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

MATHEMATICAL MODELING (CO2011)

Assignment (Semester: 231)

"Stochastic Programming and Applications"

Advisor: Nguyen Tien Thinh.
Students: Le Quoc Tuan - 2153944.
Tran Nguyen Anh Khoi - 2252383.
Vo Truc Son - 2252720.
Tran Manh Tai - 2152950.
Nguyen Minh Quan - 2252683.

HO CHI MINH CITY, DECEMBER 2023

Contents

1 Member list & Workload

No.	Fullname	Student ID	Problems	Contribution
1	Le Quoc Tuan	2153944	- Problem 1	20%
2	Tran Nguyen Anh Khoi	2252383	- Problem 1	20%
3	Vo Truc Son	2252720	- Problem 2	20%
4	Tran Manh Tai	2152950	- Problem 2	20%
5	Nguyen Minh Quan	2252683	- Problem 2	20%

2 Problem 1

In this problem, we will study the two-stage stochastic model pertaining to the product manufacturing and implement this model using Python.

2.1 Theory Background

Consider a situation where a manufacturer produces n products. There are in total m different parts which have to be ordered from third-party suppliers. A unit of product i requires a_{ij}, or zero for some combination of i and j. The demand for the products is modeled as a random vector $D = (D_1,..., D_n)$. Before the demand is known, the manufacturer may preorder the parts from outside suppliers at a cost b_j per unit of part j. After the demand D is observed, the manufacturer may decide which portion of the demand is to be satisfied, so that the available numbers of parts are not exceeded. It costs additionally l_i to satisfy a unit of demand for product i, and the unit selling price of this product is q_i. The parts not used are assessed salvage values $s_j < b_j$. The unsatisfied demand is lost.

Suppose the numbers of parts ordered before production are equal to x_j, $j = 1,...,m$. After the demand D becomes known, we need to determine how much of each product to make. Denote the numbers of units produced by z_i, $i = 1,...,n$, and the numbers of parts left in inventory by y_j, $j = 1,...,m$. For an observed value (a realization) $d = (d_1,...,d_n)$ of the random demand vector D, we can find the best production plan by solving the following linear programming problem

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

subject to:

$$y_j = x_j - \sum_{i=1}^{n} a_{ij} z_i, \quad j = 1, ..., m$$

$$0 \le z_i \le d_i, \;\; i = 1, ..., n, \;\; y_j \ge 0, \;\; j = 1, ..., m.$$

Introducing the matrix A with entries a_{ij} , where $i = 1,...,n$ and $j = 1,...,m$, we can write this problem compactly as follows:

$$\min_{z,y} (l - q)^T z - s^T y$$

subject to:

$$\begin{aligned} y &= x - A^T z \\ 0 \le z &\le d, \;\; y \ge 0. \end{aligned} \tag{1}$$

Observe that the solution of this problem, that is, the vectors z and y, depend on realization d of the demand vector D as well as on x. Let $Q(x,d)$ denote the optimal value of problem (1). The quantities x_j of parts to be ordered can be determined from the following optimization problem

$$\min_{x \ge 0} b^T x + \mathbf{E}[Q(x, D)], \tag{2}$$

where the expectation is taken with respect to the probability distribution of the random demand vector D. The first part of the objective function represents the ordering cost, while the second part represents the expected cost of the optimal production plan, given ordered quantities

x. Clearly, for realistic data with $q_i > l_i$, the second part will be negative, so that some profit will be expected.

Problem (1) is called the second-stage problem, and (2) is called the first-stage problem. As the second-stage problem contains random data (random demand D), its optimal value $Q(x,D)$ is a random variable. The distribution of this random variable depends on the first-stage decisions x, and therefore the first-stage problem cannot be solved without understanding the properties of the second-stage problem. In the special case of finitely many demand scenarios $d_1,...,d_K$ occurring with positive probabilities $p_1,...,p_K$, with $\sum_{k=1}^{K} p_k = 1$, the two-stage problem (1)–(2) can be written as one large scale linear programming problem:

$$\min b^T x + \sum_{k=1}^{K} p_k[(l-q)^T z^k - s^T y^k]$$

subject to:

$$\begin{gathered} y^k = x - A^T z^k, \quad k = 1, ...K \\ 0 \le z^k \le d^k, \quad y^k \ge 0 \quad k = 1, ..., K, \\ x \ge 0, \end{gathered} \tag{3}$$

where the minimization is performed over vector variables x and z_k, y_k, $k = 1,...,K$. We have integrated the second stage problem (1) into this formulation, but we had to allow for its solution (z_k, y_k) to depend on the scenario k, because the demand realization d_k is different in each scenario. Because of that, problem (3) has the numbers of variables and constraints roughly proportional to the number of scenarios K.

There are three types of decision variables here. Namely, the numbers of ordered parts (vector x), the numbers of produced units(vector z) and the numbers of parts left in the inventory (vector y). These decision variables are naturally classified as the first and the second stage decision variables. That is, the first-stage decisions x should be made before a realization of the random data becomes available and hence should be independent of the random data, while the second-stage decision variables z and y are made after observing the random data and are functions of the data. The first-stage decision variables are often referred to as "here-and-now" decisions(solution), and second-stage decisions are referred to as "wait-and-see" decisions(solution). It can also be noticed that the second-stage problem (1) is feasible for every possible realization of the random data; for example, take $z = 0$ and $y = x$. In such a situation we say that the problem has relatively complete recourse.

2.2 Implementation

In this section, we will elucidate the implementation of the two-stage multi-product model above using Python, as well as provide some explanation regarding the requirements of the assignment.

- Data vector: According to the Assignment file, the manufacturer will produce n = 8 products satisfied **production≥demand**. The number of parts to be ordered before production is m = 5. Data vector b, l, q, s and matrix A of size 8x5 are randomly simulated. However, this program allows users to determine themselves the number of products n and the number of parts m.
- The number of scenarios: Similarly, the implemented code enables users to decide the number of scenarios of stage 2, as well as to decide whether the model is stochastic or deterministic.

