algorithm

The successive shortest path algorithm maintains optimality of the solution at every step and strives to attain feasibility. It maintains a solution x that satisfies the **flow bound constraints**, but violates the **mass balance constraints** of the nodes. At each step, the algorithm sends flow from *s(dummy source)* to *t(dummy sink)* along a shortest path in the residual network. The algorithm terminates when the current solution satisfies all the mass balance constraints (balance of s and t equal to 0). And the node potentials play a very important role in this algorithm, we use them to maintain nonnegative arc lengths so that we can solve the shortest path problem more efficiently by Dijkstra Algorithm.

algorithm *successive shortest path;*

begin

$x := 0$ and $\pi := 0$*;*

Add dummy source s and dummy sink t;

while $v(s) \neq 0$ and $v(t) \neq 0$ **do**

begin

determine the shortest path distances from s to all other nodes by Dijkstra;

let P denote the shortest path from s to t;

update potentials $\pi := \pi - d$;

Find max flow δ for P;

augment δ units of flow along P;

update x, residual network and reduced costs;

end;

Calculate the total cost (just consider edge existing in initial network(i.e. cost>0));

end;

Then we will illustrate our algorithm through an example. This is the initial network that satisfies all the assumptions. At first reduced time equals to initial time as well as the residual capacity equals to initial capacity. Each edge associated with 3 values are reduced time, residual capacity and flow, respectively.

```
node 1, balance: 4, potential:0
destination:2 residualCapacity: 1, reducedTime: 1, time: 1
destination:3 residualCapacity: 3, reducedTime: 5, time: 5
node 2, balance: 0, potential:0
destination:3 residualCapacity: 2, reducedTime: 1, time: 1
destination:4 residualCapacity: 1, reducedTime: 4, time: 4
node 3, balance: 0, potential:0
destination:4 residualCapacity: 3, reducedTime: 2, time: 2
node 4, balance: -4, potential:0
```

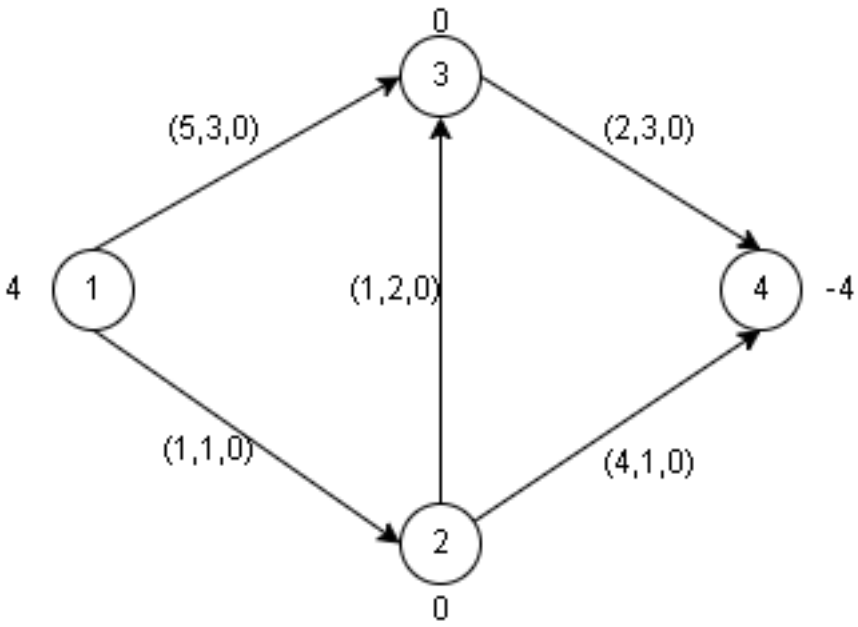


Figure 15: Initial Network

Then we add the dummy source node(node 0) and the dummy sink node(node 5).

```
node 1, balance: 4, potential:0
destination:2 residualCapacity: 1, reducedTime: 1, time: 1
destination:3 residualCapacity: 3, reducedTime: 5, time: 5
node 2, balance: 0, potential:0
destination:3 residualCapacity: 2, reducedTime: 1, time: 1
destination:4 residualCapacity: 1, reducedTime: 4, time: 4
node 3, balance: 0, potential:0
destination:4 residualCapacity: 3, reducedTime: 2, time: 2
node 4, balance: -4, potential:0
destination:5 residualCapacity: 4, reducedTime: 0, time: 0
node 0, balance: 4, potential:0
destination:1 residualCapacity: 4, reducedTime: 0, time: 0
node 5, balance: -4, potential:0
```

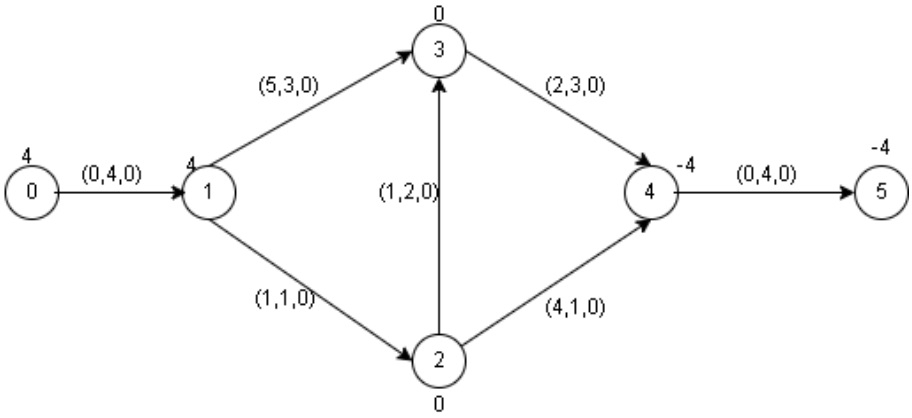


Figure 16: Network after add dummy source and sink

Next we start the algorithm, and this is output after executing first loop. That mean the shortest path from dummy source to dummy sink is $0 \to 1 \to 2 \to 3 \to 4 \to 5$ and the flow we send this path is 1.

```
[0, 1, 2, 3, 4, 5]
flow 1
node 1, balance: 4, potential:0
destination:2 residualCapacity: 0, reducedTime: 0, time: 1
destination:3 residualCapacity: 3, reducedTime: 3, time: 5
destination:0 residualCapacity: 1, reducedTime: 0, time: 0
node 2, balance: 0, potential:-1
destination:3 residualCapacity: 1, reducedTime: 0, time: 1
destination:4 residualCapacity: 1, reducedTime: 1, time: 4
destination:1 residualCapacity: 1, reducedTime: 0, time: -1
node 3, balance: 0, potential:-2
destination:4 residualCapacity: 2, reducedTime: 0, time: 2
destination:2 residualCapacity: 1, reducedTime: 0, time: -1
node 4, balance: -4, potential:-4
destination:5 residualCapacity: 3, reducedTime: 0, time: 0
destination:3 residualCapacity: 1, reducedTime: 0, time: -2
node 0, balance: 3, potential:0
destination:1 residualCapacity: 3, reducedTime: 0, time: 0
node 5, balance: -3, potential:-4
destination:4 residualCapacity: 1, reducedTime: 0, time: 0
```

