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

LECTURER: NGUYỄN TIẾN THỊNH.

Student 1:	Vũ Hoàng Tùng	2252886
Student 2:	Huỳnh Văn Anh Hoàng	2252229
Student 3:	Ngô Chấn Phong	2053321
Student 4:	Trần Xuân Bách	2252058
Student 5:	Phạm Đoàn Bảo Trung	2252858

Ho Chi Minh City, December 2023

Contents

1 Members and Workload

NAME	SPECIFIC TASK	CONTRIBUTION
Phong	Investigate problem 1 Develop software and modify report	100
Hoàng	Investigate problem 1 Develop software	100
Bách	Solution Approach for problem 2 Explain the model and two subproblem Mention the pseudocode and example	100
Trung	Solution approach for problem 2 Mention the pseudocode, explain and implement algorithm 1	100
Tùng	Solution Approach for problem 2 Verify the effectiveness of the successive shortest path algorithm	100

2 Question 1: Use the 2-SLPWR model to produce n products satisfying production $\geq$ demand.

2.1 Introduction

2.1.1 Definition and concept

In problem 1, we will use the theory of stochastic simulation as a concept to get more understanding about this problem.

A stochastic simulation of a system S is a specific simulation of S where we allow uncertainty or precisely uncertain fluctuation of the operations of a few or all processes and constituents of the system. Going through this, there is a simulation study which is used to evaluate the statistical methods. A simulation study is usually undertaken to determine the value of some quantity θ connected with a particular stochastic model M. The parameter of interest θ of a simulation of the relevant system often results from the output data X, a random variable whose expected value

$$E[X] = \mu$$

The parameter θ could be μ itself or any other parameters of M.

As a result, a discrete-time Markov chain (DTMC) is chosen when it is a sequence of random variables, known as stochastic process, in which the value of the next variable depends only on the value of the current variable, and not any variables in the past. Stochastic process X(t) is Markov if for any $t_1 < t_2 < ..., t_n$, and any sets $A; A_1, A_2, ..., A_n$.

$$P[X(t) \in A | X(t_1) \in A_1, ..., X(t_n) \in A_n] = P[X(t) \in A | X(t_n) \in A_n]$$

2.1.2 Elucidation

Suppose we have a sequence of M of consecutive trials, numbered n=0,1,2,... The outcome of nth trials is represented by the random variable X_n which we assume to be discrete and to take one of the values j in a finite set Q of discrete outcomes $e_1, e_2, e_3, .., e_s$. Sequence M is called a (discrete time) Markov chain if, while occupying Q states at each of the unit time points 1,2,3,...,n-1,n,n+1,... and M satisfies the following property.

$$P[X_n = j | X_{n-1} = i, X_{n-1} = k, ..., X_0 = a] = P[X_n = j | X_{n-1} = i]$$

In each step n-1 to n, the process can either stay at the same state e_i (at both n-1,n) or move to other state e_j (at n) with respect to the memory-less rule, saying the future behavior of system depends only on the present and not on its past history.

A Markovian process n_t or a combination of a Makovian process n_t and an independent process ξ_t is assumed to be the underlying data process in a Markov chain. The programming equations for t=T,...,1 are

$$Q_t(x_{t-1}, \xi_t, n_t) = min_{A_t x_t + B_t z_t + C_t y_t >= b_t} {u_t}^T x_t + {v_t}^T y_t + Q_{t+1}(x_t, n_t)$$

where the expected cost-to-go functions,

$$Q_{t+1}(x_t, n_t) := \begin{cases} E_{|n_t} Q_{t+1}(x_t, \xi_{t+1}, n_{t+1}, T = T-1, ..., 1 \\ 0, t = T \end{cases}$$

2.2 Application

We will apply this problem by using GAMSPY and Python to simulate the whole system.

2.3 Mathematical formulation

2.3.1 Decision variables, Parameters used in mathematical formulation

There are 3 decision variables in this problem and we also have to find the optimal value of these three ones:

$x = \{ x_1, x_2, \ldots, x_m \}$ is the number of parts to be ordered before production

$y = \{ y_1, y_2, \ldots, y_m \}$ is the number of parts left in inventory

$z = \{ z_1, z_2, \ldots, z_n \}$ is the number of units produced

, with m and n is in turn the number of 3-rd suppliers and the number of production locations.

We also have some parameters available for calculating the optimal decision variables:

$b = \{ b_1, b_2, \ldots, b_n \}$ is pre-order cost b_j per unit of part j (before the demand is known).

$l = \{ l_1, l_2, \ldots, l_n \}$ is the additional cost to satisfy a unit of demand for product i.

$q = \{ q_1, q_2, \ldots, q_n \}$ is the unit selling price of the product i cost additionally d_i to satisfy a unit of demand.

$s = \{ s_1, s_2, \ldots, s_n \}$ is the salvage value for the parts are not use.

$d = \{ d_1, d_2, \ldots, d_n \}$ is the random demand vector with the probability of each elements follows the binomial distribution Bin(10, $\frac{1}{2}$).

$S = \{ S_1, S_2 \}$ is the number of scenarios with the density $p_S = \frac{1}{2}$.

2.3.2 Model

The second-stage problem:

For an observed value (a realization) $d = (d_1, d_2, \ldots, d_n)$ of the above random demand vector D, we can find the best production plan by solving the stochastic linear program (SLP) represented by this following model:

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

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

The whole 2-SLPWR model is based on a popular rule that production $\geq$ demand.

Now follow distribution-based approach, we let $Q(x) := E[Z(z, y)] = E_\omega[x, \omega]$ denote the optimal value of problem (1). The quantities x_j are determined from the following optimization problem:

$$\min \text{ g(x, y, z)} = b^T.x + Q(x) = b^T.x + E(Z(z))$$

where $\text{Q(x)} = E_\omega(Z) = \sum_{i=1}^{n} p_i.c_i.z_i$ is taken as the probability distribution of $\omega = D$

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

2.4 Code and explanation

2.4.1 Install packages and import libraries

First, there is some packages which are not available for solving the problem so we have to install them as this following figure:

```
!pip install uvicorn
!pip install python-multipart
!pip install kaleido
!pip install tiktoken
!pip install openai
!pip install fastapi
!pip install cohere
!pip install gamspy
```

Next, we will import available library, some of which is just installed, for using of elements and attributions to support the calculate the final results:

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

2.4.2 Declaration parameters, variable and constraints of the problem

2.4.2.a Parameters

We input or choose random value of parameters A, b, l, q, s and demand vector d of the problem. Note that, d will have 2 value, since that there are 2 scenario. Moreover, we will print

each value to confirm it is the same as we want. We randomly give numbers of D, B, L, S, Q and use library gp (Gamspy) to define the parameters.

```
n = 8
m = 5
A = []
u = 1
v = 10

for i in range(0, 8, 1):
    temper = np.random.randint(u, v, m)
    A.insert(i, temper)

D = [np.random.binomial(10, 0.5, n), np.random.binomial(10, 0.5, n)]

B = [4500, 2000, 3150, 3800, 7000]
L = [760000, 150000, 190000, 741000, 353000, 450000, 630000, 152000]
S = [1500, 1300, 1850, 2500, 2000]
Q = [900000, 250000, 300000, 800000, 750000, 850000, 900500, 200000]


C= []
for i in range(n):
    C.insert(i, L[i] - Q[i])

print(C)
print(D)
print(L)
print(Q)
print(B)
print(S)

container = gp.Container()
i = gp.Set(container,"i",description="warehouses",
           records=["1", "2", "3", "4", "5", "6", "7", "8"])
j = gp.Set(container,"j",description="suppliers",
           records=["1", "2", "3", "4", "5"])
a = gp.Parameter(container,"a",
                 description="unit of product i requires of part j",domain=[i, j],
                                                    records=np.asanyarray(A
                                                    ))
l = gp.Parameter(container,"l",
                 description="cost to produce at location j",domain=i, records=np.
                                                    asanyarray(L))
q = gp.Parameter(container,"q",
                 description="cost to sell at location i",domain=i, records=np.
                                                    asanyarray(Q))
b = gp.Parameter(container,"b",
                 description="preorder cost of the supplier j",domain=j, records=
                                                    np.asanyarray(B))
c = gp.Parameter(container,"c",
                 description="cost coefficients",domain=i, records=np.asanyarray(C
                                                    ))
s = gp.Parameter(container,"s",
                 description="salvage values in the supplier j",domain=j, records=
                                                    np.asanyarray(S))
d1 = gp.Parameter(container,"d1",
                  description="demand at first scenario",domain=i, records=np.
                                                     asanyarray(D[0]))
d2 = gp.Parameter(container,"d2",
                  description="demand at second scenaario",domain=i, records=np.
                                                     asanyarray(D[1]))
```

```
d1.records
d2.records
```

2.4.2.b Variables

```
x = gp.Variable(container, "x", domain=j, type="positive",
                description="numbers of portions have been ordered ")
y1 = gp.Variable(container, "y1", domain=j, type="positive",
                 description="numbers of portions have been left in inventory in
                                                    day one")
y2 = gp.Variable(container, "y2", domain=j, type="positive",
                 description="numbers of portions have been left in inventory in
                                                    day two")
z1 = gp.Variable(container, "z1", domain=i, type="positive",
                 description="number of portions need to be produced in day one")
z2 = gp.Variable(container, "z2", domain=i, type="positive",
                 description="number of portions need to be produced in day two")
```

2.4.2.c Constraints

In this step, we will set the constraints based on the Equation methods of GAMSPY by using the constraints and equation in the second-stage problem. It can be seen that, there are 2 scenarios, therefore, the constraints also have 2 situation, the former(demand1, constraint1) for the first scenario and the latter(demand2, constraint2) for the second one.

```
demand1 = Equation(container=container, name="demand1", domain=i,
    definition= z1[i] <= d1[i])

demand2 = Equation(container=container, name="demand2", domain=i,
    definition = z2[i] <= d2[i])

constraint1 = Equation(container=container, name="constraint1", domain=j,
    definition = y1[j] == x[j] - Sum(i,a[i,j]*z1[i]))

constraint2 = Equation(container=container, name="constraint2", domain=j,
    definition = y2[j] == x[j] - Sum(i,a[i,j]*z2[i]))
```

2.4.2.d Solving

Finally, after have enough information about the parameter, variable and constraints of the problem, we will use the available method which is LP (Linear Programming) of GAMSPY to solve the problem completely.

```
obj_func = Sum(j, b[j] * x[j]) + 1/2 * (Sum(i, c[i] * z1[i])
- Sum(j, s[j] * y1[j])) + 1/2 * (Sum(i, c[i] * z2[i]) - Sum(j, s[j] * y2[j]))

problem = gp.Model(container, "problem",  "LP",
container.getEquations(), Sense.MIN, obj_func)
```

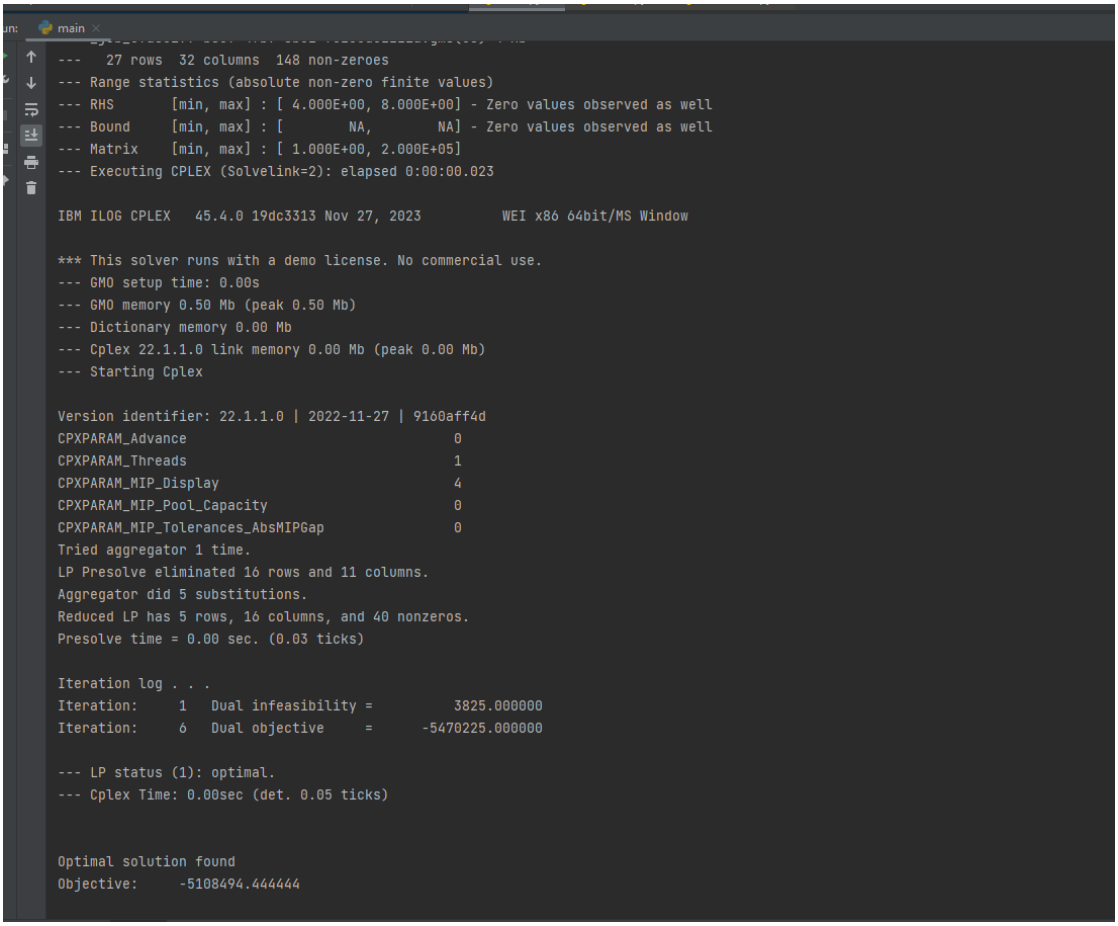
```
problem.solve(output=sys.stdout)
```