- Demand distribution: We assume the random demand vector $\omega = D = (D_1,..., D_n)$ where each ω_i with density p_i follows some particular distribution functions if the model is stochastic, or determined by the user if deterministic.

Brief explanation of the source code:

The program allows the users to choose the number of parts and products instead of using the predetermined data (m = 5, n = 8) as in the assignment requirements.

The functions `get_m_and_n`, `get_b`, `get_s`, `get_l`, `get_q`, `get_a`, `get_data_a` are simply used to collect inputs from the users.

Function `get_d`, which is called to create the demand vector according to the user's choice, is comprised of 4 helping functions. `get_number_of_scenarios` and `get_is_stochastic` are used to get the number of scenarios and for the user to decide whether the model is stochastic or not. After that, either function `get_deterministic_d` or `get_stochastic_d` is called to compute the demand vector according to the user's choice.

```
def get_deterministic_d(n_scenarios):
demand = []
scenario_probability = []
for i in range(1, n_scenarios + 1):
    print("-------------------------------------")
    print(f"Scenario {i}")
    d_row = []
    for j in range (1, number_of_products + 1):
        while True:
            try:
                d_row.append(int(input(f"Demand (d) of product {j}: ")))
                break;
            except ValueError:
                print("Invalid input. Please enter an integer.\n")

    while True:
        try:
            p_tmp = float(input("Scenario probability of happening: "))
            if p_tmp < 0 or p_tmp > 1:
                raise ValueError
            scenario_probability.append(p_tmp)
            break
        except ValueError:
            print("Invalid input. Please enter valid value for probability
            ↪  (float, 0 < p < 1).\n")
    demand.append(d_row)

demand_df = pd.DataFrame(np.array(demand))
demand_df.index.name = 'Scenario'
demand_df.columns.name = 'Product'

scenario_probability_df = pd.DataFrame({'(p)': scenario_probability})
scenario_probability_df.index.name = "Scenario"
scenario_probability_df.columns.name = "Probability"
```

```
    return demand_df, scenario_probability_df
```

If the user chooses the deterministic model, the program will ask them to enter the demand vector for each scenario as well as the scenario density p within the range from 0 to 1.

```
    def get_stochastic_d(n_scenarios):
    print("Choose a probability distribution for each scenario \n - [b]
    ↪  Binomial \n - [n] Normal \n - [p] Poisson \n - [e] Exponential \n -
    ↪  [u] Uniform \n - [g] Gamma")
    demand = []
    scenario_probability = []
    for i in range(1, n_scenarios + 1):
        print("-------------------------------------")
        print(f"- Scenario {i}")
        d_row = []
        while True:
            dis_choice = input("Distribution: ")
            if dis_choice == 'b':  # Binomial
                try:
                    n_trials = int(input("Number of trials: "))
                    probability = float(input("Probability of success: "))
                    d_row = np.random.binomial(n_trials, probability,
                    ↪  size=number_of_products)
                    break
                except ValueError:
                    print("Invalid input. Please enter valid values.\n")

            elif dis_choice == 'n':  # Normal
                try:
                    mean = float(input("Mean value (float): "))
                    std_dev = float(input("Standard deviation (float): "))
                    d_row = np.random.normal(mean, std_dev,
                    ↪  size=number_of_products)
                    break
                except ValueError:
                    print("Invalid input. Please enter valid values.\n")

            elif dis_choice == 'p':  # Poisson
                try:
                    lambda_param = float(input("Lambda parameter: "))
                    d_row = np.random.poisson(lambda_param,
                    ↪  size=number_of_products)
                    break
                except ValueError:
                    print("Invalid input. Please enter a valid value.\n")

            elif dis_choice == 'u':  # Uniform
                try:
                    lower_bound = float(input("Lower bound: "))
```

```
                upper_bound = float(input("Upper bound: "))
                d_row = np.random.uniform(lower_bound, upper_bound,
                ↪  size=number_of_products)
                break
            except ValueError:
                print("Invalid input. Please enter valid values.\n")

        elif dis_choice == 'g':  # Gamma
            try:
                shape_param = float(input("Shape parameter (a or k): "))
                scale_param = float(input("Scale parameter (b or 1/): "))
                d_row = np.random.gamma(shape_param, scale_param,
                ↪  size=number_of_products)
                break
            except ValueError:
                print("Invalid input. Please enter valid values.\n")

        else:
            print("Invalid distribution choice, please choose again.\n")

    # Get the probability of happening of each scenario
    ↪  -------------------------
    while True:
        try:
            p_tmp = float(input("Scenario probability of happening: "))
            if p_tmp < 0 or p_tmp > 1:
                raise ValueError
            scenario_probability.append(p_tmp)
            break
        except ValueError:
            print("Invalid input. Please enter valid value for probability
            ↪  (float, 0 < p < 1).\n")

    demand.append(d_row)

demand_df = pd.DataFrame(np.array(demand))
demand_df.index.name = 'Scenario'
demand_df.columns.name = 'Product'

scenario_probability_df = pd.DataFrame({'(p)': scenario_probability})
scenario_probability_df.index.name = "Scenario"
scenario_probability_df.columns.name = "Probability"
return demand_df, scenario_probability_df
```

Should the user want their model to be stochastic, the program will demand their choice of distribution function(binomial, normal, Poisson, exponential, uniform and Gamma distribution) for each scenario along with the density. For each type of distribution functions, the user will be asked to enter some data for calculation of the demand vector.