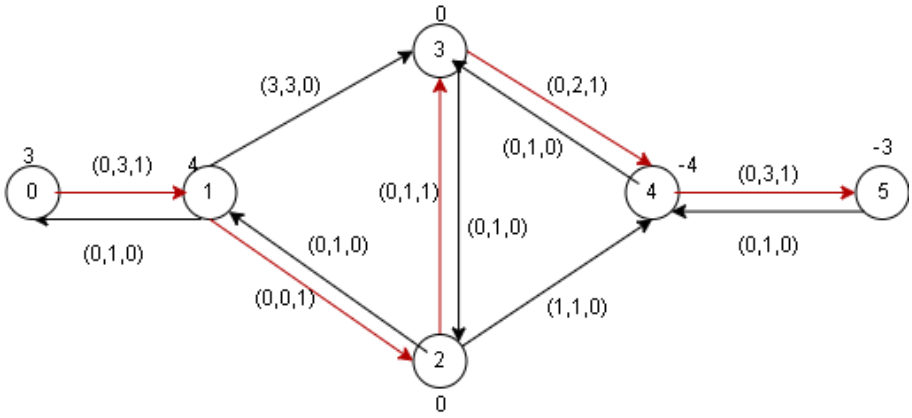


Figure 17: Residual network after first loop

Next we execute the second loop. The shortest path from dummy source to dummy sink is $0 \to 1 \to 3 \to 4 \to 5$ and the flow we send this path is 2.

```
[0, 1, 3, 4, 5]
flow 2
node 1, balance: 4, potential:0
destination:2 residualCapacity: 0, reducedTime: 0, time: 1
destination:3 residualCapacity: 1, reducedTime: 0, time: 5
destination:0 residualCapacity: 3, reducedTime: 0, time: 0
node 2, balance: 0, potential:-4
destination:3 residualCapacity: 1, reducedTime: 0, time: 1
destination:4 residualCapacity: 1, reducedTime: 1, time: 4
destination:1 residualCapacity: 1, reducedTime: 3, time: -1
node 3, balance: 0, potential:-5
destination:4 residualCapacity: 0, reducedTime: 0, time: 2
destination:2 residualCapacity: 1, reducedTime: 0, time: -1
destination:1 residualCapacity: 2, reducedTime: 0, time: -5
node 4, balance: -4, potential:-7
destination:5 residualCapacity: 1, reducedTime: 0, time: 0
destination:3 residualCapacity: 3, reducedTime: 0, time: -2
node 0, balance: 1, potential:0
destination:1 residualCapacity: 1, reducedTime: 0, time: 0
node 5, balance: -1, potential:-7
destination:4 residualCapacity: 3, reducedTime: 0, time: 0
```

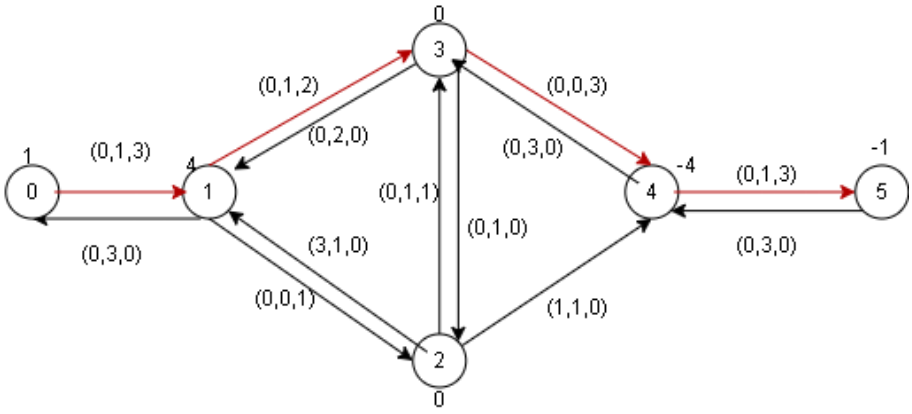


Figure 18: Residual network after second loop

Next we execute the third loop. The shortest path from dummy source to dummy sink is $0 \to 1 \to 3 \to 2 \to 4 \to 5$ and the flow we send this path is 1. In this loop, the speciality is the $3 \to 2$ edge which means we send back 1 unit which was sent from $2 \to 3$ in the first loop.

```
[0, 1, 3, 2, 4, 5]
flow 1
node 1, balance: 4, potential:
destination:2 residualCapacity: 0, reducedTime: 0, time: 1
destination:3 residualCapacity: 0, reducedTime: 0, time: 5
destination:0 residualCapacity: 4, reducedTime: 0, time: 0
node 2, balance: 0, potential:-4
destination:3 residualCapacity: 2, reducedTime: 0, time: 1
destination:4 residualCapacity: 0, reducedTime: 0, time: 4
destination:1 residualCapacity: 1, reducedTime: 3, time: -1
node 3, balance: 0, potential:-5
destination:4 residualCapacity: 0, reducedTime: 0, time: 2
destination:2 residualCapacity: 0, reducedTime: 0, time: -1
destination:1 residualCapacity: 3, reducedTime: 0, time: -5
node 4, balance: -4, potential:-8
destination:5 residualCapacity: 0, reducedTime: 0, time: 0
destination:3 residualCapacity: 3, reducedTime: 1, time: -2
destination:2 residualCapacity: 1, reducedTime: 0, time: -4
node 0, balance: 0, potential:0
destination:1 residualCapacity: 0, reducedTime: 0, time: 0
node 5, balance: 0, potential:-8
destination:4 residualCapacity: 4, reducedTime: 0, time: 0
```

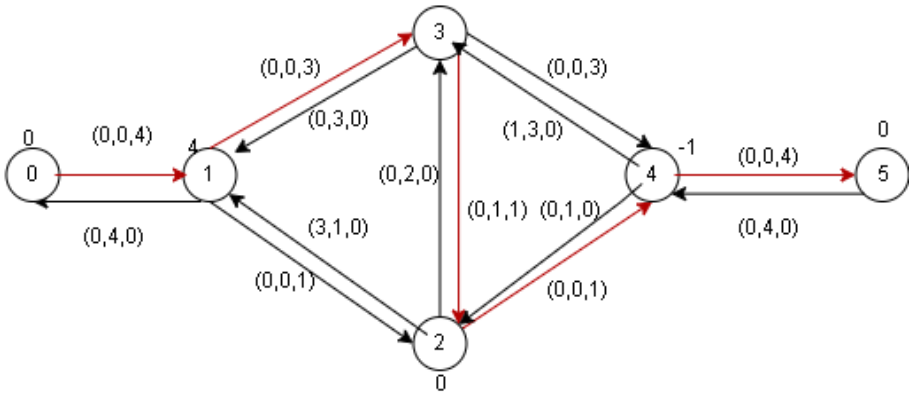


Figure 19: Residual network after third loop

Then the while loop terminates because the mass balance constraints is satisfied. Then here is the output.

```
Minimum cost of succesiveShortestPathAlgorithm: 26
1->2         Flow: 1
1->3         Flow: 3
2->3         Flow: 0
2->4         Flow: 1
3->4         Flow: 3
```

4.2.4 Implementation

4.2.4.a Data generation

To verify the effectiveness of the studied Algorithm, we need a short code to generate a large enough data set for testing. The problems in this program is that we have to handle well the capacity value for each edge as well as insurance that there is a solution for that graph. The idea to create this program is that:

- We assume that there are 4 source nodes and 4 sink nodes. Between of them is 4xn matrix of nodes.
- We connect all nodes together to create a complete multipartite graph.
- Randomly generate time and capacity values such that the capacity values must satisfy the total path from one part to another must be greater than the number of people.