2.4.3 Implemented and results

At first it will run to display the D,B,L,Q,S and it will execute on GAMS, as can be seen, there are some paths displayed on our computer

```
C:\Users\phong\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\phong\PycharmProjects\pythonProject1\main.py
[array([6, 7, 7, 5, 4, 4, 4, 5]), array([5, 6, 8, 6, 7, 8, 6, 4])]
[760000, 150000, 190000, 741000, 353000, 450000, 630000, 152000]
[900000, 250000, 300000, 800000, 750000, 850000, 900500, 200000]
[4500, 2000, 3150, 3800, 7000]
[1500, 1300, 1850, 2500, 2000]
[-140000, -100000, -110000, -59000, -397000, -400000, -270500, -48000]
--- Job _job_e9a80199-b8e9-4fb7-8b52-fe26cde2221a.gms Start 12/14/23 23:52:40 45.4.0 19dc3313 WEX-WEI x86 64bit/MS Windows
--- Applying:
    C:\Users\phong\PycharmProjects\pythonProject1\venv\Lib\site-packages\gamspy_base\gmsprmNT.txt
--- GAMS Parameters defined
    LP CPLEX
    MIP CPLEX
    RMIP CPLEX
    NLP CONOPT
    MCP PATH
    MPEC NLPEC
    RMPEC CONVERT
    CNS CONOPT
    DNLP CONOPT
    RMINLP CONOPT
    MINLP SBB
    QCP CONOPT
    MIQCP SBB
    RMIQCP CONOPT
    EMP CONVERT
    Restart C:\Users\phong\AppData\Local\Temp\tmpjósxlkx7\_save_df0033fb-0732-4361-86f9-76ccdc426e76.g00
    Input C:\Users\phong\AppData\Local\Temp\tmpjósxlkx7\_job_e9a80199-b8e9-4fb7-8b52-fe26cde2221a.gms
    Output C:\Users\phong\AppData\Local\Temp\tmpjósxlkx7\_job_e9a80199-b8e9-4fb7-8b52-fe26cde2221a.lst
    Save C:\Users\phong\AppData\Local\Temp\tmpjósxlkx7\_restart_df0033fb-0732-4361-86f9-76ccdc426e76.g00
    ScrDir C:\Users\phong\AppData\Local\Temp\tmpjósxlkx7\225a\
    SysDir C:\Users\phong\PycharmProjects\pythonProject1\venv\Lib\site-packages\gamspy_base\
    CurDir C:\Users\phong\AppData\Local\Temp\tmpjósxlkx7\
    LogOption 3
    LogFile C:\Users\phong\AppData\Local\Temp\tmpjósxlkx7\_job_e9a80199-b8e9-4fb7-8b52-fe26cde2221a.log
    Trace C:\Users\phong\AppData\Local\Temp\tmpjósxlkx7\trace.txt
    TraceOpt 3
```

After that, it will solve based on the equation, we will have the optimal solution with the objective answers at last line.



3 Question 2: Stochastic Linear Program for Evacuation Planning In Disaster Responses

3.1 Introduction

3.1.1 Definition and concept

Two-stage stochastic programming is a mathematical optimization framework for solving decision-making problems under uncertainty. It is a type of stochastic programming, which deals with problems where some or all of the problem parameters are uncertain. In two-stage stochastic programming, the uncertainty is captured through a set of possible scenarios, each with a given probability. The decision-making process is divided into two stages:

First stage: In the first stage, the decision-maker makes decisions without knowing the exact values of the uncertain parameters. These decisions, known as first-stage decisions, are typically related to long-term planning and resource allocation.

Second stage: After the first-stage decisions have been implemented, the actual values of the uncertain parameters are revealed. The decision-maker then makes additional decisions, called

second-stage decisions, to adapt to the realized scenario. These decisions are typically related to real-time operations and resource adjustments.

The goal of two-stage stochastic programming is to find a set of first-stage decisions that minimize the expected cost or maximize the expected utility over all possible realizations of the uncertain parameters.

3.1.2 Application in disaster evacuation

Following the research of Li Wang, the earthquake will be taken into account to represent a two-stage stochastic programming problem. The first stage, or pre-event response period, is the pre-earthquake stage, where we are being forecasted about the disaster and details about the optimal evacuation plan will be given to minimize casualties in the shortest amount of time. In the second stage, or post-event period, after the realization of the unknown information is known, a real-time decision will be made to form an optimal plan by considering different scenarios.

3.2 Mathematical formulation

3.2.1 Decision variables, Parameters used in mathematical formulation

Symbol	*Definition*
V	*the set of nodes*
A	*the set of links*
i, j	*the index of nodes,* $i,\ j \in V$
(i, j)	*the index of directed links,* $(i, j) \in A$
s	*the index of scenario*
S	*the total number of scenarios*
v	*the supply value of source node*
$\widetilde{T}$	*the time threshold*
T	*the total number of time intervals*
u_{ij}	*the capacity on physical link* (i, j)
$u_{ij}^s(t)$	*the capacity of link* (i, j) *in scenario* s *at time* t
$c_{ij}^s(t)$	*the travel time of link* (i, j) *in scenario* s *at time* t
μ_s	*the probability in scenario* s

Decision Variables Used in Mathematical Formulation.