Function `get_user_input` includes all the above helping functions to create simulated data

for solving the model. `data_double_check` enables the users to recheck and edit some data vectors if they want.

```
    def solve_model(unit_preorder_cost, unit_prod_cost, unit_price,
    ↪  salvage_cost, a_matrix, demand, scenario_probability):
    from gamspy import Container, Set, Parameter, Variable, Equation, Model,
    ↪  Sum, Sense
    number_of_products = unit_prod_cost.shape[0]
    number_of_parts = unit_preorder_cost.shape[0]
    number_of_scenarios = scenario_probability.shape[0]
    products = []
    for i in range(1, number_of_products + 1):
        products.append(f'product {i}')
    parts = []
    for i in range(1, number_of_parts + 1):
        parts.append(f'parts {i}')
    scenarios = []
    for i in range(1, number_of_scenarios + 1):
        scenarios.append(f'scenario {i}')
    m = Container()
    i = Set(
        container=m,
        name="products",
        records=np.array(products),
        description="Products domain",
    )
    j = Set(
        container=m,
        name="parts",
        records=np.array(parts),
        description="Parts domain",
    )
    k = Set(
        container=m,
        name="scenarios",
        records=np.array(scenarios),
        description="scenarios domain"
    )
    b = Parameter(
        container=m,
        name="b",
        domain=j,
        records=np.array(unit_preorder_cost),
    )
    l = Parameter(
        container=m,
        name="l",
        domain=i,
        records=np.array(unit_prod_cost),
```

```
)
q = Parameter(
    container=m,
    name="q",
    domain=i,
    records=np.array(unit_price),
)
s = Parameter(
    container=m,
    name="s",
    domain=j,
    records=np.array(salvage_cost),
)
A = Parameter(
    container=m,
    name="A",
    domain=[i,j],
    records=np.array(a_matrix),
)
d = Parameter(
    container=m,
    name="d",
    domain=[k, i],
    records=np.array(demand)
)
p = Parameter(
    container=m,
    name="p",
    domain=[k],
    records=np.array(scenario_probability)
)
x = Variable(
    container=m,
    name="x",
    type="integer",
    domain=j,
)
y = Variable(
    container=m,
    name="y",
    type="integer",
    domain=[k, j]
)
z = Variable(
    container=m,
    name="z",
    type="integer",
    domain=[k, i]
)
```

```
    obj = Sum(j, b[j]*x[j]) + Sum(k, p[k]*(Sum(i, (l[i] - q[i])*z[k, i]) -
    ↪  Sum(j, s[j]*y[k, j])))
    demand_constraint = Equation(
        container=m,
        name="demand_constraint",
        domain=[k, i]
    )
    relationship = Equation(
        container=m,
        name="relationship",
        domain=[k, j]
    )
    demand_constraint[k, i] = 0 <= z[k, i] <= d[k, i]
    relationship[k, j] = y[k, j] == x[j] - Sum(i, A[i, j]*z[k, i])
    problem_1 = Model(
        container=m,
        name="problem_1",
        equations=m.getEquations(),
        problem="MIP",
        sense=Sense.MIN,
        objective=obj,
    )
    viewSolverconsole = input("View solver's console [y/n]? ")
    clearTerminal()
    if viewSolverconsole == 'y':
        import sys
        problem_1.solve(output=sys.stdout)
        input("Press enter to show result summary")
        print("------------------------------------------------------------")
    else:
        problem_1.solve()
    return problem_1, x, y, z
```

The function `solve_model` will set up the collected data vector, as well as the objective function and the constraints of the two-stage stochastic programming problem and then use gamspy module to find the optimal solution to the problem.

2.3 Testing

In this section, we will run a simulated implementation of the codes based on the data requirements in the Assignment file, with n = 8 and m = 5, the number of scenarios S = 2 with density $p_s = 0.5$. The demand vector $\omega = D = (D_1,..., D_n)$ where each ω_i with density p_i follows the binomial distribution Bin(10, 1/2). Vectors b, l, q, s and matrix A will be randomly chosen.

```
Number of parts (m): 5
Number of products (n): 8

------------- PRE-ODER COST (b) -------------
Preorder cost (b) of part 1: 8
Preorder cost (b) of part 2: 7
Preorder cost (b) of part 3: 9
Preorder cost (b) of part 4: 5
Preorder cost (b) of part 5: 6

------------- SALVAGE COST (s) --------------
Salvage cost (s) of part 1: 4
Salvage cost (s) of part 2: 3
Salvage cost (s) of part 3: 2
Salvage cost (s) of part 4: 1
Salvage cost (s) of part 5: 3

---------- UNIT PRODUCTION COST (l) ---------
Unit production (l) cost of product 1: 2000
Unit production (l) cost of product 2: 1200
Unit production (l) cost of product 3: 1500
Unit production (l) cost of product 4: 700
Unit production (l) cost of product 5: 1400
Unit production (l) cost of product 6: 2600
Unit production (l) cost of product 7: 1600
Unit production (l) cost of product 8: 2000

-------------- UNIT PRICE (q) ---------------
Unit price (a) of product 1: 2000
Unit price (a) of product 2: 1200
Unit price (a) of product 3: 1500
Unit price (a) of product 4: 700
Unit price (a) of product 5: 1400
Unit price (a) of product 6: 2600
Unit price (a) of product 7: 1600
Unit price (a) of product 8: 2000
```

```
---------- NUMBER OF PARTS NEEDED FOR PRODUCTION (a) ----------
Please enter matrix a in the pop-up window
Part      0   1   2   3   4
Product
0         2   4   5  10  16
1         4   6  10  12  20
2         8  12  22   2   6
3        25   7  12  30  29
4        12  13  23   4   9
5        32  15   7   9  10
6        12  35  36  14  20
7        24  12  16   8  10
```

