

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



## MATHEMATICAL MODELING (CO2011)

---

Assignment (Semester: 231, Duration: 10 weeks)

# Stochastic Programming and Applications

---

Instructor: 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: Cao Ngọc Lâm - 2252419 - Leader  
Nguyễn Châu Hoàng Long - 2252444  
Lê Nhân Văn - 2252899  
Lý Triều Uy - 2252889

HO CHI MINH CITY, NOVEMBER 2023



## Contents

<b>1</b>	<b>Member list &amp; Workload</b>	<b>2</b>
<b>2</b>	<b>Stochastic Programming for Industry - Manufacturing problem</b>	<b>3</b>
2.1	Theory . . . . .	3
2.2	Solution steps . . . . .	4
2.3	Program Implementation . . . . .	4
2.3.1	Import libraries . . . . .	4
2.3.2	Initialize parameters . . . . .	5
2.3.3	Build first stage model . . . . .	6
2.3.4	Build second stage model . . . . .	7
<b>3</b>	<b>Stochastic Linear Program for Evacuation Planning In Disaster Responses (SLP-EPDR)</b>	<b>9</b>
3.1	Introduction . . . . .	9
3.1.1	Aim and motivation . . . . .	9
3.1.2	Overview of Two-Stage Stochastic Programming . . . . .	11
3.1.3	Introduction to Min Cost flow problem . . . . .	12
3.2	Problem Analysis . . . . .	12
3.2.1	Evacuation process illustration . . . . .	12
3.2.2	Multiple sources and sinks conversion . . . . .	13
3.3	Model formulation . . . . .	14
3.3.1	Notations and decision variables . . . . .	14
3.3.2	System constraints . . . . .	15
3.4	Solution Algorithm . . . . .	17
3.4.1	Model decomposition and sub problems . . . . .	17
3.4.2	Successive shortest path algorithm for solving sub problem 1 . . . . .	19
3.4.2.a	Bellman-Ford algorithm . . . . .	19
3.4.2.b	Ford-Fulkerson algorithm . . . . .	21
3.4.2.c	Successive shortest path algorithm for min-cost max-flow problem . . . . .	24
3.5	Experimental Design . . . . .	32
3.5.1	Experiment network generation . . . . .	32
3.5.2	Solving problem with two-stage stochastic programming using min-cost flow algorithm . . . . .	37
3.6	Conclusion . . . . .	42



## 1 Member list & Workload

No.	Name	Student ID	Work	Percentage of work
1	Cao Ngọc Lâm	2252419	<ul style="list-style-type: none"><li>- Write the main LaTeX file and Log Diary file.</li><li>- Write the Introduction, Problem analysis and Model decomposition section for SPL-EPDR.</li><li>- Writing code illustrating a simple network generation.</li></ul>	30%
2	Nguyễn Châu Hoàng Long	2252444	<ul style="list-style-type: none"><li>- Write the Solution Algorithm Section for SLP-EPDR</li><li>- Implement and verify the effectiveness of Min-cost flow Algorithm on simple network</li></ul>	30%
3	Lý Triều Uy	2252889	<ul style="list-style-type: none"><li>- Write code to generate data and find the optimal solution for Industry Manufacturing problem</li></ul>	20%
3	Lê Nhân Văn	2252899	<ul style="list-style-type: none"><li>- Study and writing theory, model formulation, solution steps for Industry Manufacturing problem</li></ul>	20%

The Log Diary file included in the submission will show the weekly work process for all 10 weeks before the deadline, as well as our discussion about the assignment. The references we used for finishing this assignment is included in the final part of the report. Besides the listed references, the support file from instructors also helped us to finish the report.

## 2 Stochastic Programming for Industry - Manufacturing problem

### 2.1 Theory

In Industry - manufacturing problem, we consider a manufacturer can produce  $n$  products. There will be  $m$  parts of the product, which have to be supplied from third-party suppliers. A unit of  $i$  required  $a_{ij} \geq 0$  units of part  $j$ , where  $i = 1, 2, \dots, n$  and  $j = 1, 2, \dots, m$ . For conveniently transforming to the model latter on, we will make a matrix  $A = [a_{ij}]$  of dimension  $n \times m$ . The demand of the product is modelled as a random vector  $D = \{D_1, D_2, \dots, D_n\}$ .

For an observed scenario  $d = (d_1, d_2, \dots, d_n)$ . We considered the equation below:

$$\begin{cases} \min_{z,y} Z = c^T \cdot z - s^T \cdot y \\ s.t. \\ c = (c_i := l_i - q_i) : \text{cost coefficients} \\ y = x - A^T \cdot z, A = [a_{ij}] \\ 0 \leq z \leq d, y \geq 0 \end{cases} \quad (1)$$

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  $x = (x_1, x_2, \dots, x_n)$

The objective of Equation (1) is to minimize the linear combination of the decision variables  $z$  and  $y$ , subject to certain constraints. The coefficients  $c_i$  in the cost vector  $c$  represent the difference between holding costs ( $l_i$ ) and order quantities ( $q_i$ ) for each product, capturing the trade-off between these costs. The vector  $s$  represents the setup costs associated with production. The decision variable  $z$  typically represents the number of units produced, while  $y$  represents the number of parts left in inventory.

The solution to this optimization problem, which determines the values of  $z$  and  $y$ , depends not only on the 1st-stage decision  $x$  but also on the realization of the random demand  $\omega = D$ . Hence, the values of  $z$  and  $y$  will vary based on the specific demand scenario.

The objective is indeed to minimize the total cost, taking into consideration the trade-off between holding costs, order quantities, and setup costs. By finding the optimal values for  $z$  and  $y$ , the goal is to achieve the most cost-effective production and inventory management strategy.

Following the distribution-based approach, we consider the second equation:

$$\min g(x, y, z) = b^T x + Q(x) = b^T x + E[Z(z)] \quad (2)$$

With:  $Q(x) = E_\omega[Z] = \sum_{i=1}^n p_i \cdot c_i \cdot z_i$

We will consider clearly all the notations in equation (2):

- $b$ : the vector representing the data associated with the problem.
- $x$ : the decision variables
- $b^T x$ : the dot product of  $b^T$  and  $x$ , which represents the linear contribution of the data vector  $b$  to the objective function.
- $Q(x)$ : the function representing the expected cost associated with the decision variables  $x$ .
- $p_i$  represents the density or probability associated with each demand value. It reflects the likelihood or likelihood-weighted average of different demand scenarios.
- $c_i$  is the cost coefficient associated with each product. It captures the cost of producing or procuring each product.
- $z_i$  is the decision variable representing the production quantities or ordering quantities for each product.

It can be observed that the product of  $p_i$ ,  $c_i$ , and  $z_i$  captures the expected cost for each demand scenario and product. The summation over all products and demand scenarios gives the overall expected cost  $Q(x)$ .

**In summary**, Equation (2) combines the linear contribution of the data vector  $b$  with the expected cost  $Q(x)$  associated with the decision variables  $x$ . The objective is to minimize the combined cost,  $g(x, y, z)$ , by finding the optimal values for the decision variables that minimize the total cost considering both the linear contributions and the expected cost under different demand scenarios.

## 2.2 Solution steps

To solve the problem using the 2-SLPWR model given in Equations (1) and (2), you can follow these steps:

1. Generate the Data: Randomly simulate the data vectors  $b$ ,  $l$ ,  $q$ , and  $s$ , as well as the matrix  $A$ , based on the problem description. These data vectors and matrix represent the cost coefficients, production information, and ordering information for the problem.
2. Calculate the cost coefficients: Compute the cost coefficients  $c$  as the element-wise difference between  $l$  and  $q$ . The cost coefficients represent the cost associated with each product.
3. Define the Objective Function: Define the objective function based on Equation (1). In this case, the objective function is  $\min_{z,y} Z = c^T \cdot z - s^T \cdot y$ . The objective is to minimize the cost  $Z$ , which is the dot product of the cost coefficients  $c$  and the decision variables  $z$ , minus the dot product of the decision variables  $y$  and  $s$ .
4. Define the Constraints: Define the constraints based on Equation (1). The constraint is given as  $y = x - A^T \cdot z$ , where  $Y$  is the difference between actual production and demand. The constraint ensures that the production quantities and demand are balanced.
5. Define the Q Function: Define the Q function based on Equation (2). The Q function is defined as  $Q(x) = \sum_{i=1}^n p_i \cdot c_i \cdot z_i$ , where  $p_i$  is the density associated with each demand value,  $c_i$  is the cost coefficient, and  $z_i$  is the decision variable. The Q function represents the expected cost associated with the decision variables  $x$ .
6. Solve the Problem: Use an optimization algorithm to solve the problem and find the optimal solution. You can use techniques such as linear programming, quadratic programming, or other optimization methods. The specific choice of the optimization algorithm depends on the problem characteristics and constraints.

## 2.3 Program Implementation

In order to implement the solution steps above in Python, we will use Gurobipy, a mathematical optimization software library for solving mixed-integer linear and quadratic optimization problems. The program will be divided into 4 parts below:

### 2.3.1 Import libraries

```
1 import gurobipy
2 import numpy as np
3 from numpy import random
4
```

Import gurobipy: this library is often used to solve linear, nonlinear and mixed optimization problems.

Import numpy as np: Imports the NumPy library and gives it the alias np. NumPy is often used for numerical operations and array manipulations in Python.

From numpy import random: Specifically imports the random module from NumPy, which provides functions for generating random numbers.

### 2.3.2 Initialize parameters

```
1 n=8
2 m=5
3 A = np.array([
4     np.random.randint(low=1,high=10,size=m),
5     np.random.randint(low=1,high=10,size=m),
6     np.random.randint(low=1,high=10,size=m),
7     np.random.randint(low=1,high=10,size=m),
8     np.random.randint(low=1,high=10,size=m),
9     np.random.randint(low=1,high=10,size=m),
10    np.random.randint(low=1,high=10,size=m),
11    np.random.randint(low=1,high=10,size=m),
12 ])
13
```

Next, we will initialize two variables m,n. These two variables are used to initialize the size of matrix A. Matrix A has m as the number of columns and n as the number of rows and m is initialized as 5 and n is initialized as 8. So matrix A has size is 8x5

```
1 b = np.array([np.random.randint(50,150),
2               np.random.randint(50,150),
3               np.random.randint(50,150),
4               np.random.randint(50,150),
5               np.random.randint(50,150)])
6
```

Next we come to matrix B, this matrix has size 5x1 with any five integers. These 5 integers range from 50 to 150.

```
1 l = np.array([np.random.randint(low=100,high=200,size=n)])
2
```

Next we will create an array named l with size 8. These 8 variables of the array are any 8 integers with values from 100 to 200.

```
1 q = np.matmul(A,b)+l + np.array([np.random.randint(low=200,high=400,size=n)])
2
```

This stage of the code calculates the selling cost by using the amount of money to produce plus the revenue you expect from that product

```
1 s = np.array([np.random.randint(1,10),
2               np.random.randint(1,10),
3               np.random.randint(1,10),
4               np.random.randint(1,10),
5               np.random.randint(1,10)])
6
```

Next we will create an array S of size 5 with any 5 integer values. These integer values will be placed in the range (1,10).

```
1 D = np.array([np.random.binomial(10,0.5,8),np.random.binomial(10,0.5,8)])
2
```

Create an array D of size 8 with 8 variables representing the results of a binomial random variable based on binomimal distribution.

```
1 print("The matrix A size 8x5 with 8 is products and j is suppliers: ")
2 print(A)
3 print("The random demand D for the product:")
4 print(D)
5 print("The selling price of the product:")
6 print(q)
7 print("The preorder cost b:")
8 print(b)
9 print("The salvage value s:")
10 print(s)
11 print("The addition cost l:")
12 print(l)
13
```

This will print the input information.

This is the input information for the model

```
The matrix A size 8x5 with 8 is products and j is suppliers:
[[9 8 7 5 3]
 [8 6 6 2 4]
 [6 2 2 1 3]
 [4 6 3 3 4]
 [9 3 5 2 6]
 [2 6 5 1 1]
 [3 5 7 1 6]
 [9 1 7 6 5]]
The random demand D for the product:
[[3 4 3 4 4 5 5 4]
 [4 5 7 2 6 5 5 4]]
The selling price of the product:
[[4298 3643 2204 2787 3649 2164 3300 3759]]
The preorder cost b:
[124 107 139 84 144]
The salvage value s:
[1 8 1 2 8]
The addition cost l:
[[196 166 198 189 169 112 169 126]]
```

### 2.3.3 Build first stage model

```
1 firststage=gurobipy.Model("S2P")
2 firststage.ModelSense=gurobipy.GRB.MINIMIZE
3 firststage.setParam('OutputFlag',0)
4
```

This code sets up an optimization model in Gurobi called S2P to prepare for the definition of variables, conditions and functions to prepare for the next steps of the problem.

```
1 x = firststage.addMVar((m,1), vtype = gurobipy.GRB.INTEGER, name = "x")
2 y1 = firststage.addMVar((m,2), vtype = gurobipy.GRB.INTEGER, name = "y1")
3 z1 = firststage.addMVar((n,2), vtype = gurobipy.GRB.INTEGER, name = "z1")
4
```

This step of the lesson is used to add new variables x,y1 and z1. The variables y1 and z1 will have 2 scenarios while the variable x will only have 1 scenario.

```
1 firststage.addConstr(x>=0)
2 firststage.addConstr(y1>=0)
3 firststage.addConstr(z1>=0)
4 firststage.addConstr(z1<=D.T)
5 firststage.addConstr(y1 == x - A.T @ z1)
6
```

Next we will add to the model some necessary constraints for the variables and relationships between variables in the article.

```
1 firststage.setObjective(b.T @ x + gurobipy.quicksum(((1-q) @ z1[:,k] - s @ y1[:,k]
2   )*0.5 for k in range(2)))
3 firststage.optimize()
4
```

The goal of this code is to minimize the equation calculated based on the variables  $x, y1, z1$  and the data vectors  $b, l, q, s$  using the equation:  $\min g(x, y, z) = b^T x + Q(x) = b^T x + E[Z(z)]$

```
1 x_value = x.x
2 y1_value = y1.x
3 z1_value = z1.x
4
```

Next we will get the values of the variables after the model is optimized.

```
1 print("Optimal Solution:")
2 print('Obj: %g' % firststage.objVal)
3 print(f"x = {x_value}")
4 print(f"y1 = {y1_value}")
5 print(f"z1 = {z1_value}")
6
```

Next we will print its values.