Decision Variable	*Definition*
x_{ij}	*the flow in link* (i, j)
$y_{ij}^s(t)$	*the flow in link* (i, j) *in scenario* s *at time* t

3.2.2 Model

$$\begin{cases} \min \sum_{(i,j)\in A} p_{ij}x_{ij} + \sum_{s=1}^{S} \left(\mu \cdot \sum_{(i,j)\in A_s} c_{ij}^s(t) \cdot y_{ij}^s(t)\right) + \sum_{s=1}^{S} \sum_{t\leq \widetilde{T}} \sum_{(i,j)\in A} \alpha_{ij}^s(t)(y_{ij}^s(t) - x_{ij}) \\ s.t. \\ \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ij} = d_i, \forall i \in V \\ 0 \leq x_{ij} \leq u_{ij}, \forall (i,j) \in A \\ s.t. \\ \sum_{(i_t,j_t)\in A_s} y_{ij}^s(t) - \sum_{(j_t,i_t)\in A_s} y_{ij}^s(t) = d_i^s(t), \forall i \in V,\ t \in \{0,1,...,T\},\ s \in \{1,2,...,S\} \\ 0 \leq y_{ij}^s(t) \leq \mu_{ij}^s(t), \forall (i,j) \in A,\ t \in \{0,1,...,T\},\ s \in \{1,2,...,S\} \end{cases}$$

The objective function is that minimize the $p_{ij}x_{ij}$ which mean eliminate loops on the generated physical evacuation path with p_{ij} is a constant, $\forall (i,j) \in A$, concurrently minimize the expected overall evacuation time of each scenario's adaptive evacuation plan and the probability of the corresponding scenario occurring.

Formula $\sum_{(i,j)} p_{ij}x_{ij} \sum_{t\leq \widetilde{T}} \sum_{(i,j)\in A} \alpha_{ij}^s(t)(y_{ij}^s - x_{ij})$ is the relaxed form of constraint $\sum_{t\leq \widetilde{T}} y_{ij}^s(t) = x_{ij}, (i,j) \in A, s = 1,2,...S$ is that minimize the gap between local maxima and minima which means decrease the difference between the flow on link (i, j) and that link in scenario s at time t.

Constraint $\sum_{(i,j)\in A_S} x_{ij} - \sum_{(j,i)\in A_S} x_{ij} = d_i, \forall i \in V$, the balance of the flow (i, j)

$$d_i = \begin{cases} \text{v, i = s} \\ \text{-v, i = t} \\ \text{0 otherwise} \end{cases}$$

the flow on each link must satisfy the flow balance which equal the supply value of source node when i is super-source, negative when i is super-sink and equal zero with other case.

Constraint $0 \leq x_{ij} \leq u_{ij}, \forall (i,j) \in A$ mean the flow on link (i , j) must be smaller than other equal to the capacity of that link and be greater than or equal zero.
Similarly with the previous the two last constraint is the same but in time t and scenario s with $t \in \{0,1,...,T\}$ and $s \in \{1,2,...,S\}$.

3.3 SubProblem 1

$$\begin{cases} minSP1(\alpha) = \sum_{(i,j)\in A}(P_{ij} - \sum_{s=1}^{S} \sum_{t\leq \widetilde{T}} \alpha_{ij}^s(t))x_{ij} \\ s.t. \\ \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ij} = d_i, \forall i \in V \\ 0 \leq x_{ij} \leq u_{ij}, \forall (i,j) \in A \end{cases}$$

The the objective function is trying to find the solution that minimizes the cost of violating the constraints, while still ensuring that the actual cost does not exceed the target cost, by using lagrangien multiplier to relaxed the constraint into the objective function $\sum_{s=1}^{S} \sum_{t\leq \widetilde{T}} \alpha_{ij}^s(t))x_{ij}$ is used to represent the cost of violating the constraints. p_{ij} is to ensure that the solution obtained must be smaller than the target value to reduce the loop path. The two constraints are that $\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ij} = d_i, \forall i \in V$ the flow balance must be achieved simultaneously $0 \leq x_{ij} \leq u_{ij}, \forall (i,j) \in A$ the flow must be less than the capacity.Therefore, the sub-Problem 1 can be solved by the successive shortest path algorithm which is represent in **4.3**.

3.4 SubProblem 2

$$\begin{cases} minSP2(\alpha) = \sum_{s=1}^{S} \sum_{(i,j)\in A} \left(\sum_{t\in\{0,1,...,T\}} \mu_s.c_{ij}^s(t) + \sum_{t\leq\widetilde{T}} \alpha_{ij}^s(t) \right) y_{ij}^s(t) \\ s.t. \\ \sum_{(i_t,j_t)\in A_s} y_{ij}^s(t) - \sum_{(j_t,i_t)\in A_s} y_{ij}^s(t) = d_i^s(t), \forall i \in V,\ t \in \{0,1,...,T\},\ s \in \{1,2,...,S\} \\ 0 \leq y_{ij}^s(t) \leq \mu_{ij}^s(t), \forall (i,j) \in A,\ t \in \{0,1,...,T\},\ s \in \{1,2,...,S\} \end{cases}$$

For each scenario s $\in \{1, 2, \ldots, S\}$ subproblem 2 has a similar structure as subproblem 1 with time-dependent link cost $c_{ij}^s(t)$ and link capacity $u_{ij}^s(t)$ and the generalized cost $g_{ij}^s(t)$. Since subproblem 2 is a time-dependent min-cost flow problem so it can be achieve by Modified Successive Shortest Path Algorithm for Time-Dependent Min-Cost Flow Problem or the modified label-correcting algorithm (Ziliaskopoulos Mahmassani, 1992) will be adopted to find the time-dependent min- cost path in the residual network, which has been mentioned in **3.3.2**.

3.5 Solution approaches

In order to solve the model of the above section, the model is then be decomposed into 2 subproblems: the min-cost flow problem and the time - dependent ones. The algorithm which is used to resolve min - cost flow will be explained further in 2 subsections.