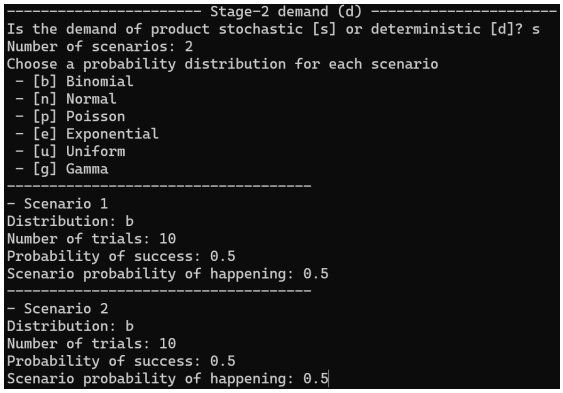


Figure 1: Input data

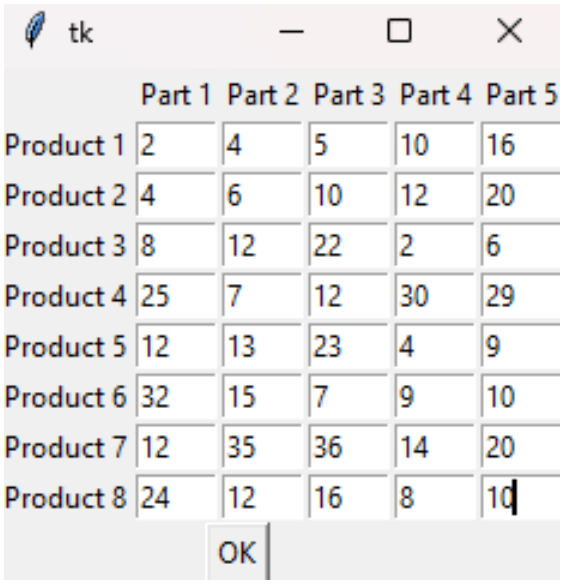


Figure 2: Matrix A input using tkinter module

The images above represent the inputs of the model determined by the user.

```
PLEASE DOUBLE CHECK YOUR DATA:
--------------------------------------------------
Unit preorder cost (b):
     b
0  8.0
1  7.0
2  9.0
3  5.0
4  6.0
Enter [b] to edit
--------------------------------------------------
Unit production cost (l):
      l
0  19.0
1  11.0
2  14.0
3   6.0
4  13.0
5  25.0
6  15.0
7  19.0
Enter [l] to edit
--------------------------------------------------
Unit price (q):
      q
0  2000
1  1200
2  1500
3   700
4  1400
5  2600
6  1600
7  2000
Enter [q] to edit
```

```
--------------------------------------------------
Salvalge cost (s):
     s
0  4.0
1  3.0
2  2.0
3  1.0
4  3.0
Enter [s] to edit
--------------------------------------------------
Number of parts needed for each product (A):
Part      0   1   2   3   4
Product
0         2   4   5  10  16
1         4   6  10  12  20
2         8  12  22   2   6
3        25   7  12  30  29
4        12  13  23   4   9
5        32  15   7   9  10
6        12  35  36  14  20
7        24  12  16   8  10
Enter [a] to edit
--------------------------------------------------
Stage-2 demand (d):
Product   0  1  2  3  4  5  6  7
Scenario
0         7  6  6  6  3  8  3  8
1         3  3  7  6  4  4  3  4

Scenario probabilities (p):
Probability  (p)
Scenario
0            0.5
1            0.5

Enter [d] to edit
--------------------------------------------------
Or press [ok] to solve:
```

Figure 3: Double check data

It can be seen that besides binomial distribution, the users can also choose other distribution functions for each scenario, including normal, Poisson, exponential, uniform and Gamma

distribution. The users can also make changes to the data if they want.

```
Objective value =  -45639.0
---------------------------------------------------------------------------
The numbers of parts to be ordered before production (x) - here-and-now
     parts  level  marginal  lower  upper  scale
0  parts 1  606.0       8.0    0.0    inf    1.0
1  parts 2  496.0       7.0    0.0    inf    1.0
2  parts 3  588.0       9.0    0.0    inf    1.0
3  parts 4  386.0       5.0    0.0    inf    1.0
4  parts 5  515.0       6.0    0.0    inf    1.0
---------------------------------------------------------------------------
The numbers of parts left in inventory (y) - wait-and-see
    scenarios    parts  level  marginal  lower  upper  scale
0  scenario 1  parts 1    0.0      -2.0    0.0    inf    1.0
1  scenario 1  parts 2    0.0      -1.5    0.0    inf    1.0
2  scenario 1  parts 3    0.0      -1.0    0.0    inf    1.0
3  scenario 1  parts 4   42.0      -0.5    0.0    inf    1.0
4  scenario 1  parts 5    0.0      -1.5    0.0    inf    1.0
5  scenario 2  parts 1   74.0      -2.0    0.0    inf    1.0
6  scenario 2  parts 2   75.0      -1.5    0.0    inf    1.0
7  scenario 2  parts 3   25.0      -1.0    0.0    inf    1.0
8  scenario 2  parts 4    0.0      -0.5    0.0    inf    1.0
9  scenario 2  parts 5   15.0      -1.5    0.0    inf    1.0
---------------------------------------------------------------------------
The number of units to be produced (z) - wait-and-see
     scenarios   products  level  marginal  lower  upper  scale
0   scenario 1  product 1    7.0    -990.5    0.0    inf    1.0
1   scenario 1  product 2    6.0    -594.5    0.0    inf    1.0
2   scenario 1  product 3    6.0    -743.0    0.0    inf    1.0
3   scenario 1  product 4    0.0    -347.0    0.0    inf    1.0
4   scenario 1  product 5    3.0    -693.5    0.0    inf    1.0
5   scenario 1  product 6    8.0   -1287.5    0.0    inf    1.0
6   scenario 1  product 7    3.0    -792.5    0.0    inf    1.0
7   scenario 1  product 8    8.0    -990.5    0.0    inf    1.0
8   scenario 2  product 1    3.0    -990.5    0.0    inf    1.0
9   scenario 2  product 2    3.0    -594.5    0.0    inf    1.0
10  scenario 2  product 3    7.0    -743.0    0.0    inf    1.0
11  scenario 2  product 4    6.0    -347.0    0.0    inf    1.0
12  scenario 2  product 5    4.0    -693.5    0.0    inf    1.0
13  scenario 2  product 6    4.0   -1287.5    0.0    inf    1.0
14  scenario 2  product 7    3.0    -792.5    0.0    inf    1.0
15  scenario 2  product 8    4.0    -990.5    0.0    inf    1.0
```

Figure 4: Result

This is the optimal solution of the problem and the values of 3 decision variables x, y, z. Here, the negative solution means that we can expect to gain some profit.