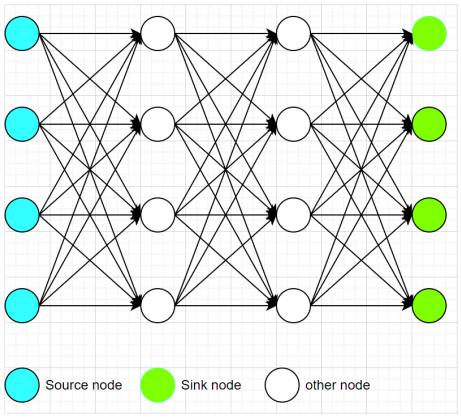


Figure 20: Random graph.

4.2.4.b Dijkstra Algorithm

Using the previous ideas that are demonstrated in the above section. First of all, we have create a Priority Queue data structure for further use. Here is how we implement it using Python:

```python
class PriorityQueue:
    def __init__(self,element):
        self.queue = [element]

    def add(self, element):
        i = 0

        if self.isEmpty():
            self.queue = [element]
            return

        while i < len(self.queue) and element[1] < self.queue[i][1]:
            i += 1

        self.queue.insert(i, element)

    def pop(self):
        return self.queue.pop()[0]

    def contains(self,element):
        return element in self.queue

    def isEmpty(self):
        return len(self.queue) == 0
```

This priority queue implementation uses a list to store elements, and the **add()** method ensures that elements are inserted in the correct order based on their priority. The priority is determined by the second element of each tuple. The **pop()** method removes and returns the element with the highest priority.

After creating the **PriorityQueue** class, our goal of this section is to implement the Dijkstra Algorithm.

```python
def dijkstra(nodes,root):
      costs = []
      for i in range(len(nodes)):
          costs.append((sys.maxsize, root)) # sys.maxsize == infinity

      costs[root] = (0, root)
      visited = [False for i in range(len(nodes))]
      queue = PriorityQueue((root,0))

      while not queue.isEmpty():
          node = queue.pop()
          visited[node] = True
          for edge in nodes[node].edges.values():
                if not visited[edge.desNode.nodeID] \
                      and edge.reducedTime + costs[node][0] < costs[edge.desNode.nodeID][0] \
                      and edge.residualCapacity > 0:
                    costs[edge.desNode.nodeID] = (edge.reducedTime + costs[node][0], node)

                if not queue.contains(edge.desNode.nodeID):
                    queue.add((edge.desNode.nodeID,costs[edge.desNode.nodeID]))

      print("costs list: " + str(costs))
      return costs
```

The function is designed to be passed a given source node (root) into it to find the shortest path from the source node to all other nodes in a graph. Our first step is to create a cost list. The costs list is initialized to store the current shortest distances (costs) from the source node to all other nodes. Each element is a tuple **(distance, previous_node)**, where distance represents the current shortest distance from the source to that node, and previous_node is the previous node in the path.

Next is to initialize the **visited list** for keeping track of whether each node has been visited. Additionally, we establish a priority queue, initializing it with an element representing the source node and its corresponding distance (cost) as the priority.

In the while loop, the algorithm iterates as long as the priority queue is not empty. It pops a node from the priority queue, marks it as visited, and explores its neighboring nodes.

For each neighboring edge, it checks if the neighbor node in the edge has not been visited and the capacity of the edge is validity and the newer cost calculated is less than the cost in the list, it will then update the newer cost into the list and also replace the **previous_node** element in the list into the correct one. Moreover, it also checks if the destination node in the current edge is visited or not. If not, then append the new node to the priority queue.

Take an example of an network as follow:
The output of **costs list** with the network as above would be:

```
costs list: [(0, 0), (0, 0), (1, 1), (2, 2), (4, 3), (4, 4)]
```

The list comprises six elements, each being a tuple. In each tuple, the first element represents the **minimum cost**, and the second element provides details about the **previous nodeID**. This implies that, to reach the current nodeID with the minimum cost, the path must have passed through the specified previous nodeID.

The list contains six elements, unlike the network, which has four vertexes. This difference arises

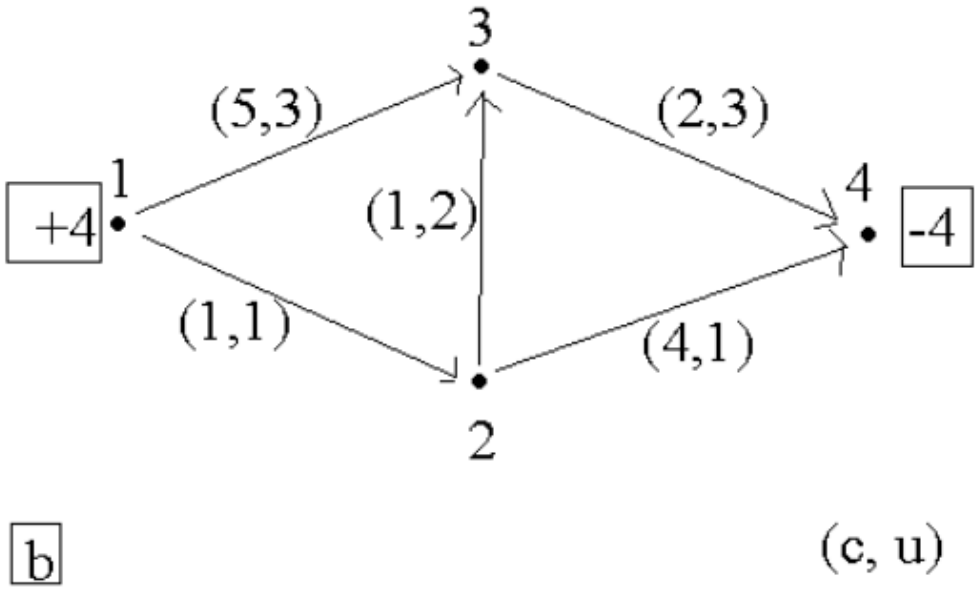


Figure 21: Example Network.

because our graph incorporates a function that generates the **Super Source** and **Super Sink**, requiring two additional spaces in the list.