3.5.1 Successive shortest path algorithm for min-cost flow problem.

First, let's consider some components that constitute the resolutions:
Network flow: A directed graph G with vertexes V and edges E. On each edge (i, j) ∀ i , j ∈ V there exists flow x_{ij} which satisfies:

$$x_{ij} \leq u_{ij}$$

And finally let the c_{ij} be the cost on that flow; at the beginning, all flows are 0. The network is modified as follows: for every edge (i, j) we add a reverse edge (j, i) to the network where the capacity u_{ij} and cost are 0 and $c_{ji} = -c_{ij}$, respectively. We still have a network that is not a multigraph because, in accordance with our constraints, the edge (j, i) was not in the network previously (graph with multiple edges). Furthermore, the condition $x_{ji} = -x_{ij}$ will always be true throughout the algorithm's phases.
Residual Graph: residual graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called ***residual capacity*** which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge. The residual graph contains only unsaturated edges (edges in which $x_{ij} \leq u_{ij}$)

To solve the first subproblem, the algorithms used to solve the maximum - flow problem and minimum - cost flow are applied. The framework of this resolution can be divided into 3 steps:
Step 1: Find a path which has minimum travel time using a shortest path algorithm
Step 2: Find the maximum number of people that can go on that path by applying a maximum flow algorithm
Step 3: Update the total flow of the graph

3.5.1.a Dijkstra's shortest path algorithm - Searching for minimum travel time path

This algorithm will try to find the shortest travel time of all possible path from supersource to supersink. By relaxing and updating labels, an optimal path can be constituted. The instructions of this algorithm will be presented as below:

Step 1: Set every node as unvisited. Make an unvisited set, which is the collection of all the unvisited nodes.

Step 2: Assign a preliminary cost value to each node: zero for our starting node and infinite for all others. The tentative cost of a node v during the algorithm run is the length of the shortest path identified so far between the node v and the beginning node.

Step 3: Consider all of the current node's unvisited neighbors and determine their approximate distances through the current node. Comparing the newly determined tentative cost to the one currently given to the neighbor, if the comparison satisfies:

$$\begin{cases} \mathrm{u}_{ij} - x_{ij} > 0 \text{ is the residual capacity} \\ \mathrm{d}_j > d_i + c_{ij} \text{ with } d_j, d_i \text{ are the tentative values of nodes j and i} \end{cases}$$

Then the neighbor nodes will be relaxed.

Step 4: Mark the current node as visited and delete it from the unvisited list once we have finished examining all of its unvisited neighbors. Never again will a visited node investigated.

Step 5: If the destination node has been marked visited (when designing a route between two specified nodes) or if the least tentative cost among the nodes in the unvisited set is infinite (this happens when there is no link between the starting node and the remaining unvisited nodes when planning a full traverse) then stops the algorithm. Otherwise, select the unvisited node that is marked with the smallest tentative cost, set it as the new current node, and go back to step 3.

```
Function Dijkstra(Graph, source)
for each vertex v in Graph.Vertices: do
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
    add v to Q
end
dist[source] ← 0
while Q is not empty: do
    u ← vertex in Q with min dist[u]
    remove u from Q
    for each neighbor v of u still in Q: do
        alt ← dist[u] + Graph.Edges(u, v)
        if alt < dist[v]: then
            dist[v] ← alt
            prev[v] ← u
        end
    end
end
return dist[], prev[]
```

Algorithm 1: Dijkstra(Graph, source):

3.5.1.b Ford-Fulkerson's algorithm Optimizing number of people evacuating on the path

Once the minimum cost path has been created, Ford-Fulkerson's method will complete the rest of Algorithm 1.

This algorithm will utilize the residual graph as aforementioned and optimize the flow by a new definition:

Augmenting path: An augmenting path is a simple path in the residual graph, along the edges whose residual capacity is positive. If such a path is found, then that path is increased by the following way: x_{ij} += ξ and x_{ji} −= ξ with ξ is the smallest residual capacity of the minimum path. Once an augmenting process doesn't exist anymore, the path achieves maximal flow.

We keep on searching minimum travelling time roads and apply augmenting until a more optimal path cannot be found. At that point, the problem will be solved.

Data: G: a directed connected graph, s: source node, t: sink node,c: residual capacity.
Function Ford-Fulkerson(G, s, t)
for ***each*** *edge(u, v)* $\in$ *E(G)* **do**
  f (u, v) ←0
  f (v, u) ← 0
end
while $\exists$ *an augmenting path* $p \in G_f$ **do**
  $c_f(p)$ ← min $c_f(u, v)$: (u,v) $\in$ p
  for ***each*** *edge(u, v)* $\in$ p **do**
    f (u, v) ← f (u, v) + $c_f(p)$
    f (v, u) ← -f (u, v)
  end
end

Algorithm 2: Ford-Fulkerson(Graph, source):

3.5.1.c Fast successive shortest path algorithm

After studying 2 above algorithms, we end up with a combination of them.
The pseudocode is presented as following:

Data: G: a directed connected graph, s: source node, t: sink node,c: residual capacity.
Function Fast Succesive Shortest Path($V, E, c, b, \$$)
for ***each*** *edge* $e \in E$ **do**
  $\psi(e) \leftarrow 0$
end
for ***each*** *vertex* $v \in V$ **do**
  $\pi(e) \leftarrow 0$
end
$B \leftarrow \sum_v |b(v)|/2$
while $B > 0$ **do**
  construct G_ψ
  s $\leftarrow$ any vertex with $b_\psi(s) < 0$
  t $\leftarrow$ any vertex with $b_\psi(t) > 0$
  dist $\leftarrow$ **Dijkstra(G_ψ, s)**
  $\sigma \leftarrow$ shortest path in G_ψ from s to t
  Augment(s,t, σ)
  for ***each*** *vertex* $v \in V$ **do**
    $\pi(v) \leftarrow$ dist(v)
  end
end

Algorithm 3: Fast Succesive Shortest Path(Graph, source):

3.5.2 Modified Successive Shortest Path Algorithm for Time-Dependent Min-Cost Flow Problem

Algorithm

Step 0.Initialization

Initialize the node labels :

$\lambda_i(t) = \infty \ \forall\, i \in \vartheta \backslash N,\ t \in \varphi$

$\pi i(t) = \infty \ \forall\, i \in \vartheta \backslash N,\ t \in \varphi$

$\lambda_N(t) = 0 \ \forall\, t \in \varphi$

Initialize the scan-eligible list:

Create the scan-eligible list, SE, and insert N.

Step 1. Choose Current Node

if *SE list is empty* **then**

  Go to step 3

end

else

  Select the first node from the SE list.