2.4 Conclusion

The two-stage stochastic programming model for manufacturing proves to be a very realistic and effective way of predicting the unknown demand as well as minimizing the production cost with the aim of gaining more profit. However, in real-life situations, in order to achieve the optimal values, the manufacturers need to take into account as many scenarios as possible, together with other external factors affecting the demand of the products, which can lead to a large number of variables for calculation. Therefore, the model can sometimes fail to include some random cases.

3 Problem 2

In problem 2, there are a few approaches to tackle this. Briefly, the objective is to make the optimal evacuation planning in the 1^{st} stage under uncertainty to be faced in the 2^{nd} stage. However, in this specific assignment, we will focus on the Algorithm 1 based on the min-cost flow problem in the **Reference [1]**.

3.1 Min-cost flow problem:

To begin with this problem, we need to study max flow problem and its special case which is min-cost flow problem.

3.1.1 Max flow:

The max flow problem is a classic optimization problem in graph theory that involves finding the maximum amount of flow that can be sent through a network of pipes, channels, or other pathways, subject to capacity constraints. The problem can be used to model a wide variety of real-world situations, such as transportation systems, communication networks, and resource allocation.

In the max flow problem, we have a directed graph with a source node s and a sink node t, and each edge has a capacity that represents the maximum amount of flow that can be sent through it. The goal is to find the maximum amount of flow that can be sent from s to t, while respecting the capacity constraints on the edges.

***Definition*:**
Max flow problem is defined as the maximum amount of flow that the network would allow to flow from source to sink. Multiple algorithms exist to solve this problem.

Solving max flow problem:
The goal of the maximum flow problem is to determine the maximum amount of flow that can be sent from the source to the sink without violating capacity constraints along the edges. One common approach to solving the max flow problem is *the Ford-Fulkerson algorithm*, which is based on the idea of augmenting paths. The algorithm starts with an initial flow of zero, and iteratively finds a path from s to t that has available capacity, and then increases the flow along that path by the maximum amount possible. This process continues until no more augmenting paths can be found.

- Advantages:

  1. The max flow problem is a flexible and powerful modeling tool that can be used to represent a wide variety of real-world situations.

  2. The Ford-Fulkerson and Edmonds-Karp algorithms are both guaranteed to find the maximum flow in a graph, and can be implemented efficiently for most practical cases.

  3. The max flow problem has many interesting theoretical properties and connections to other areas of mathematics, such as linear programming and combinatorial optimization.

- Disadvantages:

  1. In some cases, the max flow problem can be difficult to solve efficiently, especially if the graph is very large or has complex capacity constraints.

  2. The max flow problem may not always provide a unique or globally optimal solution, depending on the specific problem instance and algorithm used.

Ford-Fulkerson Algorithm:

The Ford-Fulkerson algorithm is an algorithm that tackles the max-flow min-cut problem. It was discovered in 1956 by Ford and Fulkerson. This algorithm is sometimes referred to as a method because parts of its protocol are not fully specified and can vary from implementation to implementation. An algorithm typically refers to a specific protocol for solving a problem, whereas a method is a more general approach to a problem.

Ford-Fulkerson has a complexity of O(|E|.f*). where f* is the maximum flow of the network.

Below are two versions of pseudocode for this algorithm (graph G, source s, sink t):

Algorithm 1 Simplified version of Ford-Fulkerson algorithm

```
initialize flow to 0
find augmenting path from s to t
while path exists:
    augment flow along path
    create residual graph G_f
    find augmenting path from s to t in G_f
return flow
```

Algorithm 2 In-depth version of Ford-Fulkerson algorithm

```
initialize flow to 0
for each edge (u, v) in G:
    flow(u, v) = 0
while there is a path, p, from s -> t in residual network G_f:
    residualCapacity(p) = min(residualCapacity(u, v) : for (u, v) in p)
    flow = flow + residualCapacity(p)
    for each edge (u, v) in p:
        if (u, v) is a forward edge:
            flow(u, v) = flow(u, v) + residualCapacity(p)
        else:
            flow(u, v) = flow(u, v) − residualCapacity(p)
return flow
```

Residual Graph:

In the Ford-Fulkerson algorithm, a residual graph is used to represent remaining capacity along edges after some flow has been sent. This graph helps in finding augmenting paths efficiently.

Residual graphs are an important middle step in calculating the maximum flow. As noted in the pseudo-code, they are calculated at every step so that augmenting paths can be found from the source to the sink. To understand how these are created and how they are used, we can use the graph from the intuition section.

However, one important attribute is important to understand before looking at residual graphs. Residual capacity is a term used in the above pseudo-code, and it plays an important role in residual graph creation. Residual capacity is defined as the new capacity after a given flow has been taken away. In other words, for a given edge (u,v), the residual capacity, cf is defined as

$$c_f(u, v) = c(u, v) - f(u, v)$$

About intuition:

The intuition behind the algorithm is quite simple (even though the implementation details can obscure this). Imagine a flow network that is just a traffic network of cars. Each road can hold a certain number of cars. This can be illustrated in **Figure 5**.