So here is the output for the first stage model:

```
Optimal Solution:
Obj: -7993
x = [[182.]
     [137.]
     [165.]
     [ 74.]
     [121.]]
y1 = [[0. 0.]
      [0. 0.]
      [0. 0.]
      [0. 0.]
      [0. 0.]]
z1 = [[3. 3.]
      [4. 4.]
      [3. 3.]
      [2. 2.]
      [4. 4.]
      [5. 5.]
      [5. 5.]
      [4. 4.]]
```

### 2.3.4 Build second stage model

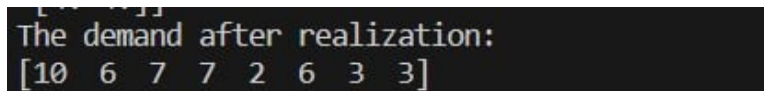
```
1 D = np.array([10,6,7,7,2,6,3,3])
2
```

Then we will create an array  $D$  with 8 elements, all of which are any integer values ranging from 1 to 100.



```
1 print("The demand after realization:")
2 print(D)
3
```

Here is the demand after realization:



```
[10 6 7 7 2 6 3 3]
```

After that, we will print the input

```
1 secondstage=gurobipy.Model("SecondStage")
2 secondstage.ModelSense=gurobipy.GRB.MINIMIZE
3 y2 = secondstage.addMVar((m,1), vtype = gurobipy.GRB.INTEGER, name = "y2")
4 z2 = secondstage.addMVar((n,1), vtype = gurobipy.GRB.INTEGER, name = "z2")
5
```

Next we will create an optimization model named second stage. Then we create variables y2 and z2.

```
1 secondstage.addConstr(y2>=0)
2 secondstage.addConstr(z2>=0)
3 secondstage.addConstr(z2<=D)
4 secondstage.addConstr(y2 == x_value - A.T @ z2)
5
```

Then we will set the binding conditions for the variables y2 and z2 as well as the relationship between the variables and the data vectors in the lesson.

```
1 secondstage.setObjective(gurobipy.quicksum(((1-q) @ z2 - s @ y2)))
2 secondstage.optimize()
3
```

At this step, this code will minimize the function based on the variables z2 and y2 and the data vectors l,q,s in the article.

```
1 y2_value = y2.x
2 z2_value = z2.x
3
```

Then we get the value of the variable after the model is optimized

```
1 print('Obj: %g' % secondstage.objVal)
2 print(f"y2 = {y2_value}")
3 print(f"z2 = {z2_value}")
4
```

And then print those values out.

And this is the output of the second stage model:

```
Obj: -50145
y2 = [[82.]
      [63.]
      [81.]
      [32.]
      [57.]]
z2 = [[2.]
      [2.]
      [2.]
      [2.]
      [2.]
      [2.]
      [2.]]
```

### 3 Stochastic Linear Program for Evacuation Planning In Disaster Responses (SLP-EPDR)

This part will introduce the application of Two-Stage Stochastic Programming - a practical optimization method for the evacuation planning in disaster responses in theory based. There will be four subsections, with the initial one will introduce the aim and motivation for studying evacuation planning, as well as the overview of Two-stage Stochastic Programming. Subsection 2 will depicts clearly the process of Two-stage Stochastic Programming on a sample network, while subsection 3 discuss the mathematical formulation of the Stochastic model. Subsection 4 mentions about 2 subproblems decomposed from the relaxation model, as well as the successive shortest path algorithm for min-cost flow problem to solve these 2 subproblems. The last subsection will demonstrate the application of the algorithm mentioned in subsection 4 to make a experimental design on a small grid network.

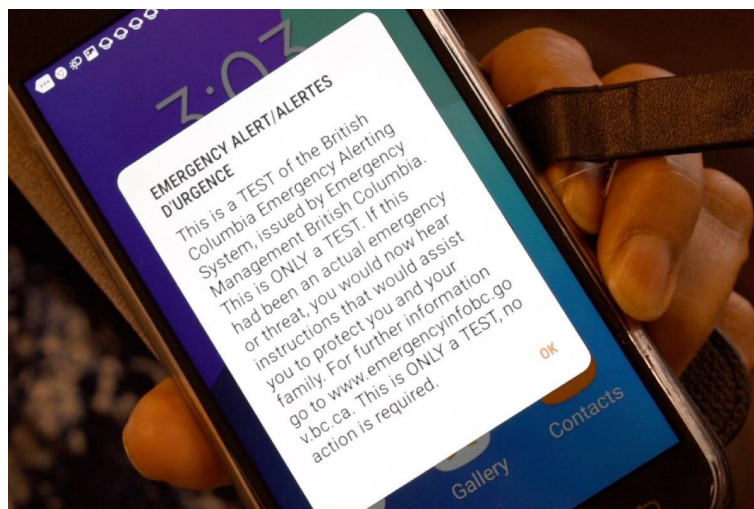
#### 3.1 Introduction

##### 3.1.1 Aim and motivation

It can be observed that extreme natural disasters like floods, cyclones and earthquakes or unnatural ones(war, terrorist) have always been an integral part of the Earth and the human social, causing to loss of lives and property. Many emergency responses are given out in order to provide shelters and necessary items for affected people as soon as possible. Evacuation planning is a critical component of disaster response, helping save lives by strike out the plan for people to safely evacuate from the affected areas to safe ones with minimum travel time. As can be seen, the evolution of a disaster scenario is inherently uncertain in capacity, travel time and can unfold in unpredictable ways, making the evacuation planning more complex to solve. To address this uncertainty and enhance the resilience of evacuation plans, many methods were given out. In this assignment, we will focus on **Stochastic Programming framework with Recourse (Dantzig)**, which emerges as a powerful and flexible approach that accommodates uncertainty in finding non-anticipate decisions that have to be made before knowing the realizations of random

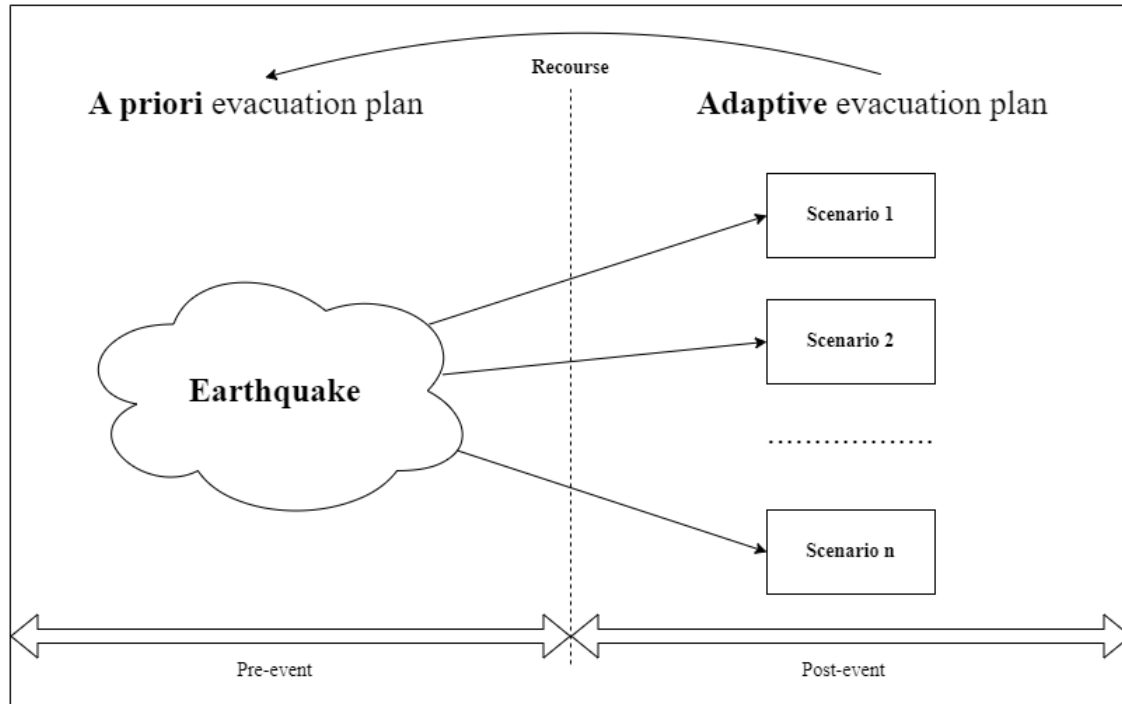


variables. Since there will be two stages taken into account, this framework can be referred to as **Two-stage Stochastic Programming with Recourse**.



In the era of Internet and Information technology, individuals can readily access real-time information after unexpected events through a multitude of channels. Consequently, this assignment will embark on determining the optimal evacuation path for affected individuals in the context of real-time information availability. Accordingly, the evacuation process will be divided into two distinct stage. In the initial stage, it is assumed that the affected people lack information about disaster level and road damage severity. After a specific time interval, they gain access to precise road network information facilitated by real-time monitoring equipment. These information will be used to form a stochastic with recourse model, which is used to find non-anticipative decisions that must be taken priori to know the realization of stochastic variable such as capacity and time travel in the transportation network.

### 3.1.2 Overview of Two-Stage Stochastic Programming



Hình 1: Illustration of Two-stage stochastic programming

A scenario is a possible realization of the state of the phenomenon. In the context of disaster responses, each scenario represents a combination of factors that affect the evacuation process, such as the capacity or the travel time in each link. For instance, some parts of the transformation network from the affected area to the shelter may be destroyed during the disaster, affecting heavily on the initial evacuation plan. In the Two-Stage Stochastic Programming, the randomness attributes of the phenomenon can be reflected by a set of finite number of scenarios  $S = \{s_1, s_2, s_3, \dots, s_n\}$ ,  $n \in \mathbb{N}^*$ ; each scenarios  $s_i$  will be associated with a known probability  $p_i$ . For simplicity, there will be 10 distinct scenarios to be considered.

In [Fig.1], the **Pre-event stage** (1st stage) represents the decision-maker makes decisions based on the current information available before the information of stochastic variables is available. In this context, the information includes demands of each region, the capacity and travel time of each link in the transportation network. The **Post-event stage** (2nd stage) will occur after the realization of stochastic travel times and capacities at the time threshold  $T$ , the goal of the 2nd stage is to update the preliminary decisions made in the priori stage based on the new information. The decision-maker may need to modify the allocation of resources, change the evacuation routes, or make other adjustments to the plan in order to adapt to the unfolding situation.

**In summary**, the objective of first stage is making the robust and reliable evacuation plan for all scenarios listed, while the second stage focuses on making adaptive evacuation plan for the initial decisions in the 1st stage.

### 3.1.3 Introduction to Min Cost flow problem

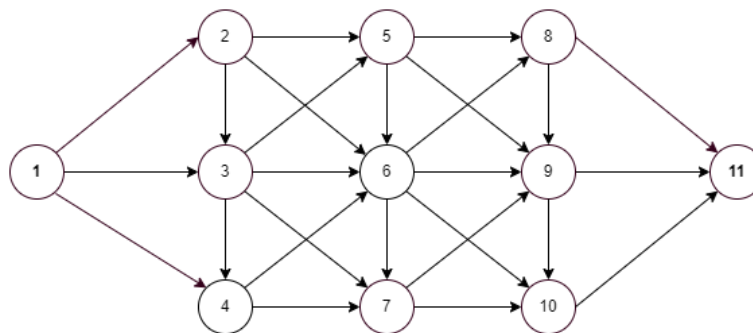
The objective of this method is to move the people from dangerous areas to safe areas with the minimum evacuation time in a capacity-cost network. In other words, the model is trying to find the best way to evacuate people from dangerous areas to safe areas in a timely and efficient manner, taking into account the random travel times and capacities of different roads. To do this, we need to use a min-cost flow problem. This is a type of optimization problem that finds the cheapest way to send a certain amount of flow from a set of source nodes to a set of sink nodes, subject to capacity constraints on the edges of the network. In the context of the evacuation problem, the source nodes represent the dangerous areas, the sink nodes represent the safe areas, and the edges represent the roads. The capacity of an edge represents the maximum number of people that can pass through that road in a given time period. The objective of the min-cost flow problem is to find a path from each source node to each sink node that minimizes the total evacuation time. This is done by taking into account the random travel times of the different roads. The algorithm proposed below will solve the min-cost flow problem using a stochastic programming approach. This means that it considers different scenarios for the random travel times and capacities, and finds the best evacuation plan for each scenario.

## 3.2 Problem Analysis

This section will depict the evacuation process as a two-stage stochastic program on simple networks.

### 3.2.1 Evacuation process illustration

[Fig. 2] represents a simple transportation network with 11 nodes and 22 links, each link is assigned a known capacity and travel time. Node 1 is considered the disaster area and node 11 is assumed to be the safe ones. In initial, there will be 4 cars namely a, b, c and d in node 1. There will be 2 scenarios after the time threshold  $T$  occur. Our objective is planning the path for 4 cars to move from node 1 to node 11 with minimum travel time while also satisfies the capacity of the network.