  Call this node the current node, j.

end

Step 2. Update the Node Labels

for ***each*** $i \in \Gamma^{-1}(j), (i.e., \forall i | (i,j) \in \ \mathcal{A})$ **do**

  Update the vector $[\lambda_i(t)]_{t \in \varphi}$

  for ***each*** $t \in \varphi$ **do**

    Calculate

$$\eta_t = \sum_{p=1}^{k_{i,j}(t)} [(\tau_{ij}^p(t) + (\lambda_j(t + \tau_{ij}^p(t))) \bullet \rho_{ij}^p(t)]$$

    where p is the set of indices of possible travel times on arc (i, j) at time t.

    if $\rho_i(t) < \lambda_i(t)$ **then**

      $\lambda_i = \rho_i(t)(\pi_i(t) = (i,j))$

    end

    if $i \notin SE$ **then**

      Put i in SE list.

    end

    if $Alli \in \Gamma^{-1}$ *have been considered* **then**

      Go to step 1.

    end

    Step 3. Stop

      The algorithm terminates with a lower bound on the expected time of the a priori LET paths,$\forall t \in \varphi$, from every node $i \in \vartheta$ to the destination node N.

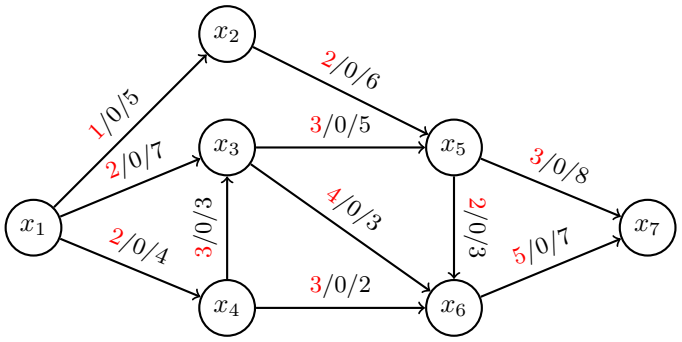
  end

end

Algorithm 4: ELB algorithm : a specialized modified label-correcting algorithm

3.5.3 Example of solving the two-stage stochastic evacuation planning model

[INITIAL STATE]

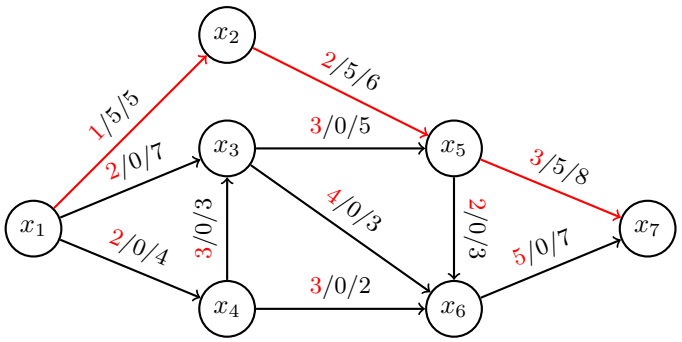


Vertex	x1	x2	x3	x4	x5	x6	x7	Parent	Queue
0	0	∞	∞	∞	∞	∞	∞	X	x1

In this example, we investigate the implementation of successive shortest path in a small network with 7 nodes and 11 links. x_1 will be the source and x_7 will be the sink of the graph. The table and graph will be updated and an explanation added at each iteration. The minimum path will be colored with red.

At the initialization step, we set all the flows to 0. The initial label of all vertexes are ∞, except for the first vertex x_1 is 0. The labels on each edge of the graph are in the form: **Cost per unit of flow / Flow on the edge / Capacity**. The first row of the table will represent all vertexes of the graph and the first column are initial vertexes which is pop from the queue in each iteration of Dijkstra's algorithm. Other cells in the table are the cost labels of each vertex. Parent column will be utilize to keep track of the minimum cost path when the Dijkstra's algorithm is terminated. The queue in first stage will contain only the vertex x_1.

ITERATION 1



Vertex	x1	x2	x3	x4	x5	x6	x7	Parent	Queue
0	0	∞	∞	∞	∞	∞	∞	X	1
x1	0	1	2	2	∞	∞	∞	X	x2 x3 x4
x2	0	1	2	2	3	∞	∞	1	x3 x4 x5
x3	0	1	2	2	3	6	∞	1	x4 x5 x6
x4	0	1	2	2	3	5	∞	1	x5 x6
x5	0	1	2	2	3	3	6	2	x6 x7
x6	0	1	2	2	3	3	6	4	x7
x7	0	1	2	2	3	5	6	5	

At the first iteration, the function Dijkstra will be called to find the minimum cost path.

Dijkstra procedure:

First, x_1 is pop from the queue and check for all possible paths from this vertex, if the adjacent vertexes satisfy:

$$\begin{cases} \mathrm{u}_{ij} - x_{ij} > 0 \text{ is the residual capacity} \\ \mathrm{d}_j > d_i + c_{ij} \text{ with } d_j, d_i \text{ are the tentative values of nodes j and i} \end{cases}$$

Those vertexes will be push into the queue. As we can observe, x_2, x_3 and x_4 are satisfied so they will be in the queue. Then, the label of that vertex will be updated with function: label of previous vertex + the cost on that edge so the labels of x_2, x_3 and x_4: 1 , 2 , 2 respectively. After that, the table will update parent's of x_2, x_3 and x_4 as x_1

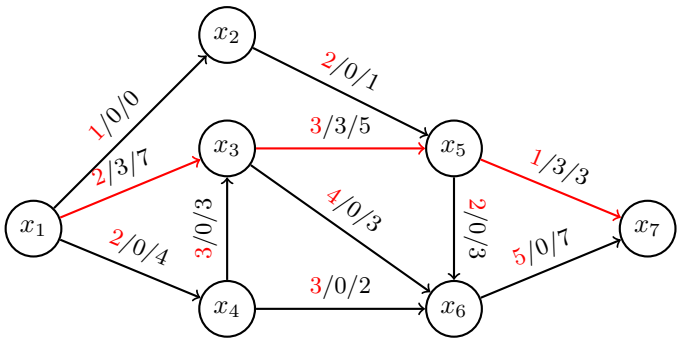
Second, x_2 is pop from the queue and the vertex adjacent with x_2 is x_5 also satisfy the condition, so label and parent of x_5 will be changed into 3 and x_2.

Similarly in the following steps in the table, we end up with the minimum cost from x_1 to x_7 is 6 and the path is x_1, x_2, x_5, x_7.

Augmenting procedure:

That minimum path will be updated with Ford-Fulkerson's method above. As we can see, the smallest residual capacity of minimum path is 5, so flow on each edge in the path x_1, x_2, x_5, x_7 will increase an amount of 5, and reverse edges in that path will decrease the same. The total max-flow will be 5 and total cost will be 6.