It was shown that 2 units of flow can be pushed along the top-most path initially. When this happens, only three edges are affected: (S, A), (A, B), and (B, T). (S, A) and (A, B) are affected in the same way because they have the same capacity. Two things happen:

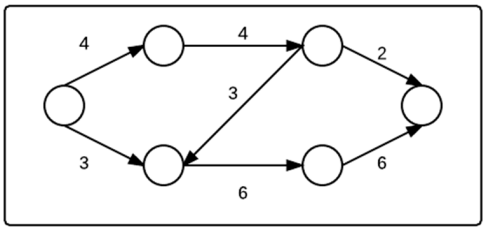


Figure 5: Flow network

1. **In the forward direction**, the edges now have a residual capacity equal to c_f(u,v) = c(u,v) − f(u,v). The flow is equal to 2, so the residual capacity of (S, A) and (A, B) is reduced to 2, while the edge (B, T) has a residual capacity of 0.
2. **In the backward direction**, the edges now have a residual capacity equal to c_f(u,v) = c(u,v) − f(u,v). Because of flow preservation, this can be written as c_f(u,v) = c(u,v) + f(u,v), and since the capacity of those backward edges was initially 0, all of the backward edges (T, B), (B, A), and (A, S) now have a residual capacity of 2.

The intuition goes like this: as long as there is a path from the source to the sink that can take some flow the entire way, we send it. This path is called an augmenting path. We keep doing this until there are no more augmenting paths.

In **Figure 6**, we could start by sending 2 cars along the topmost path (because only 2 cars can get through the last portion). Then we might send 3 cars along the bottom path for a total of 5 cars. Finally, we can send 2 more cars along the top path for two edges, send them down to bottom path and through to the sink. The total number of cars sent is now 7, and it is the maximum flow.

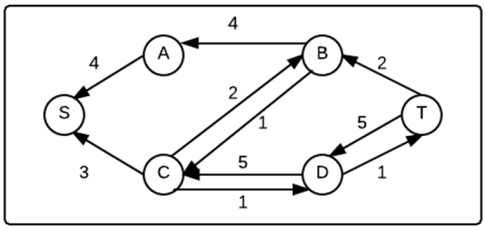


Figure 6: Maximum flow network

3.1.2 Min-cost flow:

Min-cost flow is a special case of max flow, but instead of maximizing the amount of flow that can be send from a source node, we try to minimize the cost of sending goods from source node to sink while respecting the flow constraints.

Therefore, the min-cost flow not only looks for ways to move the required amount of goods but also optimizes the cost associated with that movement.

Algorithm:

It's similar with Edmonds-Karp algorithm for computing max flow.

For the simplest case, the graph is oriented and there is at most one edge between any pairs of vertices.

Let U_{ij} be the the capacity of an edge (i, j), C_{ij} is the cost per unit of flow along this edge and F_{ij} is the flow along the edge. The initial stage is zero.

Modifying the network, add reverse edge (j, i) for each edge (i, j) to the network with capacity $U_{ji} = 0$ and the cost $C_{ji} = -C_{ij}$. As the restrictions before, the edge (j, i) was not existed in the network, we still have a network that is not multigraph. Therefore, we can keep the condition $F_{ji} = -F_{ij}$ true during the test.

Defining the residual network for some fixed flow F as follow, the residual network contains only unsaturated edges ($F_{ij} < U_{ij}$), and the residual capacity of each edge is:

$$R_{ij} = U_{ij} - F_{ij}$$

Now, at each iteration of the algorithm, we find the shortest path in the residual graph from s to t. The difference with max flow algorithm is we find the shortest path based on the cost of the path instead of the number of edges. It will terminate if it does not exist paths anymore and the stream F is the desired one. If the path was found, increasing the flow along it as much as possible (in detail, we find the minimal residual capacity R and increase the flow by it, and reduce the back edge by the same amount). If at some point the flow reach the value K, then we stop (in the last iteration, it is necessary to increase the flow by only such an amount so that the final flow value does not surpass K).

If we set K to infinite, the algorithm will find minimum-cost maximum flow.

Undirected graphs/ multigraphs:

The case of an undirected graph or a multigraph doesn't differ conceptually from the algorithm above. The algorithm will also work on these graphs. However, it becomes a little more difficult to implement it.

An undirected edge (i, j) is actually the same as two oriented edges (i, j) and (j, i) with the same capacity and values. Since the above-described minimum-cost flow algorithm generates a back edge for each directed edge, so it splits the undirected edge into 4 directed edges, and we actually get a multigraph.

Some notations when dealings with multiple edges:

1. The flow for each of the multiple edges must be kept separately.

2. When searching for the shortest path, it is necessary to take into account that it is important which of the multiple edges is used in the path. Instead of the usual ancestor array we additionally must store the edge number from which we came from along with the ancestor.

3. As the flow increases along a certain edge, it is necessary to reduce the flow along the back edge.

Since we have multiple edges, we have to store the edge number for the reversed edge for each edge. Then, there are no other obstructions with undirected graphs or multigraphs.

Time complexity:

In the worst case, it may push only as much at 1 unit of flow on each iteration taking O(F) iterations to find a minimum-cost flow of size F, making a total run-time is O(F*T).