Each index in the list represents information about the Node at that index (starting from 0). For example, in the previous list, the value at **index 4** is **(4, 3)**. This can be interpreted as the minimum cost to reach this node being 4, and to reach it, we must pass through the node with an ID equal to 3.

4.2.4.c Graph data structure:

This project is implemented on graph so it is necessary for creating a class to define a graph.However, before building a graph, we need to define two other important components: node and edge. Explanation for **"node class"**:

- **nodeID**: contain the ID of each node (integer).
- **Balance**: contain the demand of each node at a certain time(positvie for source nodes, negative for sink nodes and zero for remaining nodes).
- **originalBalance**: contain the original balance value.
- **potentialValue**: potential value is used for optimizing Dijkstra Algorithm.
- **edges**: this contain every edge from this node to other nodes. These information is saved in **dictionary** data type with key is destination node and value is **Edge** data structure which is defined below.

```
class Node:
    potentialValue = 0
    def __init__(self, nodeID, balance, edges):
        self.nodeID = nodeID
        self.balance = balance
        self.originalBalance = balance
        self.edges = edges

    def setBalance(self, balance):
        self.balance = balance

    def updatePotential(self, potentialValue):
        self.potentialValue = potentialValue

    def addEdges(self, *edges):
        for edge in edges:
            if self.nodeID != edge.desNode.nodeID:
                self.edges[edge.desNode.nodeID] = edge

    def printNode(self, printAllEdge = False):
        print("node "+str(self.nodeID) + ": balance" + str(self.balance) +
              ", potential:"+ str(self.potentialValue))
        if printAllEdge == True:
            for edge in self.edges.values():
                print("destination:" + str(edge.desNode.nodeID) + " residualCapacity: "+
                ↪  str(edge.residualCapacity)
                      + ", reducedTime: "+ str(edge.reducedTime))
```

Explanation for **"Edge class"**:

- **desNode**: save the destination node (using above node data structure).
- **time**: time travel on this edge.
- **capacity**: maximun capacity for this node.
- **residualCapacity**:the remaining capacity at the certain time.
- **reducedTime**: this is calculated from potential value to optimize Dijkstra Algorithm.

- **flow**: save the flow on that edge in a certain time.

```
class Edge:
    flow = 0
    def __init__(self, desNode, time, capacity):
        self.desNode = desNode
        self.time = time
        self.capacity = capacity
        self.residualCapacity = capacity
        self.reducedTime = time

    def updateResidualCapacity(self, residualCapacity):
        self.residualCapacity = residualCapacity

    def updateReducedTime(self, reducedTime):
        self.reducedTime = reducedTime
```

After defining 2 above classes, we started build the **graph**:

- **nodes**: this a data structure using for save information of all nodes in graph. In this program, we used **dictionary** data structure with key is the **nodeID** and value is above **node** data structure.
- **sources**: save the **nodeID** of all source nodes.
- **sinks**: save the **nodeID** of all sink nodes.

```
class Graph:
    nodes = {}
    sources = set()
    sinks = set()

    def __init__(self, nodeList = None):
        if nodeList == None: return
        for node in nodeList:
            self.nodes[node.nodeID] = node
            if node.balance > 0:
                self.sources.add(node.nodeID)
            elif node.balance < 0:
                self.sinks.add(node.nodeID)

    def updateReduceTime(self):

    def updateFlow(self ,flow, path):

    def updateBalance(self, root, end, flow):

    def updateResidualNetwork(self, path):

    def printGraph(self):

    def printFlow(self):

    def findPath(self):

    def succesiveShortestPathAlgorithm(self):

```

```
31      def checkMassBalanceConstraint(self):
32
33      def addDummySource(self):
34
35      def addDummySink(self):
36
37      def inputDataFromExcel(self, filepath):
38
39      def drawGraph(self, path, node_size=1500, node_color='orange', node_alpha=0.3,
40                        node_text_size=8, edge_alpha=0.8, edge_tickness=1, edge_text_pos=0.7,
                          ↪  text_font='sans-serif',
41                        special_edge_color='red', normal_edge_color = 'grey'):
42
43      def simplexAlgorithm(self):
```

inputDataFromExcel Function: The function is used to create a graph from excel file.

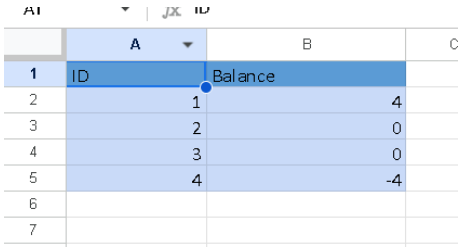
```
1   def inputDataFromExcel(self, filepath):
2         nodesList = []
3         nodeFile = pd.read_excel(filepath, sheet_name='Node')
4         edgeFile = pd.read_excel(filepath, sheet_name='Edge')
5         for index, row in nodeFile.iterrows():
6           nodesList.append(Node(int(row['ID']), int(row['Balance']), {}))
7         for node in nodesList:
8               self.nodes[node.nodeID] = node
9               if node.balance > 0:
10                  self.sources.add(node.nodeID)
11              elif node.balance < 0:
12                  self.sinks.add(node.nodeID)
13        for index, row in edgeFile.iterrows():
14          edge = Edge(self.nodes[row['end_ID']], row['time'], row['capacity'])
15          self.nodes[row['start_ID']].addEdges(edge)
```

Break out the variables in the function:

- **filepath**: path to excel file.

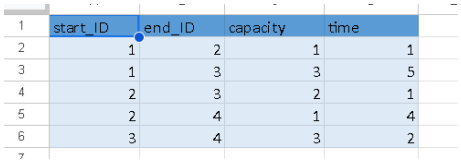
The structure of the excel file:

- The excel file must have 2 sheets. One named **"Node"** is used for node initialization and the other named **"Edge"** for edge initialization.
- In the **"Node"** sheet, there are 2 columns: **'ID'** (save the ID of node) and **'Balance'**(the original balance for each node).
- In the **"Edge"** sheet, there are 4 columns: **'start_ID'**(start node of edge), **'end_ID'**(end node of edge), **'capacity'**(capacity of edge), **'time'**(time travel on edge).