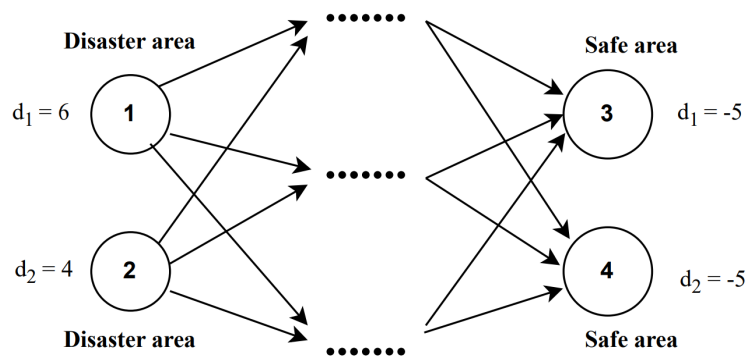
Hình 2: Sample network

In pre-event, these 4 cars are assumed to follow the priori plan, i.e car a and b follow the path 1->2->5->6->9->11, while car c and d follow the path 1->3->4->7->10->11. After the time threshold  $T$ , the later paths may change adaptively. For example, car a and b adaptive a new path: 1->2->5->6->10->11, the new path for car c is 1->3->4->6->8->11 while the new

one for car d is 1->3->4->7->9->11.

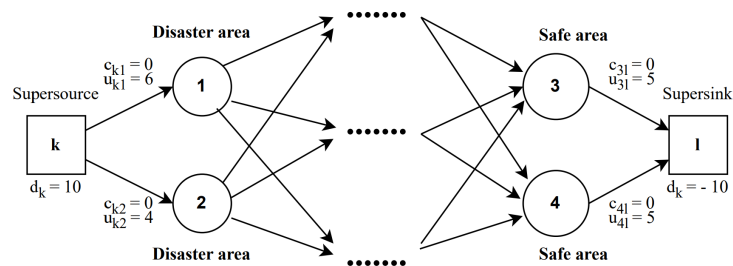
### 3.2.2 Multiple sources and sinks conversion

In transportation network, there maybe more than one source and sink. For conveniently, we should convert this network with many sources and sinks to a network with only one supersource and one supersink. Consider this physical network below:



Hình 3: Before

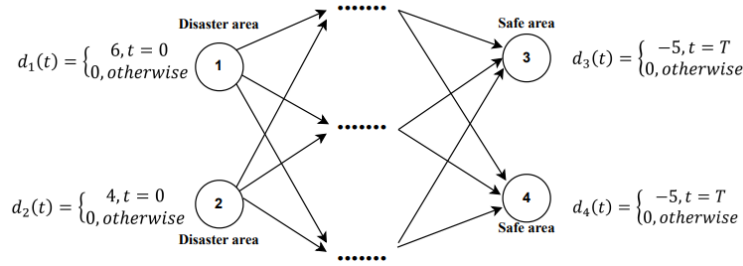
We will add a supersource  $k$  to the network, and meanwhile the dummy arcs  $(k, i)$  should be added with value 0, i.e,  $c_{ki} = 0, i \in K$ , where  $K$  is the set of source nodes, and let the capacity  $u_{ki} = d_i, i \in K$ . That leads to the supply value for supersource:  $d_k = \sum_{i \in K} d_i$ . The supersink can be converted by the same way. Applying this method, we have the converted network:



Hình 4: After

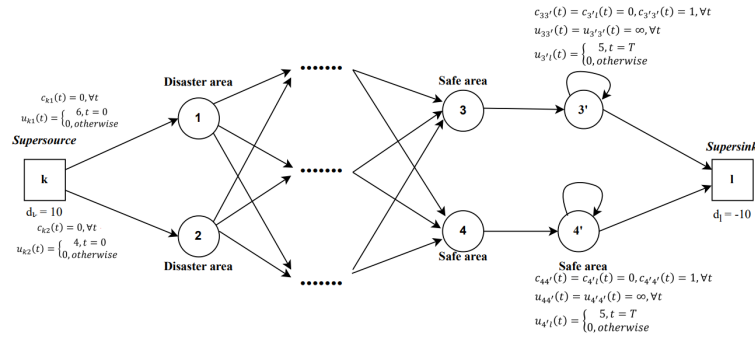
In a time-dependent network, the travel time and capacity of an arc can vary depending on the departure time. To add a supersource to such a network, we set the travel time and capacity of the dummy arcs are  $c_{kj}(t) = 0, t \in \{0, 1, \dots, T\}, i \in K$  and the capacity equals to the supply of

node  $i$  at time  $t$ , respectively. We also convert the multiple sinks to a supersink by adding a copy  $j'$  for each original sink  $j$  and connecting them with an arc of infinite capacity, i.e.,  $u_{jj'} = \infty$ , and zero travel time, i.e.  $c_{jj'}^s = 0, \forall t \in \{0, 1, \dots, T\}$ . Additionally, we add a self-loop to each node  $j'$  with infinite capacity and travel time of 1, i.e.,  $u_{j'j'} = \infty$  and  $c_{j'j'}^s = 1$ . The capacity of the arc  $(j', 1)$  is set to the demand of node  $j$  at time  $T$ , and the capacity at other times is set to 0. The travel time is assumed to be 0 in the considered time horizon. Consider this time-dependant network:



Hình 5: Before

Applying the method above, we have the converted network:



Hình 6: After

### 3.3 Model formulation

#### 3.3.1 Notations and decision variables

The following tables provide the relevant notations with their definitions used in the mathematical formulation.

In the construction of the model for evacuation planning, two types of decisions variable will be used. In the first stage, the variable  $x_{ij}$  represents the flow on link  $(i, j)$ , that is, the number of people will move from node  $i$  to node  $j$ . Noted that in the case above, we will convert the value  $x_{ji} = -x_{ij}$  (the flow from  $j$  to  $i$ ). In the second stage, the variable  $y_{ij}^s(t)$  represents the flow

Bảng 1: Table of Symbols and Definitions

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
$\mathcal{S}$	the total number of scenarios
$s_i$	the supply value of source node $i$
$\bar{T}$	the time threshold
$\mathcal{T}$	the total number of time intervals
$U_{ij}$	the capacity on physical link $(i, j)$
$u_t^s(i, j)$	the capacity of link $(i, j)$ in scenario $s$ at time $t$
$c_t^s(i, j)$	the travel time of link $(i, j)$ in scenario $s$ at time $t$
$p_s$	the probability in scenario $s$

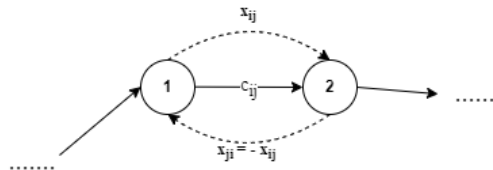
on link  $(i, j)$  in scenario  $s$  at time  $t$ . The convention of  $y_{ij}^s(t)$  will be applied as the variable  $x_{ij}$  above. The flow  $x_{ij}$  on link  $(i, j)$  must satisfy the capacity constraint:  $0 \leq x_{ij} \leq u_{ij}$ . The same condition also apply for the adaptive flow  $y_{ij}^s(t)$  at scenario  $s$  and time  $t$ :  $0 \leq y_{ij}^s(t) \leq u_{ij}^s(t)$ . Table 2 has given straightforward definition for two types of variable.

Bảng 2: Table of Decision Variables Used in Mathematical Formulation

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

### 3.3.2 System constraints

In the first stage, we will determine a feasible solution from super-source to super-sink. Consider this simple flow below:



Hình 7: Sample flow

As can be seen, the flow on node 1 must be equal to the subtraction of the flow on link  $(1, 2)$  and link  $(2, 1)$ , that is:

$$x_{12} - x_{21} = d_1 \quad (3)$$



Consider two node  $i$  and  $j$ , there can be many flow directly from  $i$  to  $j$ . By induction, from (3) we can get the equation:

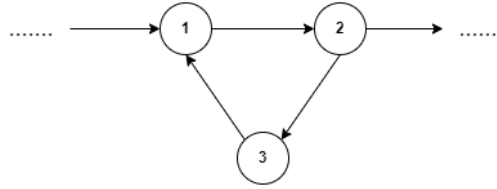
$$\sum_{i,j \in A} x_{ij} - \sum_{i,j \in A} x_{ji} = d_i \quad (4)$$

With  $d_i$  must satisfy the following equations:

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

Meanwhile, the flow on link  $(i,j)$  must satisfy the capacity constraint:

Noted that there would be the case when the flow generate a path with loops if there are potential loops in the network, which can increase the travel time for our evacuation planning :



Hình 8: Sample potential loop

In order to avoid this case, the link penalty  $p_{ij}, (i,j) \in A$  can be used in this penalty function:

$$f(X) = \sum_{i,j \in A} p_{ij} x_{ij} \quad (5)$$

Where  $X := \{x_{ij}\}_{(i,j) \in A}$

This penalty function will be used in the mathematical formulation of stochastic programming.

In the second stage, adaptive paths will be given out for the affected people to meet the real-time information after the time threshold  $T$ . According to the two-stage stochastic programming concepts we discuss above, the affected people will follow a priori plan in the first stage, which means that the evacuation plans in different scenarios is the same as the priori plan. This statement can be depicted in the formula below:

$$y_{ij}^s(t) = x_{ij}, (i,j) \in A, s = 1, 2, \dots, S \quad (6)$$

Coming up to the formulation of the model, we have the standard form of Two-stage Stochastic Programming as follows:

$$\min_x f(X) + \mathbb{E}(Q(Y, s)) \quad (7)$$

In our context,  $f(X)$  is the penalty function in (3), while  $Q(y, s)$  is the overall time of the affected people evacuated from the dangerous area to the safe area in each scenario:

$$Q(Y, s) = \min \sum_{i,j \in A_s} c_{ij}^s(t) \cdot y_{ij}^s(t) \quad (8)$$

Noted that each scenario  $s$  is associated with a known probability  $p_s$ . Combined the equation (7), (8), (4), (5), (6), we got the two-stage evacuation planning model in time-dependant and random environment is formulated as:

$$\min \sum_{i,j \in A} p_{ij} x_{ij} + \sum_{s=1}^S \left( p_s \cdot \sum_{i,j \in A_s} c_{ij}^s(t) \cdot y_{ij}^s(t) \right) \quad (9)$$

s.t.

$$\begin{cases} \sum_{i,j \in A} x_{ij} - \sum_{i,j \in A} x_{ji} = d_i, \forall i \in V \\ 0 \leq x_{ij} \leq u_{ij} \\ \sum_{i_t, j_{t'} \in A} y_{ij}^s(t) - \sum_{i_t, j_{t'} \in A} y_{ji}^s(t') = d_i^s(t), \forall i \in V \\ t \in \{0, 1, 2, \dots, T\}, s = 1, 2, \dots, S \\ 0 \leq y_{ij}^s(t) \leq u_{ij}^s(t) \\ \sum_{t \leq \hat{T}} y_{ij}^s(t) = x_{ij}, (i, j) \in A, s = 1, 2, \dots, S \end{cases}$$

The goal of this model is to develop a optimal evacuation plan to provide path for the affected people to safe area under time-dependant condition. Obviously, both travel time and capacity on each link are scenario-based random variables.

### 3.4 Solution Algorithm

#### 3.4.1 Model decomposition and sub problems

In order to solve the model (9), we will discuss the Lagrangian multiplier  $\alpha_{ij}^s(t), (i, j) \in A, s = 1, 2, \dots, S, t \leq T$ . Using this multiplier, the coupling constraint (6) can be relaxed into the following objective function:

$$\sum_{s=1}^S \sum_{(i,j) \in A} \alpha_{ij}^s(t) (y_{ij}^s(t) - x_{ij}) \quad (10)$$

After applying the above relaxation function, the model (9) can be relaxed as follow:

$$\min \sum_{i,j \in A} p_{ij} x_{ij} + \sum_{s=1}^S \left( p_s \cdot \sum_{i,j \in A_s} c_{ij}^s(t) \cdot y_{ij}^s(t) \right) + \sum_{s=1}^S \sum_{(i,j) \in A} \alpha_{ij}^s(t) (y_{ij}^s(t) - x_{ij}) \quad (11)$$

s.t.

$$\begin{cases} \sum_{i,j \in A} x_{ij} - \sum_{i,j \in A} x_{ji} = d_i, \forall i \in V \\ 0 \leq x_{ij} \leq u_{ij} \\ \sum_{i_t, j_{t'} \in A} y_{ij}^s(t) - \sum_{i_t, j_{t'} \in A} y_{ji}^s(t') = d_i^s(t), \forall i \in V \\ t \in \{0, 1, 2, \dots, T\}, s = 1, 2, \dots, S \\ 0 \leq y_{ij}^s(t) \leq u_{ij}^s(t) \end{cases}$$

For easy of solving, the model (11) can be decomposed into two subproblems:

#### SubProblem 1: Min-cost Max-flow Problem

$$\begin{cases} \min SP1(\alpha) = \sum_{i,j \in A} \left( p_{ij} - \sum_{s=1}^S \sum_{t \leq \tilde{T}} \alpha_{ij}^s(t) \right) x_{ij} \\ s.t. \\ \sum_{i,j \in A} x_{ij} - \sum_{i,j \in A} x_{ji} = d_i, \forall i \in V \\ 0 \leq x_{ij} \leq u_{ij} \end{cases} \quad (12)$$

The objective function of subproblem 1 can be defined as:  $g_{ij} := p_{ij} - \sum_{s=1}^S \sum_{t \leq T} \alpha_{ij}^s(t)$  to represent the generalized cost of each link. Therefore, sub-problem 1 can be solved by the Successive shortest path algorithm.