Implementation:

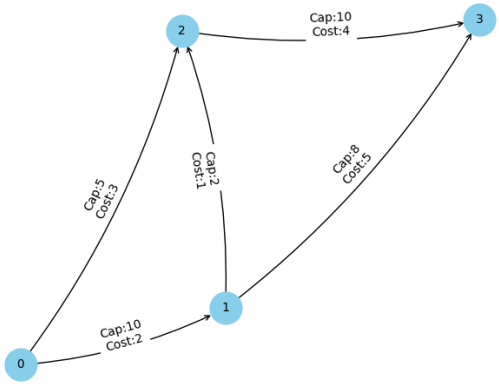
```python
from collections import defaultdict
import heapq
import networkx as nx
import matplotlib.pyplot as plt

class MinCostFlowSolver:
    def __init__(self, num_nodes):
        self.num_nodes = num_nodes
        self.graph = defaultdict(list)

    def add_edge(self, u, v, capacity, cost):
        self.graph[u].append((v, capacity, cost))
        self.graph[v].append((u, 0, -cost))  # Backward edge

    def dijkstra(self, source, target):
        dist = [float('inf')] * self.num_nodes
        parent = [-1] * self.num_nodes
        dist[source] = 0

        priority_queue = [(0, source)]

        while priority_queue:
            current_dist, u = heapq.heappop(priority_queue)

            if current_dist > dist[u]:
                continue

            for v, capacity, cost in self.graph[u]:
                if capacity > 0 and dist[u] + cost < dist[v]:
                    dist[v] = dist[u] + cost
                    parent[v] = u
                    heapq.heappush(priority_queue, (dist[v], v))

        return dist, parent

    def min_cost_flow(self, source, target, flow):
        total_cost = 0
        total_flow = 0

        while True:
            dist, parent = self.dijkstra(source, target)

            if dist[target] == float('inf'):
                break

            augmenting_flow = float('inf')
            node = target
```

```
            while node != source:
                parent_node = parent[node]
                for edge in self.graph[parent_node]:
                    if edge[0] == node:
                        augmenting_flow = min(augmenting_flow, edge[1])
                node = parent_node

            total_flow += augmenting_flow
            total_cost += dist[target] * augmenting_flow

            node = target
            while node != source:
                parent_node = parent[node]
                for i, edge in enumerate(self.graph[parent_node]):
                    if edge[0] == node:
                        self.graph[parent_node][i] = (edge[0], edge[1] -
                        ↪  augmenting_flow, edge[2])
                        break
                found = False
                for i, edge in enumerate(self.graph[node]):
                    if edge[0] == parent_node:
                        self.graph[node][i] = (edge[0], edge[1] +
                        ↪  augmenting_flow, edge[2])
                        found = True
                        break
                if not found:
                    self.graph[node].append((parent_node, augmenting_flow,
                    ↪  -self.graph[parent_node][-1][2]))
                node = parent_node

        return total_flow, total_cost
    def plot_graph(self):
        G = nx.DiGraph()

        for u, edges in self.graph.items():
            for v, capacity, cost in edges:
                if capacity > 0:  # Only include forward edges
                    G.add_edge(u, v, capacity=capacity, cost=cost)

        pos = nx.spring_layout(G)
        edge_labels = {(u, v):
        ↪  f"Cap:{G[u][v]['capacity']}\nCost:{G[u][v]['cost']}" for u, v in
        ↪  G.edges()}

        nx.draw(G, pos, with_labels=True, node_size=700, node_color="skyblue",
        ↪  font_size=10, arrows=True, connectionstyle='arc3,rad=0.1',
        ↪  arrowstyle='->')
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
```

```
        plt.show()



# Sample input
solver = MinCostFlowSolver(4)
solver.add_edge(0, 1, 10, 2)
solver.add_edge(0, 2, 5, 3)
solver.add_edge(1, 2, 2, 1)
solver.add_edge(1, 3, 8, 5)
solver.add_edge(2, 3, 10, 4)

solver.plot_graph()

source_node = 0
sink_node = 3

total_flow, total_cost = solver.min_cost_flow(source_node, sink_node,
↪  float('inf'))
print("Minimum Cost Flow:")
print("Total Flow:", total_flow)
print("Total Cost:", total_cost)
```

Testing:

The original graph after being initialized and added all the edges with theirs capacity and cost call by *solver.plot_graph()*:



The output of the program will print the total flow and total cost from source node to sink node which is from 1 to 3:

```
Minimum Cost Flow:
Total Flow: 15
Total Cost: 105
```

3.2 Two-stage stochastic programming framework for evacuation planning in disaster responses:

3.2.1 Overview:

When a disaster strikes, the greatest concern is minimizing the loss of life. This subject models the precise movement of people from risky to safe places as a minimum-cost flow issue with random link travel times and limited flow capacities.

The model, however, differs from the usual min-cost flow problem. The two-stage stochastic programming model with recourse, in particular, illustrates a situation in which the first and second stages occur at different times in the same evacuation network. It should be noted that the first-stage evacuation plan may be impractical for the given realities, which is resolved by permitting shorter evacuation time frames in the second stage. In this case, the objective function will consist of the first-stage punish costs and the expected value of the second stage recourse costs.

3.2.2 Notations and decision variables:

Before diving into the details, some essential notations and decision variables are introduced in order to solve the problem in Mathematical formulation (*Table 1 and Table 2*)

Symbol	Definition
V	the set of nodes
A	the set of links
i, j	the index of nodes with i, j∈V
(i, j)	the index of directed links with (i, j)∈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

Table 1: *Subscripts and parameters 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

Table 2: *Decision variables used in Mathematical Formulation*

3.2.3 Stage one (A Priori):

Stage one of a two-stage stochastic programming framework for evacuation planning in disaster responses involves the initial planning before the occurrence of the disaster. This phase focuses on

making decisions based on anticipated scenarios and uncertainties. Decision-makers collect and analyze data, formulate a mathematical optimization model, and make key decisions regarding resource allocation, evacuation routes, and shelter locations. The objective is to develop a robust initial evacuation plan that considers various potential disaster scenarios. This plan serves as a foundation for the second stage, where real-time adjustments can be made in response to actual events during the disaster.

Particularly, the two decision variables will be used to formulate an objective function for this stage. Other than that, the function also strictly follows some system constraints.