	A	B	C
1	ID	Balance	
2	1	4	
3	2	0	
4	3	0	
5	4	-4	
6			
7			

Figure 22: Node sheet.



	start_ID	end_ID	capacity	time
2	1	2	1	1
3	1	3	3	5
4	2	3	2	1
5	2	4	1	4
6	3	4	3	2

Figure 23: Edge sheet.

printGraph Function: print all information of graph in text.

```
def printGraph(self):
        strSources = "Sources:"
        strSinks = "Sinks:"
        for nodeID in self.sources:
            strSources += str(nodeID) +", "
        for nodeID in self.sinks:
            strSinks +=  str(nodeID) +", "
        print(strSources)
        print(strSinks)
        for node in self.nodes.values():
            node.printNode(True)
```

printFlow Function: print flow of each edge (just in initial network).

```
    def printFlow(self):
        for node in self.nodes.values():
          for edge in node.edges.values():
            if edge.time > 0:
              print(str(node.nodeID) + "->" + str(edge.desNode.nodeID) + "\t Flow: "
              ↪  +str(edge.flow))
```

drawGraph function: this function is used for graph visualization leveraging the Matplotlib and NetworkX libraries for graph rendering.

```
def drawGraph(self, path, node_size=1500, node_color='orange', node_alpha=0.3,
                      node_text_size=12, edge_alpha=0.8, edge_tickness=1, edge_text_pos=0.7,
                      ↪  text_font='sans-serif',
                      special_edge_color='red', normal_edge_color = 'grey'):
      G = nx.DiGraph()
```

```
5
6       normal_edges = []
7       path_edges = []
8
9       for node in self.nodes.values():
10        G.add_node(node.nodeID)
11        for edge in node.edges.values():
12          G.add_edge(node.nodeID, edge.desNode.nodeID, capacity=edge.residualCapacity,
            ↪  time=edge.time, flow=edge.flow)
13          normal_edges.append((node.nodeID, edge.desNode.nodeID))
14
15      for idx in range(len(path)-1):
16        path_edges.append((path[idx], path[idx+1]))
17        normal_edges.remove((path[idx], path[idx+1]))
18
19      graph_pos = nx.drawing.nx_pydot.graphviz_layout(G, 'circo')
20
21      nx.draw_networkx_edges(G, graph_pos, edgelist=normal_edges, edge_color='blue',
22                             width=edge_tickness, alpha=edge_alpha)
23
24      # Draw special edges with different color
25      nx.draw_networkx_edges(G, graph_pos, edgelist=path_edges, edge_color='red',
26                               width=edge_tickness, alpha=edge_alpha)
27
28      nx.draw_networkx_nodes(G, graph_pos, node_size=node_size, alpha=node_alpha,
        ↪  node_color=node_color)
29      node_labels = {node: str(node) for node in G.nodes()}
30      nx.draw_networkx_labels(G, graph_pos, labels=node_labels, font_size=node_text_size,
        ↪  font_family=text_font)
31      edge_labels = {(u, v): f"time:{G[u][v]['time']}, flow:{G[u][v]['flow']}"
32                     for u, v, d in G.edges(data=True) if 'time' in d and 'flow' in d}
33      nx.draw_networkx_edge_labels(G, graph_pos, edge_labels=edge_labels, label_pos=0.5,
        ↪  font_size = 8)
34
35      plt.show()
```

Break out the variables in function:

- **path:** moving path in array. This path will be hightlight in graph.
- **node_size:** Size of nodes. If an array it must be the same length as nodelist.
- **node_color:** Node color.
- **node_alpha:** The node transparency.
- **node_text_size:** Font size for text labels.
- **edge_alpha:** The edge transparency.
- **edge_tickness:** Line width of edges.
- **edge_text_pos:** position of label.
- **text_font:** font of label.

updateReduceTime Function: The function is used to update the reducedTime attribute of each edge in the whole graph.

```python
def updateReduceTime(self):
    for node in self.nodes.values():
        for edge in node.edges.values():
            if edge.residualCapacity > 0:
                reducedTime = edge.time - node.potentialValue + edge.desNode.potentialValue
                edge.updateReducedTime(reducedTime)
```

The function will iterate all nodes in the graph and access each node's edge list. For each edge in that list, it will check if the **residualCapacity** of that edge can be used or not (greater than 0), if yes, the **reducedTime** of that edge can be calculated as the theory formula: $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$. The iteration will loop until all of the nodes in the graph have been considered.

updateFlow Function: The function is used to update the flow of edges in the given path.

```python
def updateFlow(self ,flow, path):
    for idx in range(len(path)-1):
        presentNode = path[idx]
        afterNode = path[idx + 1]
        edge = self.nodes[presentNode].edges[afterNode]
        edge.residualCapacity = edge.residualCapacity - flow
        edge.flow = edge.flow + flow

        if presentNode in self.nodes[afterNode].edges:
            reverseEdge = self.nodes[afterNode].edges[presentNode]
            reverseEdge.flow = reverseEdge.flow - flow
            reverseEdge.residualCapacity += flow
```

Break out the variables in the function:

- **presentNode:** The nodeID at idx.
- **afterNode:** The nodeID at idx + 1.
- **reverseEdge:** The reverse edge.

The function takes a flow value and a path list containing nodeIDs as parameters. It proceeds to update the flow of the edge between each pair of consecutive nodes in the path list by adding the flow value. Additionally, the function decreases the residual capacity of the edge by the flow value. It checks whether there is a corresponding reverse edge for the present edge. If a reverse edge exists, the flow value is subtracted from its flow, and the residual capacity increase.

updateBalance Function: The function is used to update the balance value of root node and end node in the minimum path.

```python
def updateBalance(self, root, end, flow):
    self.nodes[root].balance -= flow
    self.nodes[end].balance += flow
```

After getting the appropriate flow value, this function will then update the balance of the dummy source node and the dummy sink node of the path. Now, the balance of the source node will be subtracted by the flow meaning that the people have gone from the source to the sink using the

minimum path. Additionally, the balance of the sink node in the path will be added with the value of flow. The algorithm terminates when the balance of these 2 nodes are 0.

updateResidualNetwork Function: The function will be used to update the our network by adding a reverse edges.

```
def updateResidualNetwork(self, path):
    for index in range(len(path) - 1):
        presentNode = path[index]
        afterNode = path[index + 1]
        if not (path[index] in self.nodes[path[index + 1]].edges):
        # Add edge
            newEdge = Edge(self.nodes[presentNode],
            ↪  -(self.nodes[presentNode].edges[afterNode].time),
            ↪  self.nodes[presentNode].edges[afterNode].flow)
            newEdge.reducedTime =  -(self.nodes[presentNode].edges[afterNode].reducedTime)
            self.nodes[afterNode].addEdges(newEdge)
```

Break out the variables in the function:

- **presentNode:** The nodeID at index.
- **afterNode:** The nodeID at index + 1.
- **newEdge:** The new edge that will be added to the network.

The concept involves creating a reverse edge when there exists a path from an initial node to a destination node. In this function, the newly generated edge has the initial node set as the preceding destination node, and the destination node is updated to the previous initial one. The attributes of the reverse edge include a time value equal to the negative of the forward edge's time, and the balance is assigned as the flow of the corresponding forward edge.

checkMassBalanceConstraint: The function is used to check the network whether the total balance of all is 0, if not we will not solve the network as we mentioned in the theory section.

```
def checkMassBalanceConstraint(self):
    sum=0
    for node in self.nodes.values():
        sum+=node.balance
    return (sum==0)
```

addDummySource: The function is used to add a dummy source as we discuss in the theory section.

```
def addDummySource(self):
    dummySource = Node(0, 0, {}) # initialize balance =0
    self.nodes[0]= dummySource
    for nodeID in self.sources:
        self.nodes[0].balance+= self.nodes[nodeID].balance
        edge=Edge(self.nodes[nodeID], 0, self.nodes[nodeID].balance)
        self.nodes[0].addEdges(edge)
```

Break out the variables in the function:

- **dummySource:** a dummy source node.