ITERATION 2



Vertex	x1	x2	x3	x4	x5	x6	x7	Parent	Queue
0	0	∞	∞	∞	∞	∞	∞	X	x1
x1	0	∞	2	2	∞	∞	∞	X	x3 x4
x3	0	∞	2	2	5	6	∞	x1	x4 x5 x6
x4	0	∞	2	2	5	5	∞	x1	x5 x6
x5	0	∞	2	2	5	5	8	x3	x6 x7
x6	0	∞	2	2	5	5	8	x4	x7
x7	0	∞	2	2	5	5	8	x5	

As we can observe, the capacity of edges x_1, x_2, x_5, x_7 has decreased an amount equal to its smallest residual capacity on that path.

Dijkstra procedure:

As the capacity from x_1 to x_2 is 0, so there is no path to that vertex. Otherwise, x_3 and x_4 will be push into the queue and both their labels are updated to 2.

Then x_3 is pop from the queue, there are suitable paths which is (x_3, x_5) and (x_3, x_6). These vertexes are push into the queue and their labels are update form ∞ to 5 and 6 respectively.

Next, x_4 is taken out from the queue, there are suitable paths which is (x_4, x_3) and (x_4, x_6). The label of x_3 is 2 which is smaller than the sum of $c_{43}(3)$ + label of $x_4(2) = 5$; therefore, x_3 will not be updated. Otherwise, the cost of the path from x_4 to x_6 is 5 which is smaller than the current label of x_6, then the label can be updated.

For the next iteration, the cost from x_5 to x_6 is 7 which is greater than the latest update of label x_6. Hence, there is no change in the graph. The only change can be made is label of x_7 as current label is ∞ so the cost can be updated to 8.

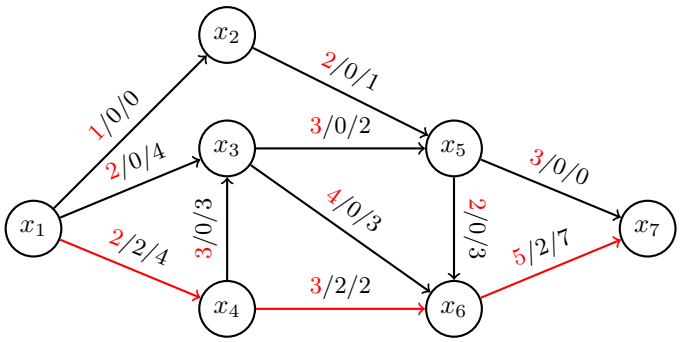
Come up with vertex x_6, the cost from x_6 to x_7 is 10 which means there is no change in the table. Finally, as x_7 is the sink, so the function Dijkstra will terminate.

The path with smallest traveling time in this iteration is x_1, x_3, x_5, x_7 which is 8.

Augmenting procedure

After finding the smallest cost path, the flow can then be increased. As the smallest residual capacity can be resulted from the edge (x_5, x_7) (3), the residual capacity and flow of edges in this path will be fell and rose respectively. In conclusion, the total flow is 8 and the cost of examined path is 54.

ITERATION 3



Vertex	x1	x2	x3	x4	x5	x6	x7	Parent	Queue
0	0	∞	∞	∞	∞	∞	∞	X	x1
x1	0	∞	2	2	∞	∞	∞	X	x3 x4
x3	0	∞	2	2	5	6	∞	x1	x4 x5 x6
x4	0	∞	2	2	5	5	∞	x1	x5 x6
x5	0	∞	2	2	5	5	∞	x2	x6 x7
x6	0	∞	2	2	5	5	10	x4	x7
x7	0	∞	2	2	5	5	10	x6	

Dijkstra procedure

First, x_3 is pop and then update the label to 2, similar with x_4. Then, from x_3, x_5 and x_6 will be updated. The cost from x_4 to x_6 is smaller so x_6's label has new value.

From x_5 there will be only one way to x_6 as the residual capacity of edge (x_5, x_7) is 0 so there cannot have a flow on that edge. The label of x_6 also cannot update as the cost on that path is larger than current label.

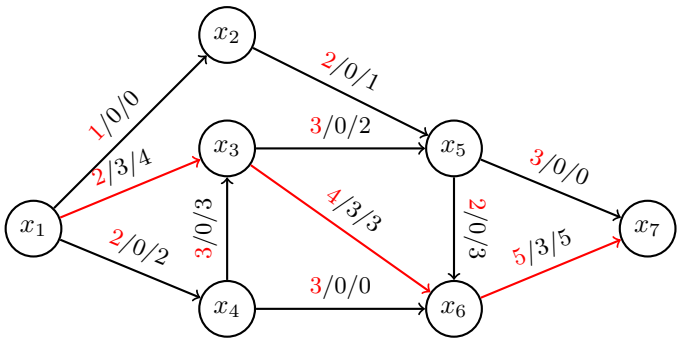
From x_6 the label of x_7 can be updated as the residual capacity is greater than 0 as well as the current label of x_7 is ∞.

After terminating Dijkstra, the min-cost path is x_1, x_4, x_6, x_7 with the cost of 10.

Augmenting procedure

For the minimum path, we can calculate the minimal residual capacity which is 2, therefore the labels of flow and residual capacity will be updated as above iterations. Finally, the current flow of the graph is 10 and total cost is 74.

ITERATION 4



Vertex	x1	x2	x3	x4	x5	x6	x7	Parent	Queue
0	0	∞	∞	∞	∞	∞	∞	X	x1
x1	0	∞	2	2	∞	∞	∞	X	x3 x4
x2	0	∞	2	2	5	6	∞	x1	x4 x5 x6
x3	0	∞	2	2	5	6	∞	x1	x5 x6
x4	0	∞	2	2	5	6	∞	x3	x6 x7
x5	0	∞	2	2	5	6	11	x3	
x6	0	∞	2	2	5	6	11	x6	

Dijkstra procedure

Similar to above iterations, from x_3 we have paths to x_5 and x_6 whose labels will be updated to 5 and 6. In vertex x_4, as the residual capacity of edge (x_4, x_6) was updated to 0 in previous iteration so we can not go through this path.