**SubProblem 2: Time - dependant Min-cost Max-flow Problem**

$$\begin{cases} \min SP2(\alpha) = \sum_{s=1}^S \sum_{(i,j) \in A} \left( \sum_{t \in \{0,1,\dots,T\}} \mu_s \cdot c_{ij}^s(t) + \sum_{t \leq \tilde{T}} \alpha_{ij}^s(t) \right) y_{ij}^s(t) \\ s.t. \\ \sum_{i_t, j_{t'} \in A} y_{ij}^s(t) - \sum_{i_t, j_{t'} \in A} y_{ji}^s(t') = d_i^s(t), \forall i \in V \\ t \in \{0, 1, 2, \dots, T\}, s = 1, 2, \dots, S \\ 0 \leq y_{ij}^s(t) \leq u_{ij}^s(t) \end{cases} \quad (13)$$

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

$$\begin{cases} \min SP2(\alpha, s) = \sum_{(i,j) \in A} \left( \sum_{t \in \{0,1,\dots,T\}} \mu_s \cdot c_{ij}^s(t) + \sum_{t \leq \tilde{T}} \alpha_{ij}^s(t) \right) y_{ij}^s(t) \\ s.t. \\ \sum_{i_t, j_{t'} \in A} y_{ij}^s(t) - \sum_{i_t, j_{t'} \in A} y_{ji}^s(t') = d_i^s(t), \forall i \in V \\ 0 \leq y_{ij}^s(t) \leq u_{ij}^s(t) \end{cases} \quad (14)$$

It can be observed that for each scenario  $s \in \{1, 2, \dots, S\}$ , the model (13) has a similar as model (12) with time dependant cost  $c_{ij}^s(t)$  and link capacity  $u_{ij}^s(t)$  and the generalized cost  $g_{ij}^s(t)$ . This cost can be defined as piecewise function below by two time stages:

$$g_{ij}^s(t) = \begin{cases} \mu_s \cdot c_{ij}^s(t) + \alpha_{ij}^s(t) & \text{if } t \leq \tilde{T} \\ \mu_s \cdot c_{ij}^s(t) & \text{if } \tilde{T} \leq t \leq T \end{cases} \quad (15)$$

Since subproblem (14) is a time-dependant min-cost flow problem, we will define parameters  $A(y(t))$ ,  $C(y(t))$  and  $U(y(t))$  in the residual network  $N(y(t))$ :

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

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

By solving two subproblems 1 and 2 with the relaxation solution  $\mathbf{X}$  and  $\mathbf{Y}$ , the optimal objective value  $Z_{LR}^*$  for the relaxed model (9) with a set of given Lagrangian multiplier vector  $\alpha$

can be expressed as follow:

$$Z_{LR}^* = Z_{SP1}^*(\alpha) + Z_{SP2}^*(\alpha) \quad (16)$$

In conclusion, the optimal objective value of the relaxed model (11) is the lower bound of the optimal objective value of the original model (9). In order to obtain a high-quality solution, it is needed to obtain a lower bound which is close to the optimal objective value of the original model:

$$Z_{LD}(\alpha^*) = \max_{\alpha \geq 0} Z_{LR}(\alpha) \quad (17)$$

### 3.4.2 Successive shortest path algorithm for solving sub problem 1

#### 3.4.2.a Bellman-Ford algorithm

The Bellman-Ford algorithm emulates the shortest paths from a single source vertex to all other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem but more versatile because it can handle graphs with some edge weights that are negative numbers. Alfonso Shimbel proposed the algorithm in 1955, but it is now named after Richard Bellman and Lester Ford Jr., who brought it out in 1958 and 1956. In 1959, Edward F. Moore published a variation of the algorithm, sometimes referred to as the Bellman-Ford–Moore algorithm. The principle of Relaxation of Edges for Bellman-Ford:

- It states that for the graph having **N** vertices, all the edges should be relaxed **N-1** times to compute the single source shortest path.
- In order to detect whether a negative cycle exists or not, relax all the edge one more time and if the shortest distance for any node reduces then we can say that a negative cycle exists. In short if we relax the edges **N** times, and there is any change in the shortest distance of any node between the **N-1th** and **Nth** relaxation than a negative cycle exists, otherwise not exist.

The following pseudocode represent the implementation of Bellman-Ford Algorithm:

---

**Algorithm 1** Bellman-Ford Algorithm

```

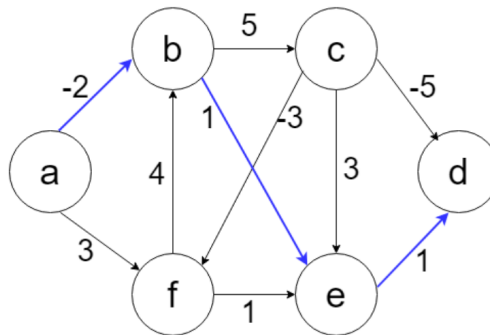
1: procedure BELLMANFORD( $G, a$ )
2:   for all vertices  $v$  do                                     ▷ Initialization Step
3:      $Label[v] := \infty$ 
4:      $Prev[v] := -1$ 
5:   end for
6:    $Label[a] := 0$                                              ▷ a is the source node
7:   for  $i$  from 1 to  $size(vertices) - 1$  do
8:     for all  $v \in V$  do
9:       if  $(Label[u] + Wt(u, v)) < Label[v]$  then
10:         $Label[v] := Label[u] + Wt(u, v)$ 
11:         $Prev[v] := u$ 
12:      end if
13:    end for
14:  end for
15:  for all vertices  $v$  do                                     ▷ Check circuit of negative weight
16:    if  $(Label[u] + Wt(u, v)) < Label[v]$  then
17:      error: "Contains circuit of negative weight"
18:    end if
19:  end for
20: end procedure

```

---

Two example given below will illustrate the Bellman-Ford algorithm:

**Example 1:**

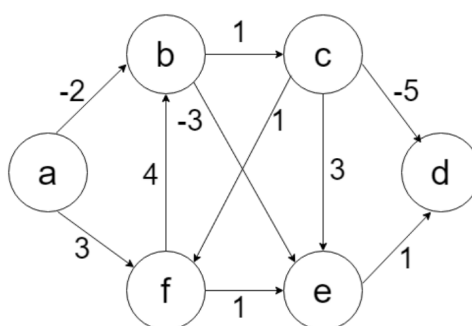


Hình 9: Example 1

Step	a	b	c	d	e	f
0	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
1	0	<b>-2a</b>	$\infty$	$\infty$	$\infty$	3a
2	0	<b>-2</b>	3b	$\infty$	<b>-5b</b>	3
3	0	-2	3	<b>-4e</b>	<b>-5</b>	3
4	0	-2	3	<b>-4</b>	-5	3

This process terminate when step 4 = step 3. The shortest path from a to d: a->b->e->d  
**Example 2:**

There exists a circle of negative length since Step 6  $\neq$  Step 5



Hình 10: Example 2

Step	a	b	c	d	e	f
0	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
1	0	-2a	$\infty$	$\infty$	$\infty$	$\infty$
2	0	-2	-1b	$\infty$	-1b	$\infty$
3	0	-2	-1	-6c	-1	-4c
4	-1f	-2	-1	-6c	-3f	-4
5	-1	-3a	-1	-6	-3	-4
6	-1	-3	-2b	-6	-3	-4
7	-1	-3	-2	-7c	-3	-4

### 3.4.2.b Ford-Fulkerson algorithm

Ford-Fulkerson algorithm is a greedy approach for calculating the maximum possible flow in a network or a graph. A term, flow network, is used to describe a network of vertices and edges with a source (S) and a sink (T). Each vertex, except S and T, can receive and send an equal amount of stuff through it. S can only send and T can only receive stuff.

Now, we will consider the implementation of Ford-Fulkerson algorithm. Let  $G(V, E)$  be the graph, and for each edge from u to v, let  $c(u, v)$  be the capacity and  $f(u, v)$  be the flow. We want to find the maximum flow from source s to the sink t. After every step in the algorithm the following is maintained:

- Capacity constraints:  $\forall (u, v) \in E : f(u, v) \leq c(u, v)$

- Skew symmetry:  $\forall (u, v) \in E : f(u, v) = -f(v, u)$
- Flow conservation  $\forall u \in V : u \neq s \text{ and } u \neq t \Rightarrow \sum_{w \in V} f(u, w) = 0$
- Value(f):  $\sum_{s, u \in E} f(s, u) = \sum_{v, t \in E} f(v, t)$

This means that the flow through the network is a *legal flow* after each round in the algorithm. We define the **residual network**  $G_f(V, E_f)$  to be the network with capacity  $c_f(u, v) = c(u, v) - f(u, v)$  and no flow. Notice that it can happen that a flow from  $v$  to  $u$  is allowed in the residual network, through disallowed in the original network: if  $f(u, v) > 0$  and  $c(u, v) = 0$  then  $c_f(v, u) = c(v, u) - f(v, u) = f(u, v) > 0$ . This is the pseudocode for Ford-Fulkerson algorithm:

---

**Algorithm 2** Ford-Fulkerson Algorithm

---

```

1:  $f(u, v) \leftarrow 0$  for all edges  $(u, v)$ 
2: while there is a path  $p$  from  $s$  to  $t$  in  $G_f$ , such that  $c_f(u, v) > 0$  for all edges  $(u, v) \in p$  do
3:   Find  $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$ 
4:   for each edges  $(u, v) \in p$  do
5:      $f(u, v) \leftarrow f(u, v) + c_f(p)$  (Send flow along the path)
6:      $f(v, u) \leftarrow f(v, u) - c_f(p)$  (The flow might be "returned" later)
7:   end for
8: end while

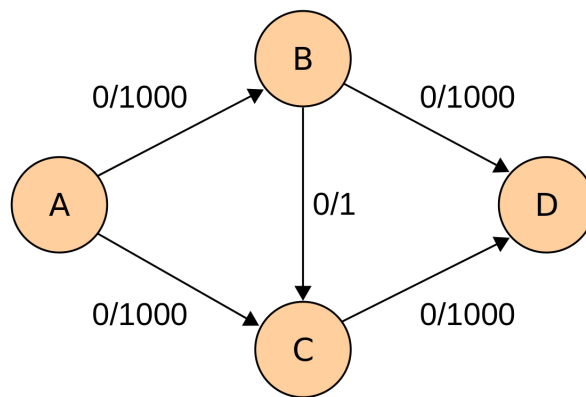
```

---

**Given example of Ford-Fulkerson algorithm is described below:**

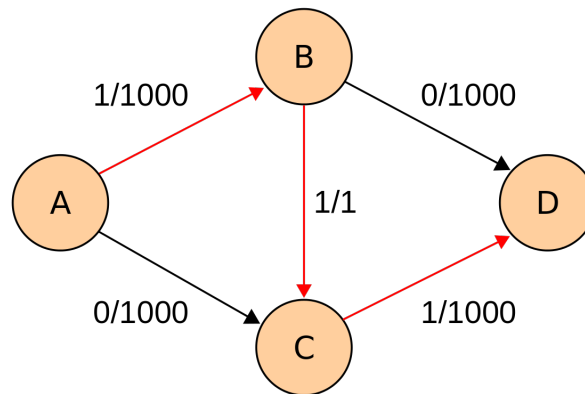
This example shows the first steps of Ford-Fulkerson in a flow network with 4 nodes, source A and sink D. This example shows the worst-case behavior of the algorithm. In each step, only a flow of 1 is sent across the network. If breadth-first-search were used instead, only two steps would be needed.

- Initial flow network:



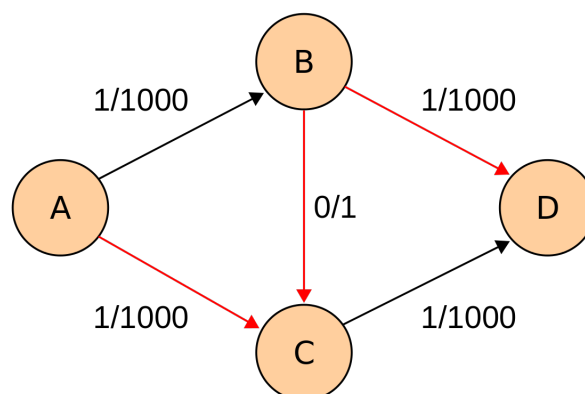
**Iteration 1:**

- Let  $A, B, C, D = \min(c_f(A, B), c_f(B, C), c_f(C, D))$   
 $= \min(c(A, B) - f(A, B), c(B, C) - f(B, C), c(C, D) - f(C, D))$   
 $= \min(1000 - 0, 1 - 0, 1000 - 0) = 1$
- Therefore, we can update the current flow as below:



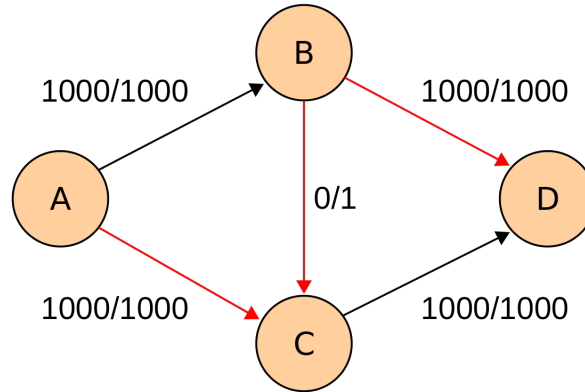
### Iteration 2:

- Let  $A, C, B, D = \min(c_f(A, C), c_f(C, B), c_f(B, D))$   
 $= \min(c(A, C) - f(A, C), c(C, B) - f(C, B), c(B, D) - f(B, D))$   
 $= \min(1000 - 0, 0 - (-1), 1000 - 0) = 1$
- Update the capacities:



- .....
- After 1998 similarly iterations, we can get the final flow network:





### 3.4.2.c Successive shortest path algorithm for min-cost max-flow problem

Given a directed flow network  $G = (V, E)$ , in the standard general formulation of the minimum-cost problem, the followings are additional data:

- A capacity function  $c: E \rightarrow R$  such that  $c(e) \geq 0$  for all  $e \in E$ ;
- A lower bound function  $l: E \rightarrow R$  such that  $0 \leq l(e) \leq c(e)$  for all  $e \in E$ ;
- A balance function  $b: V \rightarrow R$  such that  $\sum_v b(v) = 0$ ;
- A cost function  $\$: E \rightarrow R$

As usual, a feasible flow in  $G$  is a function  $f: E \rightarrow R$  that satisfies the capacity constraints  $l(e) \leq f(e) \leq c(e)$  at every edge  $e$  and the balance constraints  $\sum_u f(u \rightarrow v) - \sum_w f(v \rightarrow w) = b(v)$  at every vertex  $v$ . The minimum-cost flow problem asks for a feasible flow  $f$  with minimum total cost  $\$(f) = \sum_e \$(e) \cdot f(e)$ . In terms of minimum-cost flow problem, not only the two conditions above, a pseudoflow  $\psi: E \rightarrow R$  also satisfies that the residual graph  $G_\psi$  contains no negative-cost cycles.