We create a new dummy source node and set its balance, add edges from it to every source nodes as we wish in the theory section.

addDummySink: The function is used to add a dummy sink as we discuss in the theory section.

```
def addDummySink(self):
    dummySink = Node(len(self.nodes), 0, {}) # initialize balance =0
    for nodeID in self.sinks:
        dummySink.balance+= self.nodes[nodeID].balance
        edge=Edge(dummySink, 0, -self.nodes[nodeID].balance)
        self.nodes[nodeID].addEdges(edge)
    self.nodes[len(self.nodes)]= dummySink
```

Break out the variables in the function:

- **dummySink:** a dummy sink node.

We create a new dummy sink node and set its balance, add edges from every sink nodes to this node as we wish in the theory section.

findPath Function: The function is used to return a shortest path from the dummy source node to the dummy sink node using the Dijkstra Algorithm.

```
def findPath(self):
    costs = dijkstra(self.nodes, 0)
    end=len(self.nodes) -1
    path = [end]

    while end != 0:
        path.append(costs[end][1])
        end = costs[end][1]

    for i in range(1, len(self.nodes)):
        if costs[i][0] != sys.maxsize:
            self.nodes[i].potentialValue -= costs[i][0]

    if path:
        path.reverse()
        return path
```

Break out the variables in the function:

- **costs:** A list contains the cost returned by the Dijkstra function.
- **end:** A variable which is initialized as the last node's index. It is used to traverse the predecessors indexes.
- **path:** A list contains the nodes' indexes dedicated to the shortest path.

Starting by applying Dijkstra's algorithm using the dijkstra function, which calculates the shortest paths from the dummy source node (assumed to be node 0) to all other nodes in the graph. After running Dijkstra's algorithm, the method reconstructs the shortest path from the source node (0) to the dummy sink node in the graph.

It initializes the end variable to the last node's index and iteratively appends predecessors to the path list until reaching the source node.

After that, the method then updates the potential values of nodes based on the calculated shortest paths. It subtracts the respective distance (cost) from each node's potential value.

Finally, if the path list is not empty, it reverses the order (since it was built backward) and returns the shortest path from the dummy source node to the dummy sink node.

succesiveShortestPathAlgorithm Function: The function represents the implementation of the Successive Shortest Path algorithm to solve the minimum-cost flow problem in a network.

```
def succesiveShortestPathAlgorithm(self):
    start=timer()
    while(self.nodes[0].balance != 0 and self.nodes[len(self.nodes) - 1].balance !=0 ):
        path= self.findPath()
        print(path)

        minCapacity = sys.maxsize
        for i in range(len(path)-1):
            residual = self.nodes[path[i]].edges[path[i+1]].residualCapacity
            if residual < minCapacity:
                minCapacity = residual

        flow = min(abs(self.nodes[path[0]].balance), abs(self.nodes[path[-1]].balance),
        ↪  minCapacity)
        print("flow %d " %flow)

        self.updateReduceTime()
        self.updateFlow(flow,path)
        self.updateBalance(path[0], path[-1], flow)
        self.updateResidualNetwork(path)

        cost = 0
        for node in self.nodes.values():
            for edge in node.edges.values():
                if edge.time > 0:
                    cost += edge.time * edge.flow

        end=timer()
        print("Minimum cost of succesiveShortestPathAlgorithm: " + str(cost))
        print("Run time of succesiveShortestPathAlgorithm is: "+ str(end - start))
        return end-start
```

Break out the variables in the function:

- **start/end:** Variables that are used to calculate the time run.
- **path:** A list contains the nodes' indexes dedicated to the shortest path.
- **minCapacity:** A variable used to determine the minimum residual capacity in edges list.
- **flow:** The flow of an edge.
- **cost:** The result optimal cost.

The algorithm is enclosed in a while loop that continues until the dummy source and the dummy sink nodes are balanced (i.e., their net flow is zero). In each iteration, it calls the findPath method to determine the shortest path in the residual network from the dummy source to the dummy sink.

It then calculates the minimum residual capacity along the found path. After that find out the flow based on the minimum value between source balance, sink balance and the minimum residual capacity calculated before.

After that, the algorithm updates the flow along the path, reduces the time values, adjusts the balances of the dummy source and the sink nodes accordingly, and also updates the residual network.

After the loop completes, it calculates the total cost of the flow by summing the products of edge (just consider the edge which exists in the initial network) times and flow. It also measures and prints the runtime of the algorithm. We also return the run time to verify the efficiency.

simplexAlgorithm: The function is used to solve the problem by the simplex algorithm (compare the efficiency).

```
def simplexAlgorithm(self):
    G = nx.DiGraph()
    for node in self.nodes.values():
        if(node.balance !=0):
            G.add_node(node.nodeID, demand=-node.balance) #positive balance is sink in networkX
            ↪  and reversely
        else:
            G.add_node(node.nodeID)
        for edge in node.edges.values():
            G.add_edge(node.nodeID, edge.desNode.nodeID, weight= edge.time, capacity=
            ↪  edge.capacity)

    # graph_pos = nx.spectral_layout(G)
    # nx.draw(G, graph_pos, with_labels = True, font_color = 'white', node_shape='s')

    start=timer()
    flowCost, flowDict = nx.network_simplex(G)
    end=timer()

    print('Minimum cost of simplex Algorithm:', flowCost)
        for key_i, inner_dict in flowDict.items():
            for key_j, inner_val in inner_dict.items():
                print(f'{key_i}->{key_j} \t Flow: {inner_val}')

    print("Run time of simplex algorithm is: "+ str(end - start))
    return end-start
```

Break out the variables in the function:

- **G:** a network to pass to the library.
- **start/end:** Variables that are used to calculate the run time.
- **flowCost:** The optimal cost of the problem.
- **flowDict:** The flow of each edge after solving.

4.2.4.d Result

We create a new network which is the same as the input network, but we just modify it to fit with the library. Then we solve the network, calculate run time and print the flow. We also return the run time as well. This is the source code to solve the problem.

```
myGraph=Graph()
myGraph.inputDataFromExcel("/content/drive/MyDrive/MM/mm_1.xlsx")
if(myGraph.checkMassBalanceConstraint()==0):
    print("The problem has no solution because of violating mass balance constraint\n")
else:
    myGraph.simplexAlgorithm()
    myGraph.addDummySource()
    myGraph.addDummySink()
    myGraph.succesiveShortestPathAlgorithm()
```

The output of the above program will be:

```
Minimum cost of simplex Algorithm: 26
Run time of simplex algorithm is: 0.00037887000007685856
1->2          Flow: 1
1->3          Flow: 3
2->3          Flow: 0
2->4          Flow: 1
3->4          Flow: 3
Minimum cost of succesiveShortestPathAlgorithm: 26
Run time of succesiveShortestPathAlgorithm is: 0.0002111730000251555
1->2          Flow: 1
1->3          Flow: 3
2->3          Flow: 0
2->4          Flow: 1
3->4          Flow: 3
```

As we can see the solution of 2 algorithms is the same.