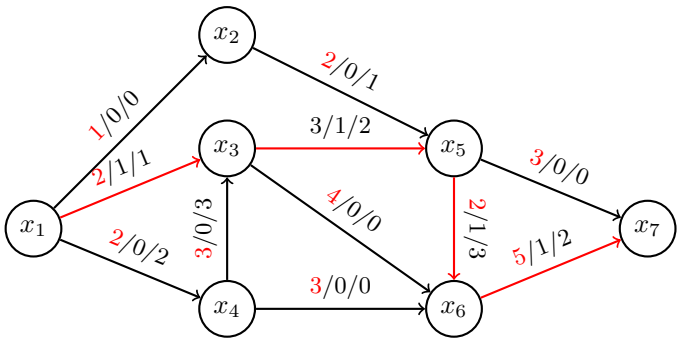
Next, vertex x_5 is taken out from the queue, as the label of x_6 is less than the cost from x_5 to x_6 so it will not be changed. From x_6 there is only a path to x_7 so it will be updated.

The min-cost path can be found by tracking their parents, starting from vertex x_7, therefore we can find the minimum cost path which is x_1, x_3, x_6, x_7 with the cost of 11.

Augmenting procedure

After find the most suitable path, we explore the mimimal residual capacity on that path which is 3. The edges on the path will be updated as above iterations. The procedure terminates with total flow is 13 and the total cost is 107.

ITERATION 5



Vertex	x1	x2	x3	x4	x5	x6	x7	Parent	Queue
0	0	∞	∞	∞	∞	∞	∞	X	x1
x1	0	∞	2	2	∞	∞	∞	X	x3 x4
x3	0	∞	2	2	5	∞	∞	x1	x4 x5
x4	0	∞	2	2	5	∞	∞	x1	x5
x5	0	∞	2	2	5	7	∞	x3	x6
x6	0	∞	2	2	5	7	12	x5	x7
x7	0	∞	2	2	5	7	12	x6	

Dijkstra procedure

Beginning with vertex x_1, x_3 and x_4 will be updated to 2. At the next step, as there is no path from x_3 to x_6, so only x_5 will be updated to 5. With vertex x_4 the residual capacity of edge (x_4, x_6) is 0 and total cost on path from x_4 to x_3 is greater than the current label of vertex x_3; therefore, no vertex is updated.

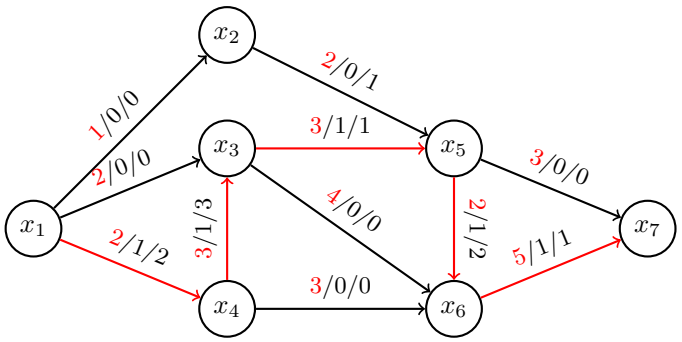
x_5 is then taken out from the queue, as the only suitable edge is (x_5, x_6) and currently the label of x_6 is ∞, so x_6 is updated to 7. Finally, x_6 to x_7 is the only satisfactory path of this graph.

Hence, we conclude the minimum cost path in this iteration is x_1, x_3, x_5,x_6 , x_7 with the min-cost of 12.

Augmenting procedure

As we can observe, the minimum residual capacity is 1, so the flows will be updated with the same instruction as above iterations. Hence, the total flow and the total cost are 14 and 119, respectively.

ITERATION 6



Vertex	x1	x2	x3	x4	x5	x6	x7	Parent	Queue
0	0	∞	∞	∞	∞	∞	∞	X	1
x1	0	∞	∞	2	∞	∞	∞	X	4
x4	0	∞	5	2	∞	∞	∞	x1	3
x3	0	∞	5	2	8	∞	∞	x4	5
x5	0	∞	5	2	8	10	∞	x3	6
x6	0	∞	5	2	8	10	15	x5	7
x7	0	∞	5	2	8	10	15	x6	

Dijkstra procedure

Beginning with vertex x_1, as the residual capacity of edge (x_1, x_3) is 0 so the only vertex that can be updated is x_4. From x_4, the only feasible path is toward the vertex x_3; therefore, label of x_3 is 5 then. With vertex x_3, there is only one suitable edge which is (x_3, x_5). Hence, vertex x_5 is updated to 8 in this step. From x_5, vertex x_6 will be updated to 10. Finally, x_7 will be updated with the label of 15.

We conclude the minimum cost path is x_1, x_4, x_3, x_5,x_6 , x_7 with the minimum cost of 15.

Augmenting procedure

As we can observe, the minimum residual capacity is 1, so the flows will be updated with the same instruction as above iterations. Hence, the total flow and the total cost are 15 and 134, respectively.

After this iteration, the residual capacity from the source vertex to other nodes are 0, so the max flow is achieved and the algorithm can be stopped. Finally, we can find the optimal solution for this example minimum cost flow problem with the shortest travelling time is 134 (mins) and the maximum people on the network is 15.

3.6 Verify the effectiveness of the successive shortest path algorithm

3.6.1 Time complexity

To investigate the time complexity of the successive shortest path algorithm, we examine the individual steps involved in each iteration. Each iteration will:

- Find shortest path using Dijkstra's algorithm: Traverse through all vertices and check all possible edges of the vertices for the shortest path to the sink. The time complexity of this procedure is O(V+E). Across each vertex, the algorithm must check for unvisited vertices in the queue and choose the vertex with minimal distance. Using min-heap map, the complexity of this process is reduced to O(log(V)). So the total complexity is O((V+E)*log(V)) with the heap map

- Augment flows: Augmenting the flow along the shortest path involves traversing the path and updating residual capacities by the minimum residual capacity. This operation has a time complexity proportional to the length of the path, but in the worst case, it is O(E) since the path may visit all edges.

In conclusion, the time complexity of each iteration is O((V+E)*E). The algorithm will repeat the process till the graph achieves maximal flow. In worst scenario, all cycles will increase the flow by 1 until reaching maximal flow. In conclusion, technically the time complexity of the whole process in worst case scenario is:

O((V+E)*log(V)*E*U) with U as maximal flow of the graph.

This is a reasonable time complexity for small and large graph. Note that in practical, large scale graphs are ideally implemented to maximize the flow in each iteration, reducing the number of iterations required to reach maximal flow to a lot lesser than U. Despite the inability to deal with negative weight edges, this algorithm is shown to be extremely efficient dealing with dense graph with large number of vertices ($V > E^2$).

References

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

[2] S. W. Wallace and W. T. Ziemba, *Applications of stochastic programming.*, SIAM, 2005.