In the successive shortest path algorithm, two simplifying assumptions are required to initialize the flow network  $G$ :

- **All lower bounds are zero.** Otherwise, think of the lower-bound function  $l$  as a pseudoflow (sending  $l(e)$  units of flow across each edge  $e$ ), and consider the residual graph  $G_l$ .
- **All costs are non-negative.** Otherwise, consider the residual graph of the following pseudoflow:

$$\bar{c}(u \rightarrow v) = \begin{cases} c(u \rightarrow v), \$(u \rightarrow v) \geq 0 \\ 0, \$(u \rightarrow v) < 0 \end{cases}$$

Enforcing these assumptions may introduce non-zero balances at the vertices, even if all balances in the initial flow network are zero. These two assumptions imply that the all-zero pseudoflow is both feasible and locally optimal, but not necessarily balanced.

The successive shortest path algorithm is described below:

(To simplify the pseudocode, assume implicitly that the residual graph  $G_\psi$  is always strongly connected. If necessary, this assumption can be enforced by adding a new vertex with zero balance and with infinite-cost edges to and from every other vertex in  $G$ . If SuccessiveShortestPaths ever tries to push flow along one of these new edges, we can immediately report that the original flow problem is infeasible.)

---

**Algorithm 3** Successive Shortest Paths Algorithm

---

```
1: procedure SUCCESSIVESHORTESTPATHS( $V, E, c, b, \$$ )
2:   for each  $e \in E$  do
3:      $\psi(e) \leftarrow 0$ 
4:   end for
5:    $B \leftarrow \sum_v |b(v)|/2$ 
6:   while  $B > 0$  do
7:     construct  $G_\psi$ 
8:      $s \leftarrow$  any vertex with  $b_\psi(s) < 0$ 
9:      $t \leftarrow$  any vertex with  $b_\psi(t) > 0$ 
10:     $\sigma \leftarrow$  shortest path in  $G_\psi$  from  $s$  to  $t$ 
11:    AUGMENT( $s, t, \sigma$ )
12:  end while
13:  return  $\psi$ 
14: end procedure
15: procedure AUGMENT( $s, t, \sigma$ )
16:    $R \leftarrow \min\{-b_\psi(s), b_\psi(t), \min_{e \in \sigma} c_\psi(e)\}$ 
17:    $R \leftarrow \min$ 
18:    $B \leftarrow B - R$ 
19:   for each (directed edge  $e \in \sigma$ ) do
20:     if  $e \in E$  then
21:        $\psi(e) \leftarrow \psi(e) + R$ 
22:     else
23:        $\psi(e) \leftarrow \psi(e) - R$ 
24:     end if
25:   end for
26: end procedure
```

---

To be more specific for comprehensive understanding, the successive shortest path begins by initializing  $\psi$  to the all-zero pseudoflow as well as set  $B$  to the value of half of total absolute value of balance function  $b$  of every vertices. Next, conduct a loop until value in  $B$  is less than or equal to zero. In each iteration, construct  $G_\psi$  (respected to  $E_\psi$  to cover only edges that has capacity  $c_\psi > 0$ ); set  $s$  to any vertex with  $b_\psi(s) < 0$  and set  $t$  to any vertex with  $b_\psi(t) > 0$ . By using Bellman-ford algorithm respect to all vertices in  $G_\psi$  that has been mentioned above, find the shortest path from  $s$  to  $t$ , which is defined as a path that has the smallest sum of the residual costs of its edges and then set  $\sigma$  to the result. After that, call Augment procedure with respect to  $s, t$  and  $\sigma$ . In Augment algorithm, define  $R$  as min of the set of values including capacity of every edges  $e$  in  $\sigma$ , absolute value of  $b_\psi(s)$  and  $b_\psi(t)$ . Following that, subtract value in  $B$  by  $R$  and redefine pseudoflow  $\psi$  in every edges  $e$  in  $\sigma$  by add them with  $R$  and subtract them by  $R$  otherwise.

### Analysis on Successive shortest path algorithm

The successive shortest path algorithm share resemblances to Ford-Fulkerson algorithm. However, while Ford-Fulkerson algorithm choose an arbitrary path in each iteration, the successive shortest path choose a shortest one by applying Bellman-Ford shortest path algorithm. (Actually, we can use Dijkstra's algorithm to compute the first shortest path, but augmenting along a path of positive-cost edges introduces residual edges with negative costs, so we can't use Dijkstra's algorithm in later iterations.).

Assuming the capacity of each edge is an integer, then each augmentation decreases the total absolute balance  $B$  by a positive integer, and therefore by at least 1. It follows that the cycle canceling algorithm halts after at most  $B$  iterations and thus runs in  $O(VEB)$  time. As one might expect from Ford-Fulkerson, this time bound is exponential in the complexity of the input when capacities are integers, the time bound are tight in the worst case, and the algorithm may not terminate or even approach a minimum-cost circulation in the limit if any capacities are irrational.

Nevertheless, the algorithm either returns a feasible flow (that is, a feasible and balanced pseudoflow) or reports correctly that no such flow exists if and only if Shortest paths are only well-defined in graphs without negative cycles! To argue that SuccessiveShortestPaths is even a well-defined algorithm, much less a correct algorithm, we need to prove the following lemma:

**Lemma:** After every iteration of the main loop of SuccessiveShortestPaths, the feasible pseudoflow  $\psi$  is locally optimal; that is, the residual graph  $G_\psi$  contains no negative cycles.

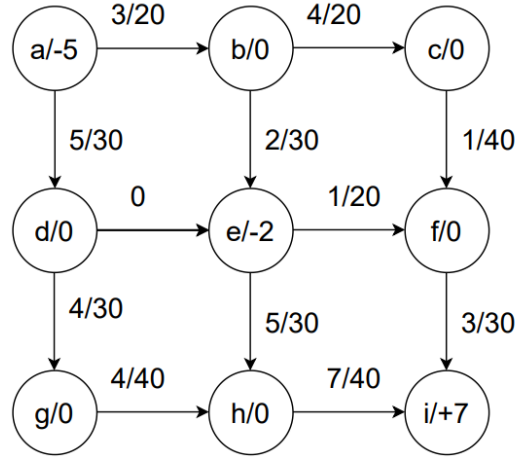
**Prove:** Suppose that  $\psi$  is locally optimal at the beginning of an iteration of the main loop. By definition, there are no negative cycles in the residual graph  $G_\psi$ , so shortest paths are well-defined. Let  $s$  be an arbitrary supply vertex and let  $t$  be an arbitrary demand vertex in  $G_\psi$ . Let  $\sigma$  be a shortest path in  $G_\psi$ , and let  $\psi_0$  be the resulting pseudoflow after augmenting along  $\sigma$ . We need to prove that  $\psi_0$  is locally optimal.

For the sake of argument, suppose the residual graph  $G_\psi$  contains a negative cycle  $\gamma$ . Let  $f = \sigma + \gamma$  denote the pseudoflow in  $G_\psi$  obtained by sending one unit of flow along  $\gamma$  and then one unit of flow along  $\sigma$ . Although  $\gamma$  may contain edges that are not in  $G_\psi$ , all such edges must be reversals of edges in  $\sigma$ ; thus,  $f$  is positive only on edges in  $G_\psi$ . Routine calculations imply that  $f$  is an integral  $(s, t)$ -flow in  $G_\psi$  with value 1 ( $f$  may not be feasible in  $G_\psi$ ). Thus, by the Flow Decomposition Lemma,  $f$  can be decomposed into a single path  $\sigma_0$  from  $s$  to  $t$  and possibly one or more cycles  $\gamma_1, \dots, \gamma_k$ . Our assumption that  $\psi$  is locally optimal implies that each cycle  $\psi_i$  has non-negative cost, and therefore  $\$(\sigma_0) \leq \$(f) \leq \$(\sigma)$ . But this is impossible, because  $\sigma$  is a shortest path from  $s$  to  $t$ .

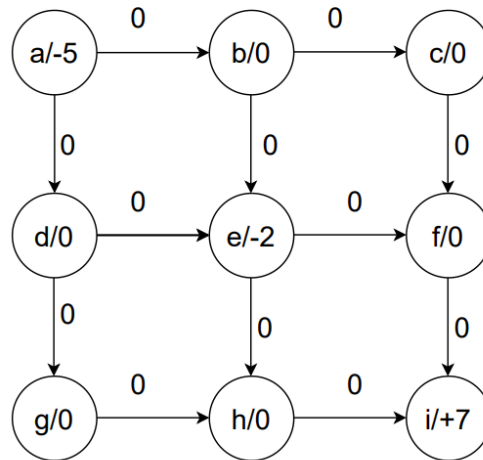
### Example of successive shortest path algorithm:

In order to verify the correctness of the successive shortest path algorithm, we construct the example of 9 nodes and 12 links. Each node contains one value that represents the balance function  $b$ . On each edge, there are two parameter that respectively symbolize for capacity and time travel, which is set up in advance ( for example, 3/40 indicates that this edge has capacity 3 and cost 40 for time travel ). To simplify for understanding, there are two node that have balance function  $b$  less than 0 and one node that has balance function  $b$  greater than 0, others are equal to 0.

The experiment example with more description is given below:



We initialize the same graph that represent the pseudoflow as well as assigning 0 to all value of each edge:



Initially, calculate  $B = \sum_v |b(v)| / 2 = 7$

**Iteration 1**

- Let 
$$\begin{cases} B = 7 \\ s \leftarrow \text{node}(e), b_\psi(s) = -2 \\ t \leftarrow \text{node}(i), b_\psi(t) = +7 \end{cases}$$

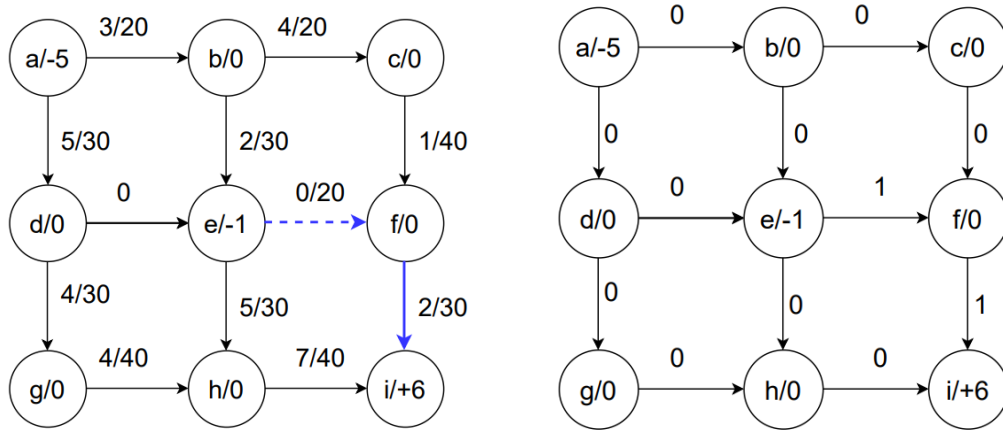
- Re-construct  $G_\psi$  with respect to  $E_\psi : E = \{e \in E \mid c_\psi(e) > 0\}$

- Applying Bellman-Ford algorithm, we find the shortest path from  $s \rightarrow t$  is:  $e \rightarrow f \rightarrow i$  (length of path is 50)

- Calculate  $\begin{cases} R = \min\{-b_\psi(s), b_\psi(t), \min_{e \in S} c_\psi(e)\} = \min\{-(-2), 7, 1\} = 1 \\ B = B - R = 7 - 1 = 6 \end{cases}$
- Update  $b(s), b(t)$  base on  $R = 1$  and for each  $e \in \sigma$ , we compute that:

$$\begin{cases} \psi(e) \leftarrow \psi(e) + 1; c_\psi(e) = c_\psi(e) - 1, e \in E \\ \psi(e) \leftarrow \psi(e) - 1; c_\psi(e) = c_\psi(e) + 1, \text{otherwise} \end{cases}$$

The update graph  $G_\psi$  and pseudoflow  $\psi$  is given below:



Hình 11: Update graph  $G_\psi$  and pseudoflow  $\psi$

- Total travel time cost: 50

### Iteration 2

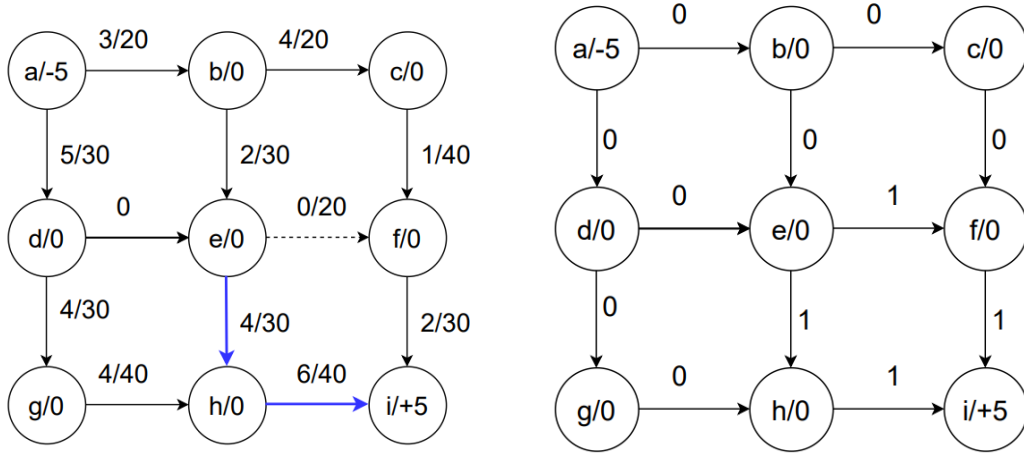
- Let  $\begin{cases} B = 6 \\ s \leftarrow \text{node}(e), b_\psi(s) = -1 \\ t \leftarrow \text{node}(i), b_\psi(t) = +6 \end{cases}$
- Re-construct  $G_\psi$  with respect to  $E_\psi : E_\psi = \{e \in E \mid c_\psi(e) > 0\}$

- Applying Bellman-Ford algorithm, we find the shortest path from  $s \rightarrow t$  is:  $e \rightarrow h \rightarrow i$  (length of path is 70)

- Calculate  $\begin{cases} R = \min\{-b_\psi(s), b_\psi(t), \min_{e \in S} c_\psi(e)\} = \min\{-(-1), 6, 5\} = 1 \\ B = B - R = 6 - 1 = 5 \end{cases}$
- Update  $b(s), b(t)$  base on  $R = 1$  and for each  $e \in \sigma$ , we compute that:

$$\begin{cases} \psi(e) \leftarrow \psi(e) + 1; c_\psi(e) = c_\psi(e) - 1, e \in E \\ \psi(e) \leftarrow \psi(e) - 1; c_\psi(e) = c_\psi(e) + 1, \text{otherwise} \end{cases}$$

The update graph  $G_\psi$  and pseudoflow  $\psi$  is given below:



Hình 12: Update graph  $G_\psi$  and pseudoflow  $\psi$

- Total travel time cost: 70

### Iteration 3

- Let 
$$\begin{cases} B = 5 \\ s \leftarrow \text{node}(a), b_\psi(s) = -5 \\ t \leftarrow \text{node}(i), b_\psi(t) = +5 \end{cases}$$

- Re-construct  $G_\psi$  with respect to  $E_\psi : E = \{e \in E \mid c_\psi(e) > 0\}$

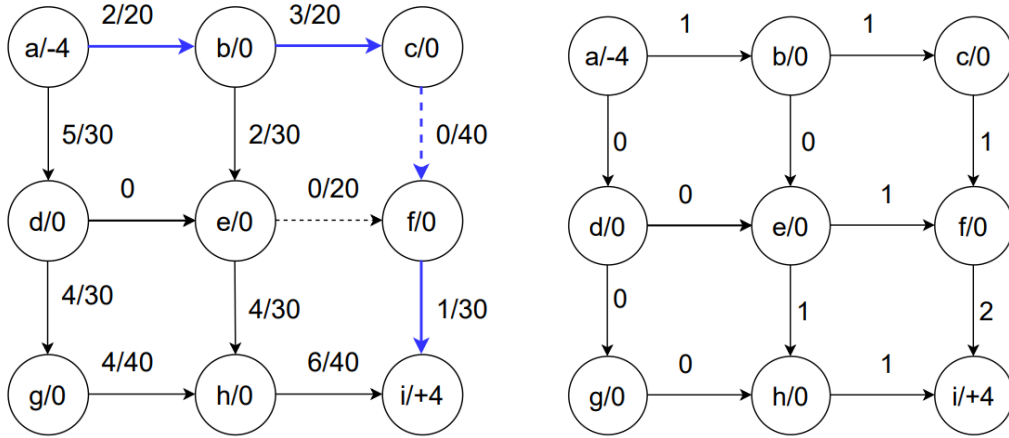
- Applying Bellman-Ford algorithm, we find the shortest path from  $s \rightarrow t$  is:  $a \rightarrow b \rightarrow c \rightarrow f \rightarrow i$  (length of path is 110)

- Calculate 
$$\begin{cases} R = \min\{-b_\psi(s), b_\psi(t), \min_{e \in S} c_\psi(e)\} = \min\{-(-5), 5, 1\} = 1 \\ B = B - R = 5 - 1 = 4 \end{cases}$$

- Update  $b(s), b(t)$  base on  $R = 1$  and for each  $e \in \sigma$ , we compute that:

$$\begin{cases} \psi(e) \leftarrow \psi(e) + 1; c_\psi(e) = c_\psi(e) - 1, e \in E \\ \psi(e) \leftarrow \psi(e) - 1; c_\psi(e) = c_\psi(e) + 1, \text{otherwise} \end{cases}$$

The update graph  $G_\psi$  and pseudoflow  $\psi$  is given below:



Hình 13: Update graph  $G_\psi$  and pseudoflow  $\psi$

- Total travel time cost: 110

#### Iteration 4

- Let 
$$\begin{cases} B = 4 \\ s \leftarrow \text{node}(a), b_\psi(s) = -4 \\ t \leftarrow \text{node}(i), b_\psi(t) = +4 \end{cases}$$

- Re-construct  $G_\psi$  with respect to  $E_\psi : E = \{e \in E \mid c_\psi(e) > 0\}$

- Applying Bellman-Ford algorithm, we find the shortest path from  $s \rightarrow t$  is:  $a \rightarrow b \rightarrow e \rightarrow h \rightarrow i$  (length of path is 120)

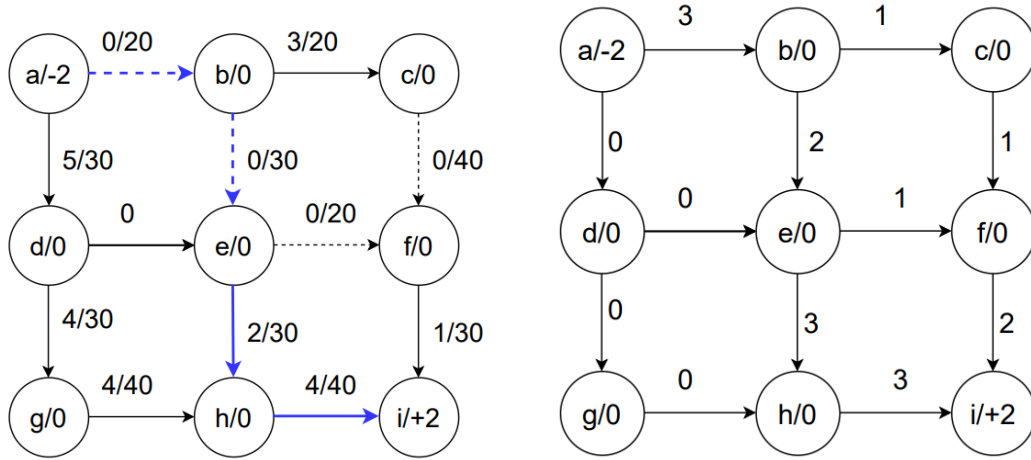
- Calculate 
$$\begin{cases} R = \min\{-b_\psi(s), b_\psi(t), \min_{e \in S} c_\psi(e)\} = \min\{-(-4), 4, 2\} = 2 \\ B = B - R = 4 - 2 = 2 \end{cases}$$

- Update  $b(s), b(t)$  base on  $R = 2$  and for each  $e \in \sigma$ , we compute that:

$$\begin{cases} \psi(e) \leftarrow \psi(e) + 1; c_\psi(e) = c_\psi(e) - 1, e \in E \\ \psi(e) \leftarrow \psi(e) - 1; c_\psi(e) = c_\psi(e) + 1, \text{otherwise} \end{cases}$$

The update graph  $G_\psi$  and pseudoflow  $\psi$  is given below:

- Total travel time cost: 120



Hình 14: Update graph  $G_\psi$  and pseudoflow  $\psi$

#### Iteration 5

- Let  $\begin{cases} B = 2 \\ s \leftarrow \text{node}(a), b_\psi(s) = -2 \\ t \leftarrow \text{node}(i), b_\psi(t) = +2 \end{cases}$
- Re-construct  $G_\psi$  with respect to  $E_\psi : E = \{e \in E \mid c_\psi(e) > 0\}$
- Applying Bellman-Ford algorithm, we find the shortest path from  $s \rightarrow t$  is:  $a \rightarrow d \rightarrow g \rightarrow h \rightarrow i$  (length of path is 140)
- Calculate  $\begin{cases} R = \min\{-b_\psi(s), b_\psi(t), \min_{e \in S} c_\psi(e)\} = \min\{-(-2), 2, 4\} = 2 \\ B = B - R = 2 - 2 = 0 \end{cases}$
- Update  $b(s), b(t)$  base on  $R = 2$  and for each  $e \in \sigma$ , we compute that:

$$\begin{cases} \psi(e) \leftarrow \psi(e) + 1; c_\psi(e) = c_\psi(e) - 1, e \in E \\ \psi(e) \leftarrow \psi(e) - 1; c_\psi(e) = c_\psi(e) + 1, \text{otherwise} \end{cases}$$

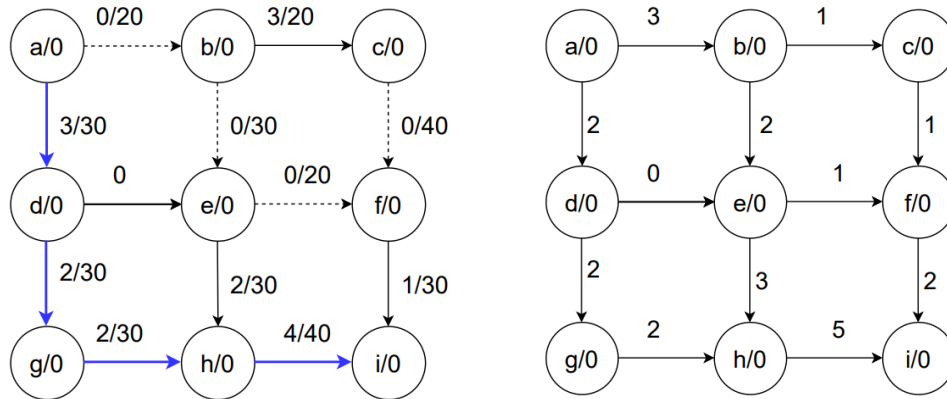
The update graph  $G_\psi$  and pseudoflow  $\psi$  is given below:

- Total travel time cost: 140

When  $B = 0$ , the algorithm terminates.

**In conclusion**, the total travel time cost is the max travel time cost in every iterations, which is equal to 140.





Hình 15: Update graph  $G_\psi$  and pseudoflow  $\psi$

## 3.5 Experimental Design

### 3.5.1 Experiment network generation

This section will proposed a way to generate data and implement a small grid network in Python, but not to implement successive path algorithm or making any computation. We will implement a class Model to generate a small grid network with 50 nodes and 95 links. For simplicity, we will consider 10 scenarios that can be applied to the network after the time threshold  $\bar{T}$

First, we need to include efficient library serving for the implementation

```
1 import numpy as np
2 import networkx as nx
3 import random as rd
4 import matplotlib.pyplot as plt
5
```

Listing 1: Package included

NumPy (Numerical Python) is an open source Python library that worki with numerical data in Python, especially array and vector. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. This library is suitable for our transportation network. Random is an in-built module of Python that is used to generate random numbers in Python, which can be used to generate data at priory stage and generate scenarios. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Now we will consider the prototype of our network model:

```
1 class Model:
2     '''
3     This class generate a grid model for evacuation planning
4     '''
5     def __init__(self):
6
7     def generate_grid(self):
8
9     def drawing(self):
10
```

```
11 def generate_scenario(self, num_scenario = 10):
12
13 def change_to_scenario(self, idx : int):
14
```

Listing 2: Python example

We will consider the constructor of this model:

```
1 def __init__(self):
2     self.G = nx.DiGraph()
3     self.edges = []
4     self.generated = False
5     self.list_threshold = []
6     self.scenarios = []
```

Listing 3: Python example

As can be observed, the init method will initialize 4 attributes for our Model object: self.G is a DiGraph object belong to networkx library, self.edges is a list contains the pair of vertex to perform an edge, self.generated is a boolean variable to indicate if the model has generated data or not, obviously its initial value will be False, self.scenarios is a list contains 10 scenarios may applied to our model.

```
1 def generate_grid(self):
2     #This function generate edges pair for the grid
3     def generate_adjacent_pairs(rows, cols):
4         pairs = []
5         for row in range(1, rows + 1):
6             for col in range(1, cols + 1):
7                 current_node = col + (row - 1) * cols
8                 right_node = col + 1 + (row - 1) * cols
9                 bottom_node = col + row * cols
10                if col == cols:
11                    right_node = 0
12                if row == rows:
13                    bottom_node = 0
14                if right_node != 0:
15                    pairs.append((current_node, right_node))
16                if bottom_node != 0:
17                    pairs.append((current_node, bottom_node))
18            return pairs
19
20    #Initialize capacity and travel time in the grid
21    pairs = generate_adjacent_pairs(5, 10)
22    for i in range(len(pairs)):
23        self.G.add_edge(*pairs[i], time_travel = rd.randrange(3,9), capacity = rd.
24        randrange(10, 20))
25    #Set demand in each node
26    nodes_demand = {}
27    nodes_demand[1] = 600
28    nodes_demand[50] = -600
29    for i in range(2, 50):
30        nodes_demand[i] = 0
31    nx.set_node_attributes(self.G, nodes_demand, name='demand')
32    self.generated = True
33    self.edges = pairs
```