4.2.5 Verifying Efficiency Of Successive Shortest Path Algorithm

4.2.5.a Theoretical complexity

Time complexity:

Consider the following functions that are called in the Successive Shortest Path algorithm:

Dijkstra Algorithm Call (Inside findPath()): Each iteration of Dijkstra has a time complexity of $O((V + E) \times log(V))$, where V is the number of vertices and E is the number of edges. In the worst case, this is performed once for each node, resulting in $O(V \times (V + E) \times log(V))$.

In the worst case:

- The Dijkstra algorithm is called for each node, resulting in $O(V)$ iterations.
- In each iteration of Dijkstra, the time complexity is $O((V + E) \times log(V))$.
- Therefore, the worst-case time complexity is $O(V \times (V + E) \times log(V))$.

Updating Flow and Residual Capacities (Inside updateFlow()): Updating flow and residual capacities also has a time complexity of $O(P)$, where P is the length of the path.

Updating Reduce Time (Inside updateReduceTime()): The nested loops iterate over all nodes and edges, contributing $O(V \times E)$, where V is the number of vertices and E is the number of edges.

Updating Balances: Updating balances has a time complexity of $O(1)$ per node.

Updating Residual Network (Inside updateResidualNetwork()): The loop iterates over the path, and for each pair of nodes, it may add an edge. In the worst case, this operation has a time complexity of $O(P \times E)$, where P is the length of the path and E is the number of edges.

Total Time Complexity: The overall time complexity is dominated by the Dijkstra algorithm, making it $O(V \times (V + E) \times log(V))$.

Space complexity:

For the space complexity of the algorithm, we will consider the following:

Dijkstra Algorithm (Inside findPath()): The space complexity of the function will depend mainly on the data structured used. The space complexity of this function will be $O(V)$.

- Priority Queue: $O(V)$. In the worst case, which means that initial vertex has edges to all vertexes, the space complexity is also $O(V)$.
- List data structure: $O(V)$.

Other functions: The space complexity for other functions used in the SSP algorithm is $O(1)$, as it involves a few variables within the loop.

Total Space Complexity: The overall space complexity is determined by the Dijkstra algorithm, resulting in $O(V)$.

4.2.5.b Practical efficiency

We do not have the tool to check space complexity, so we just consider real run time in this section.

Firtsly, we just generate 10 small networks **4 nodes and 5 links**. As we can see from the 2 figures below, the run time of the Successive Shortest Path algorithm is better. The average run time of our algorithm is about $0.00016(s)$, while the counterpart is approximately $0.00043(s)$.

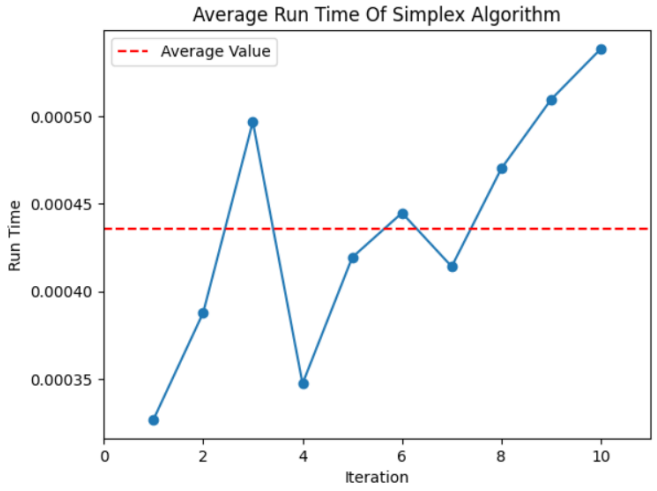


Figure 24: Run time of simplex algorithm

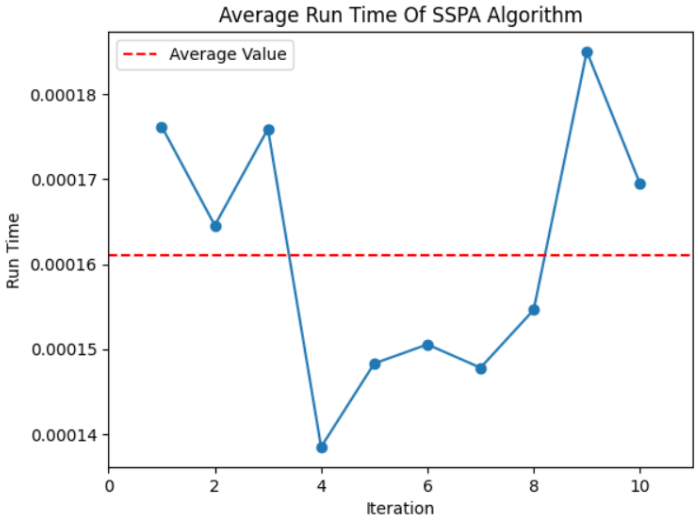


Figure 25: Run time of SSP algorithm

Next, we generate 10 networks with **20 nodes and 64 links** to check the run time of 2 al-

gorithms. Now, the run time of the simplex algorithm is better. The average run time of our algorithm is about $0.009(s)$, while the counterpart is approximately $0.0049(s)$.

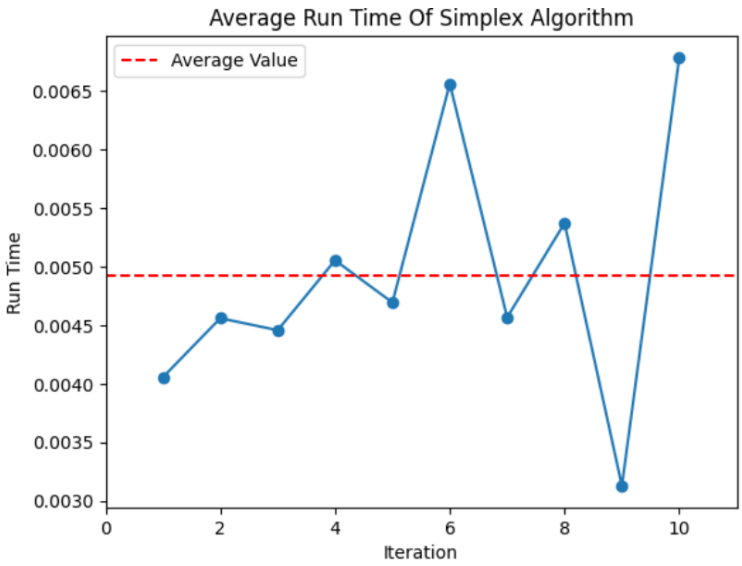


Figure 26: Run time of simplex algorithm

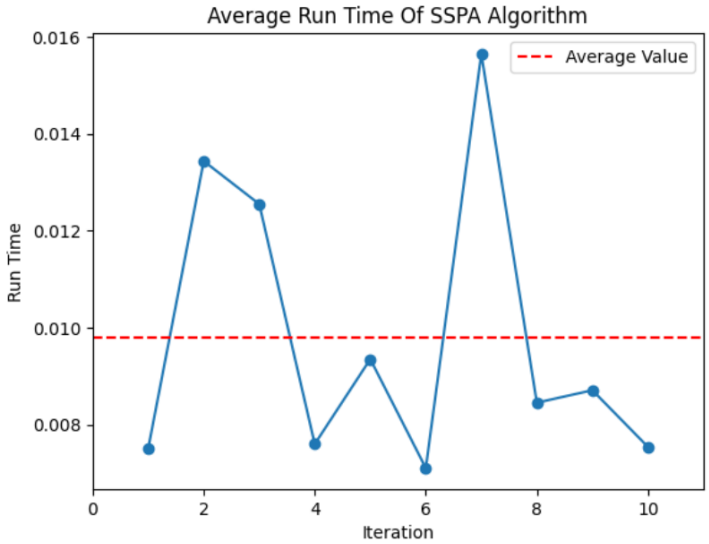


Figure 27: Run time of SSP algorithm

We generate 10 bigger networks **52 nodes and 192 links**. As we can see from the 2 figures below, the run time of the simplex algorithm is much better. The average run time of our algo-

rithm is about $0.076(s)$, while the counterpart is approximately $0.0182(s)$.

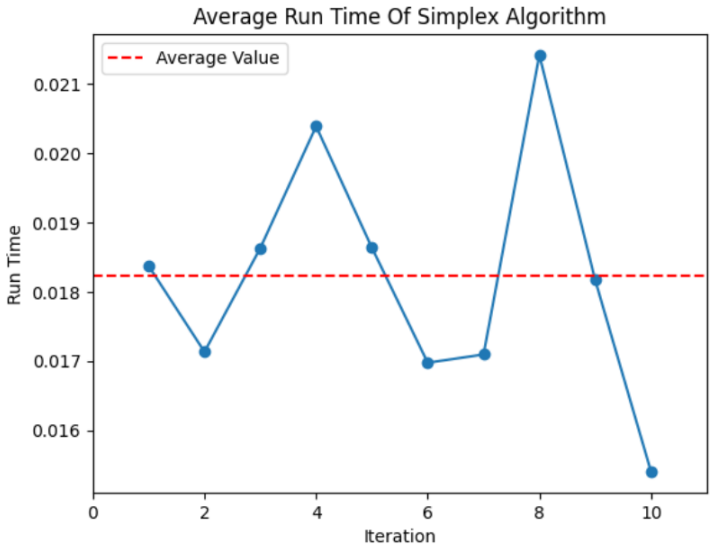


Figure 28: Run time of simplex algorithm

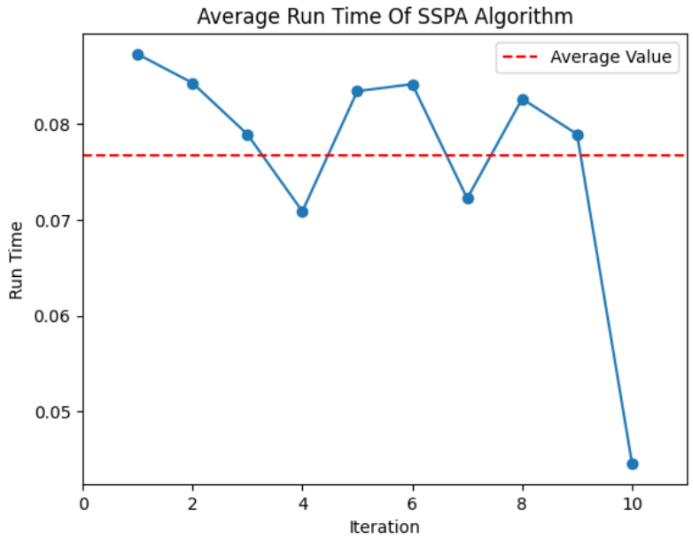


Figure 29: Run time of SSP algorithm

In summary, it seems that with small networks, the Successive Shortest Path algorithm is a good choice. Nevertheless, we should use the Simplex algorithm to solve bigger networks. And the bigger the network is, the more significant the run time gap between the 2 algorithms is. However, the Successive Shortest Path algorithm is implemented by ourselves, so it can be unoptimized. Therefore, our evaluation is just for reference only.

5 Summary

All in all, throughout a long period of time, our team has carried out and finished the assignment with all-out efforts among team members. We have got a chance to have a deeper insight into such an optimal and useful modeling approach, Two-Stage Stochastic Linear Programming. The methodology offers us an ability to incorporate uncertainty into decision-making models, which is obviously crucial in situations where future parameters, such as demand or costs, are not known with certainty. This also paves the way for us to optimize resource allocation in the face of uncertainty, helping organizations allocate resources in a more efficient way and taking into account potential variations in demand, costs, or other critical factors. Additionally, we figure the way how it is utilized in reality and could build the models in order to find the most optimal solutions that balance the costs and benefits associated with first-stage and second-stage decisions that helps us make decisions that are not only robust but also cost-effective.

What is more, several concepts have been worked out, including the Minimum-cost Flow Problem, its Successive Shortest Path Algorithm, which have provided us a great deal of useful expertise in how to implement and verify its efficiency in the most efficient manner.

Please kindly check our source code in the following link shown below:

- Problem 1
- Problem 2

Last but not least, we would like to show our sincere gratitude to Dr. Nguyen Tien Thinh for your wholehearted lessons as well as your advice throughout the time we conducted the project and the whole semester. If you have any further questions, please contact us via this email: nam.nguyenolkmphy@hcmut.edu.vn

References

[1] Alexander Shapiro & Darinka Dentcheva & Andrzej Ruszczynski, *Lectures on Stochastic Programming: Modeling and Theory.*

[2] Towards Data Science, *Solving Two-Stage Stochastic Programs in Gurobi.* https://towardsdatascience.com/solving-two-stage-stochastic-programs-in-gurobi-9372da1e3ba8

[3] Gurobi Optimization, *Solving Simple Stochastic Optimization Problems with Gurobi.* https://www.gurobi.com/events/solving-simple-stochastic-optimization-problems-with-gurobi/

[4] Topcoder, *MINIMUM COST FLOW PART ONE: KEY CONCEPTS.* https://www.topcoder.com/thrive/articles/Minimum%20Cost%20Flow%20Part%20One:%20Key%20Concepts

[5] Mircea Parpalea, *Interactive tool for the successive shortest paths algorithm in solving the minimum cost flow problem*, 2009.

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