Listing 4: Python example

The generate\_grid(self) method will generate the network with random initial values. In this method, there is *generate\_adjacent\_pairs(rows, cols)* method that generate the pairs of vertices used to represent the edges.

```
1 def drawing(self):
2     if not self.generated:
3         self.generate_grid()
4         based_node_pos = {1 : (1, 5), 11 : (1, 4), 21 : (1, 3), 31 : (1, 2), 41 :
5         (1, 1)}
6         node_pos = dict(based_node_pos)
7         for i in range(1, 10):
8             for key in based_node_pos:
9                 x, y = based_node_pos[key]
10                node_pos[key + i] = tuple((x + i, y))
11
12                label1 = nx.get_edge_attributes(self.G, 'capacity')
13                nx.draw(self.G, node_pos, with_labels = True, node_color = 'green',
14                font_color= 'whitesmoke', font_size = 10)
15                nx.draw_networkx_edge_labels(self.G, node_pos, edge_labels=label1)
16                plt.show()
```

Listing 5: Python example

The generate\_scenario(self) method will generate 10 scenario randomly for self.scenarios.

```
1 def generate_scenario(self, num_scenario = 10):
2     #This function generate 10 random scenario for the 2nd stage
3     for i in range(num_scenario):
4         random_scenario = np.random.randint(10, 20, size=85)
5         self.scenarios.append(random_scenario)
6     return
```

Listing 6: Python example

The change\_to\_scenario(self, idx) will choose the scenario with index idx in self.scenarios and apply that scenario to the network.

```
1 def change_to_scenario(self, idx : int):
2     if len(self.scenarios) == 0:
3         self.generate_scenario()
4     elif not self.generated:
5         self.generate_grid()
6     chosen_scenario = self.scenarios[idx]
7     dictionary = dict({self.edges[i] : dict({"capacity" : chosen_scenario[i]}) for
8     i in range(len(self.edges))})
9     nx.set_edge_attributes(self.G, dictionary)
```

Listing 7: Python example

By combine all the methods above, we get the complete class model below:

```
1 import numpy as np
2 import networkx as nx
3 import random as rd
4
5 class Model:
6     '''
7     This class generate a grid model for evacuation planning
8     '''
9     def __init__(self):
10         self.G = nx.DiGraph()
11         self.edges = []
12         self.generated = False
13         self.list_thredhold = []
14         self.scenarios = []
15         self.solutions = []
16
17         #This function generate the grid for the model
```

```
18 def generate_grid(self):
19     #This function generate edges pair for the grid
20     def generate_adjacent_pairs(rows, cols):
21         pairs = []
22         for row in range(1, rows + 1):
23             for col in range(1, cols + 1):
24                 current_node = col + (row - 1) * cols
25                 right_node = col + 1 + (row - 1) * cols
26                 bottom_node = col + row * cols
27                 if col == cols:
28                     right_node = 0
29                 if row == rows:
30                     bottom_node = 0
31                 if right_node != 0:
32                     pairs.append((current_node, right_node))
33                 if bottom_node != 0:
34                     pairs.append((current_node, bottom_node))
35         return pairs
36
37     #Initialize capacity and travel time in the grid
38     pairs = generate_adjacent_pairs(5, 10)
39     for i in range(len(pairs)):
40         self.G.add_edge(*pairs[i], time_travel = rd.randrange(3,9), capacity =
rd.randrange(10, 20))
41     #Set demand in each node
42     nodes_demand = {}
43     nodes_demand[1] = 600
44     nodes_demand[50] = -600
45     for i in range(2, 50):
46         nodes_demand[i] = 0
47     nx.set_node_attributes(self.G, nodes_demand, name='demand')
48     self.generated = True
49     self.edges = pairs
50
51 def drawing(self):
52     if not self.generated:
53         self.generate_grid()
54     based_node_pos = {1 : (1, 5), 11 : (1, 4), 21 : (1, 3), 31 : (1, 2), 41 :
(1, 1)}
55     node_pos = dict(based_node_pos)
56     for i in range(1, 10):
57         for key in based_node_pos:
58             x, y = based_node_pos[key]
59             node_pos[key + i] = tuple((x + i, y))
60
61     label1 = nx.get_edge_attributes(self.G, 'capacity')
62     nx.draw(self.G, node_pos, with_labels = True, node_color = 'green',
font_color= 'whitesmoke', font_size = 10)
63     nx.draw_networkx_edge_labels(self.G, node_pos, edge_labels=label1)
64     plt.show()
65
66 def generate_scenario(self, num_scenario = 10):
67     #This function generate 10 random scenario for the 2nd stage
68     for i in range(num_scenario):
69         random_scenario = np.random.randint(10, 20, size=85)
70         self.scenarios.append(random_scenario)
71     return
72
73 def change_to_scenario(self, idx : int):
74     if len(self.scenarios) == 0:
75         self.generate_scenario()
76     elif not self.generated:
```

```

77         self.generate_grid()
78         chosen_scenario = self.scenarios[idx]
79         dictionary = dict({self.edges[i] : dict({"capacity" : chosen_scenario[i]})
80         for i in range(len(self.edges))})
81         nx.set_edge_attributes(self.G, dictionary)
82

```

Listing 8: Python example

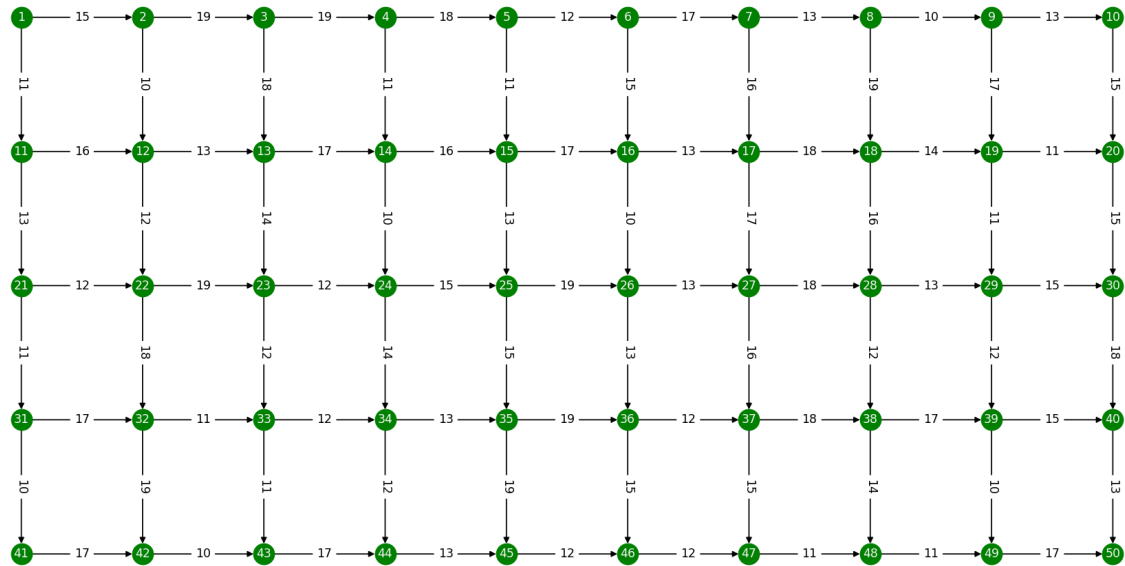
We will initialize the model and drawing the network:

```

1 x = Model()
2 x.drawing()

```

We get the output:



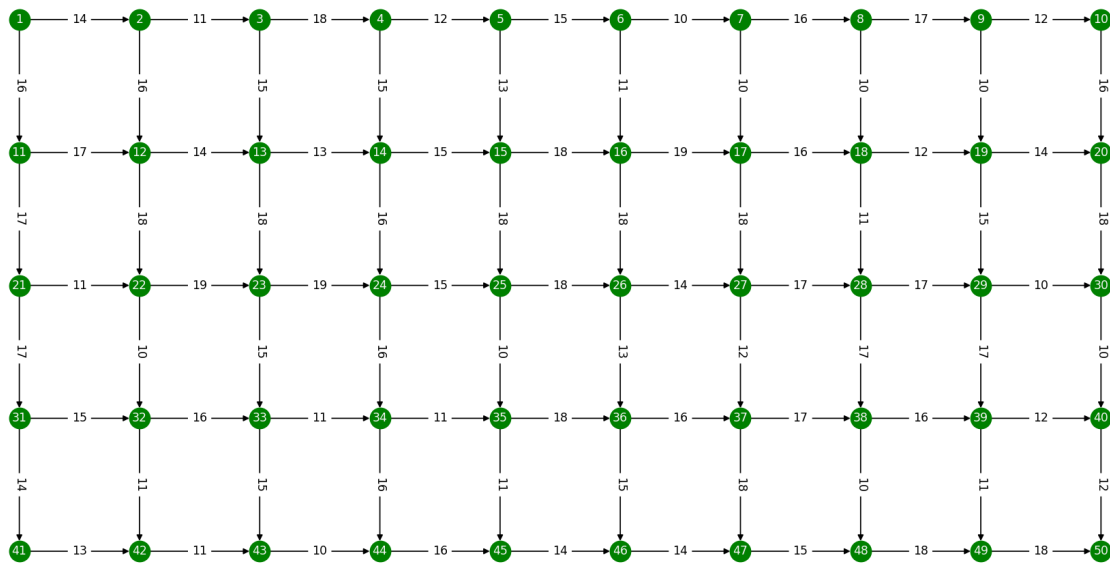
Noted that the number on each edge is the capacity of that link. If we want to change the network to scenario 4, we will use the statement:

```

1 x.change_to_scenario(4)
2 x.drawing()

```

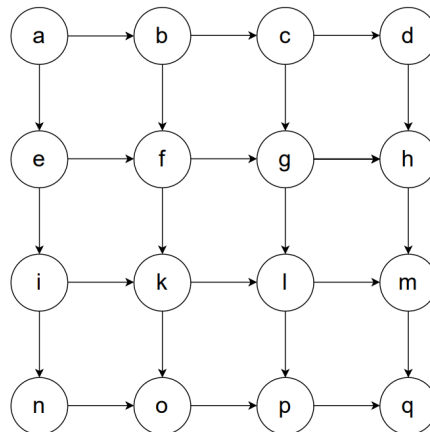
We will get the output:



### 3.5.2 Solving problem with two-stage stochastic programming using min-cost flow algorithm

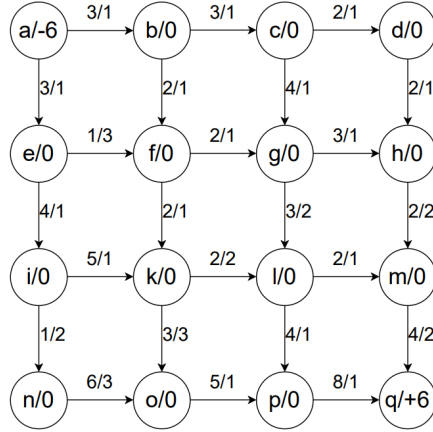
In this part, we will apply min-cost flow algorithm on a 4x4 grid, considering two stage: priori and 2nd stage (subproblem 1 and 2).

To simplify, we solve the two-stage stochastic problem in small grid 4x4:



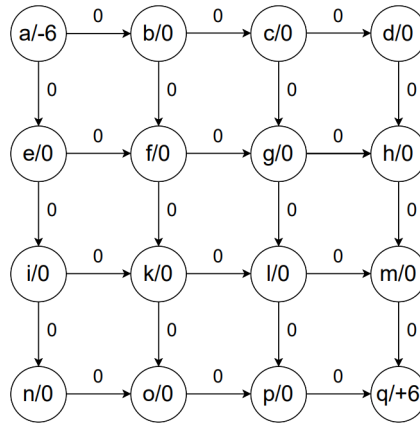
Suppose that there are 6 cars in node a needed to be evacuated to a safe area in node q, and the duration of the disaster is 10 minutes. The unit growth interval is one minute, and thus the disaster period is divided into 10 intervals. In the data initialization phase, according to the link length, the corresponding scenario-based time-dependent link travel times and traffic capacities are randomly generated. In each node, there is a number representing the balance function  $b$ , in this case is the number of cars needed to be evacuated in this node, and in each link, there

are two parameters which representing the traffic capacities and the time travel, respectively, separated by '/'. The grid with more specific description are given below:



### Stage 1: Priory evacuation plan

Firstly, We initialize the same graph that represent the pseudoflow  $\psi$  as well as assigning 0 to all value  $\psi$  of each edge:



Initially, calculate  $B = \sum_v |b(v)| / 2 = 6$

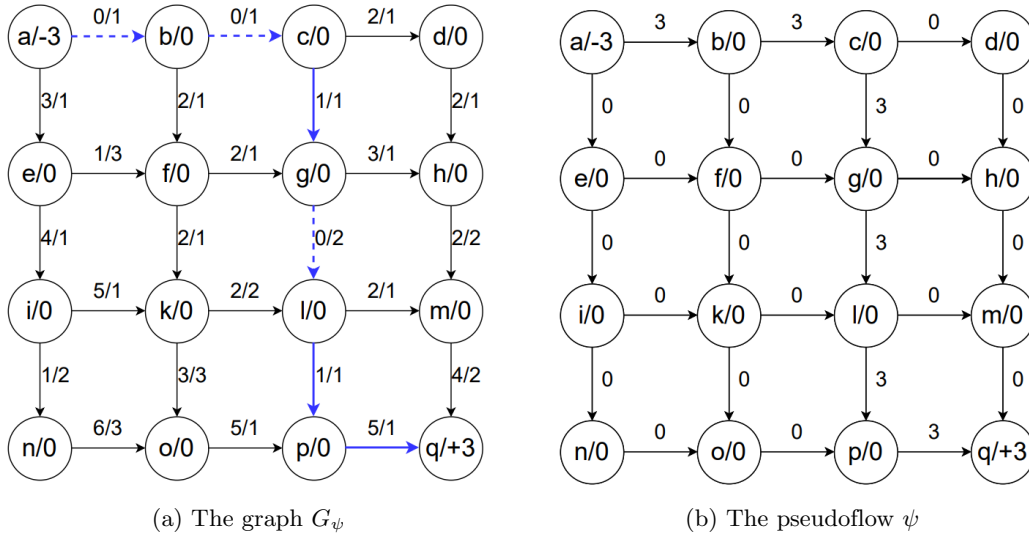
#### Iteration 1

- Let  $\begin{cases} B = 6 \\ s \leftarrow \text{node}(a), b_\psi(s) = -6 \\ t \leftarrow \text{node}(i), b_\psi(t) = +6 \end{cases}$
- Re-construct  $G_\psi$  with respect to  $E_\psi : E = \{e \in E \mid c_\psi(e) > 0\}$ .
- Applying Bellman-Ford algorithm, we find the shortest path from  $s \rightarrow t$  is:  $a \rightarrow b \rightarrow c \rightarrow g \rightarrow l \rightarrow p \rightarrow q$  (length of path is 7)
- Calculate  $\begin{cases} R = \min\{-b_\psi(s), b_\psi(t), \min_{e \in S} c_\psi(e)\} = \min\{-(-6), 6, 3\} = 3 \\ B = B - R = 6 - 3 = 3 \end{cases}$

- Update  $b(s), b(t)$  base on  $R = 3$  and for each  $e \in \sigma$ , we compute that:

$$\begin{cases} \psi(e) \leftarrow \psi(e) + 3; c_\psi(e) = c_\psi(e) - 3, e \in E \\ \psi(e) \leftarrow \psi(e) - 3; c_\psi(e) = c_\psi(e) + 3, \text{otherwise} \end{cases}$$

The update graph  $G_\psi$  and pseudoflow  $\psi$  is given below:



- Total travel time is: 7 minutes.

#### Iteration 2

- Let  $\begin{cases} B = 3 \\ s \leftarrow \text{node}(a), b_\psi(s) = -3 \\ t \leftarrow \text{node}(i), b_\psi(t) = +3 \end{cases}$

- Re-construct  $G_\psi$  with respect to  $E_\psi : E = \{e \in E \mid c_\psi(e) > 0\}$ .

- Applying Bellman-Ford algorithm, we find the shortest path from  $s \rightarrow t$  is:  $a \rightarrow e \rightarrow i \rightarrow k \rightarrow o \rightarrow p \rightarrow q$  (length of path is 8)

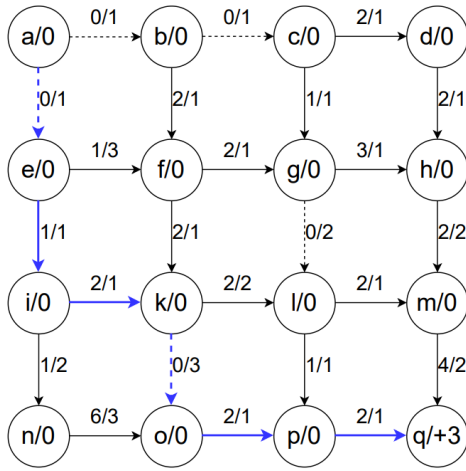
- Calculate  $\begin{cases} R = \min\{-b_\psi(s), b_\psi(t), \min_{e \in S} c_\psi(e)\} = \min\{-(-3), 3, 3\} = 3 \\ B = B - R = 3 - 3 = 0 \end{cases}$

- Update  $b(s), b(t)$  base on  $R = 3$  and for each  $e \in \sigma$ , we compute that:

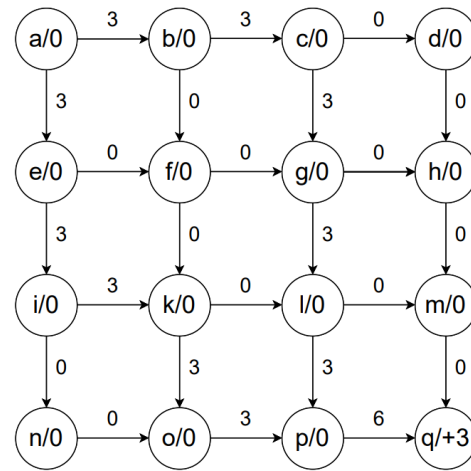
$$\begin{cases} \psi(e) \leftarrow \psi(e) + 3; c_\psi(e) = c_\psi(e) - 3, e \in E \\ \psi(e) \leftarrow \psi(e) - 3; c_\psi(e) = c_\psi(e) + 3, \text{otherwise} \end{cases}$$

The update graph  $G_\psi$  and pseudoflow  $\psi$  is given below:





(a) The graph  $G_\psi$



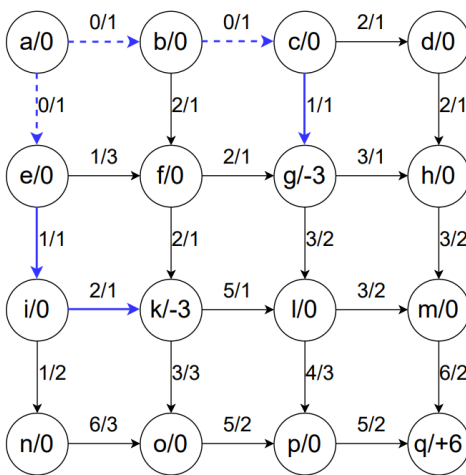
(b) The pseudoflow  $\psi$

- Total travel time is: 8 minutes.
- It is terminated when  $B=0$ .

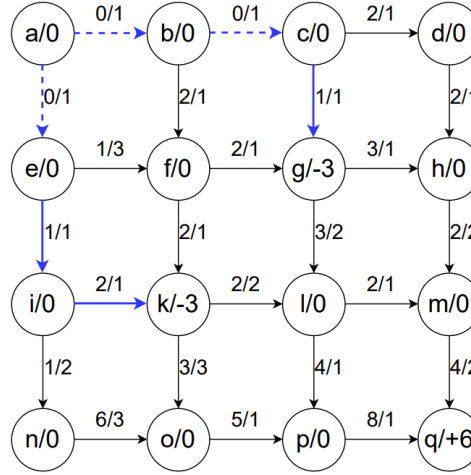
In the first stage, we have the priority evacuation plan:  $a \rightarrow b \rightarrow c \rightarrow g \rightarrow l \rightarrow p \rightarrow q$  and  $a \rightarrow e \rightarrow i \rightarrow k \rightarrow o \rightarrow p \rightarrow q$  to move all 6 cars from node a to node q (safe area), each of path have flow value of 3.

### Stage 2: Time-dependent with threshold T

In the post-event period, assume that the accurate road information is received, after the time threshold  $T=3$ , the latter part of the plan may be changed adaptively. As the threshold, the adaptive plan and the priority plan share resemblances before threshold T, for simplicity, two random scenarios are taken into account to illustrate the planning generation, which is described specifically below:



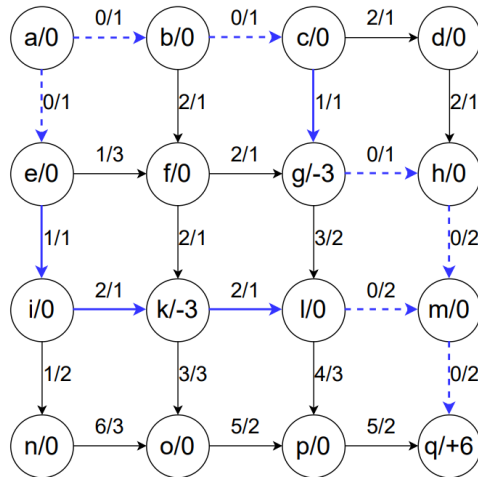
(a) The graph  $G_\psi$  - Scenario 1



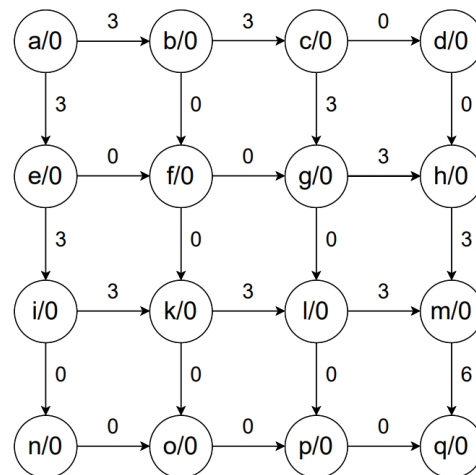
(b) The pseudoflow  $\psi$  - Scenario 2

Solve similarly in stage 1, we can find the adaptive plan for each scenario after threshold  $T=3$ , which are described below:

- Adaptive plan of scenario 1:



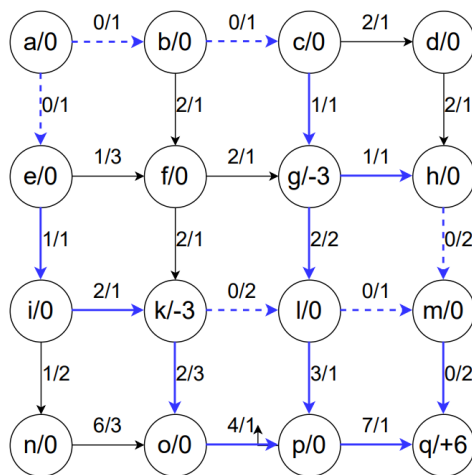
(a) The graph  $G_\psi$  - Scenario 1



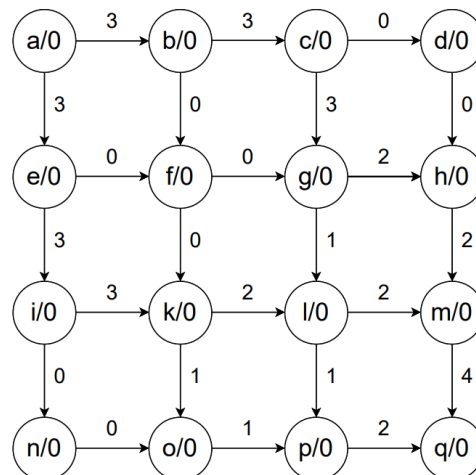
(b) The pseudoflow  $\psi$  - Scenario 1

\* For the adaptive plan in scenario 1, after threshold T, the path from node g to node q (safe area) has turned into:  $a \rightarrow b \rightarrow c \rightarrow g \rightarrow h \rightarrow m \rightarrow q$  compared to :  $a \rightarrow b \rightarrow c \rightarrow g \rightarrow l \rightarrow p \rightarrow q$  in the priory plan, while the path from node k to node q (safe area) has turned into:  $a \rightarrow e \rightarrow i \rightarrow k \rightarrow l \rightarrow m \rightarrow q$  compared to:  $a \rightarrow e \rightarrow i \rightarrow k \rightarrow o \rightarrow p \rightarrow q$  in the priory plan.

- Adaptive plan of scenario 2:



(a) The graph  $G_\psi$  - Scenario 2



(b) The pseudoflow  $\psi$  - Scenario 2

\* For the adaptive plan in scenario 1, after threshold T, the path from node g to node q (safe area) has turned into two separated paths:  $a \rightarrow b \rightarrow c \rightarrow g \rightarrow h \rightarrow m \rightarrow q$  (two cars) and  $a \rightarrow b \rightarrow c \rightarrow g \rightarrow l \rightarrow m \rightarrow q$  (1 cars) compared to one path:  $a \rightarrow b \rightarrow c \rightarrow g \rightarrow l \rightarrow p \rightarrow q$  in the priory plan, while the path from node k to node q (safe area) has divided into two paths:  $a \rightarrow e \rightarrow i \rightarrow k \rightarrow l \rightarrow m \rightarrow q$  (1 car);  $a \rightarrow e \rightarrow i \rightarrow k \rightarrow l \rightarrow p \rightarrow q$  (1 car) and



$a \rightarrow e \rightarrow i \rightarrow k \rightarrow o \rightarrow p \rightarrow q$  compared to one path:  $a \rightarrow e \rightarrow i \rightarrow k \rightarrow o \rightarrow p \rightarrow q$  in the priority plan.

### 3.6 Conclusion

This initiative presents a sophisticated two-stage stochastic programming methodology designed to assess the impact of disasters, offering a flexible framework adept at managing the inherent unpredictability and dynamic characteristics associated with such occurrences. The comprehensive report centers on the development of a pragmatic model specifically tailored for analyzing the road network. This model takes into account multiple factors, including travel times and capacities, to effectively simulate the dynamic behavior of the disaster response process. These intricately crafted models serve as invaluable testbeds, facilitating a thorough evaluation of the efficacy of the minimum cost flow algorithm within the context of disaster management and response strategies.

## References

- [1] A. Shapiro, D. Dentcheva, and A. Ruszczyński, *Lectures on stochastic programming: modeling and theory*. SIAM, 2021
- [2] L. Wang, "A two-stage stochastic programming framework for evacuation planning in disaster responses", *Computers & Industrial Engineering*, vol. 145, p. 106458, 2020
- [3] Jeff Erickson. Algorithms, 1st Edition, June 2019. Section 7: Minimum Spanning Trees pp. 257-272
- [4] Moore, Edward F. (1959). The shortest path through a maze. Proc. Internat. Sympos. Switching Theory 1957, Part II. Cambridge, Massachusetts: Harvard Univ. Press. pp. 285–292. MR 0114710.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0262032937. Section 26.2: The Ford-Fulkerson method, pp. 651–664.