***Constraints*:**

- The *flow balance constraint* ensures flow balance for each node, considering the supply value for the source and demand value for the target.:

$$\sum_{i,j \in A} x_{ij} - \sum_{j,i \in A} x_{ji} = d_i \tag{1}$$

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

- The *capacity constraint* on each link to limit the flow to its capacity in the network:
$0 \leqslant \text{x}_{ij} \leqslant \text{u}_{ij}, \forall(\text{i, j}) \in \text{A}$ (2)

***Objective function*:** In the first stage, the objective function is the penalty function whose purpose is to eliminate the potential loops being created while traversing the generated path in the network. So as to achieve this task, a new variable which is the link penalty p_{ij} is introduced. Therefore, the objective function can now be defined as

$$f(\mathbf{X}) = \sum_{(i,j) \in A} p_{ij} x_{ij} \tag{3}$$

where $\mathbf{X} \coloneqq \{\text{x}_{ij}\}_{(i,j) \in A}$: the vector of flow

3.2.4 Stage two (Adaptive):

Stage two of a two-stage stochastic programming framework for evacuation planning in disaster responses occurs in real-time during the unfolding disaster. This phase involves adapting the initial evacuation plan based on updated information and changing conditions. Decision-makers make real-time adjustments to evacuation routes, resource allocations, and shelter locations to minimize the evacuation time as well as the cost of evacuation to the evolving situation. This adaptability enhances the overall effectiveness of the evacuation response, ensuring a timely and well-coordinated effort to protect and evacuate the affected population.

Similar to stage one, this post-disaster stage also has some constraints to satisfy so that the objective function can be formulated.

***Constraints*:**

- The coupling constraint before the time threshold, which represents the relationship between the physical link and the space-time arc:

$$y_{ij}^s(t) = x_{ij}, \forall t \in \widetilde{T}, (i,j) \in A, s = 1,2,3,...,S \tag{4}$$

  In other words, in any scenarios, in advance of the time threshold, the evacuation plan in each scenario of the second stage is the same as the priori evacuation plan.

Objective function is to minimize the overall evacuation time considering the travel time and flow on links in scenario s:

$$Q(Y,s) = min \sum_{(i,j)\in A_s} c_{ij}^s(t).y_{ij}^s(t) \tag{5}$$

subject to:

$$\sum_{(i_t,j_{t'})\in A_s} y_{ij}^s(t) - \sum_{(j_{t'},i_t)\in A_s} y_{ij}^s(t'), \forall i \in V, t \in 0,1,...,T; s = 1,2,...,S \tag{6}$$

$$0 \leq y_{ij}^s(t) \leq u_{ij}^s(t), \forall (i,j) \in A, t \in 0,1,...,T, s = 1,2,...,S \tag{7}$$

$$\sum_{t\leq\widetilde{T}} y_{ij}^s(t) = x_{ij}, (i,j) \in A, s = 1,2,...,S \tag{8}$$

where (5) is the overall evacuation time of all traffic volumes in the minimization scenario s. Constraints (6) and (7) are flow balance constraint and traffic capacity constraint respectively.The constraint (8) is a coupling constraint to ensure that the evacuation scheme in each scenario in the second stage before the time threshold T is the same as the a priori plan in the first stage.

3.2.5 Two-stage stochastic evacuation planning model in a time-dependent network:

In this assignment, we need make a priori evacuation plan in the first stage which is able to adapt to all the possible scenarios in the second stage. Therefore, a priori plan need to be made in an estimated period of time before and after the occurrence of the disaster. Plus, there are a probability for each scenario that can happen in the disastrous event.
In order to minimize the penalty for the prior evacuation plan and the expected overall evacuation time of each scenario's adaptive evacuation plan, a two-stage evacuation planning model in time-dependent and random environment is formulated as:

$$min \sum_{(i,j)\in A} p_{ij}x_{ij} + \sum_{s=1}^{S} \mu_s.Q(Y,s)$$

subject to:

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

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

where,

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

subject to:

$$\sum_{(i_t,j_{t'}) \in A_s} y_{ij}^s(t) - \sum_{(j_{t'},i_t) \in A_s} y_{ij}^s(t') = d_t^s, \forall i \in V, t \in 0,1,...,T, s = 1,2,...,S$$

$$0 \leq y_{ij}^s(t) \leq u_{ij}^s(t), \forall (i,j) \in A, t \in 0,1,...,T, s = 1,2,...,S$$

$$\sum_{t \leq \tilde{T}} y_{ij}^s(t) = x_{ij} \in A, s = 1,2,...,S \tag{9}$$

Since the second stage of the model has a limited number of scenarios, the above time-dependent and stochastic two stage evacuation planning model is equivalent to the following single stage model which combines the penalty for the prior evacuation plan and the expected overall evacuation time for all scenarios:

$$min \sum_{(i,j) \in A} p_{ij} x_{ij} + \sum_{s=1}^{S} (\mu . \sum_{(i,j) \in A_s} c_{ij}^s(t).y_{ij}^s(t))$$

subject to:

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

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

$$\sum_{(i_t,j_{t'}) \in A_s} y_{ij}^s(t) - \sum_{(j_{t'},i_t) \in A_s} y_{ij}^s(t') = d_i^s(t), \forall i \in V, t \in 0,1,...,T, s = 1,2,...,S$$

$$0 \leq y_{ij}^s(t) \leq u_{ij}^s(t), \forall (i,j) \in A, t \in 0,1,...,T, s = 1,2,...,S$$

$$\sum_{t \leq \tilde{T}} y_{ij}^s(t) = x_{ij} \in A, s = 1,2,...,S \tag{10}$$

In summary, the formulated mathematical model provides a comprehensive framework for evacuation planning, considering both a priori planning and real-time adaptability to uncertain disaster scenarios. The inclusion of probabilities and the two-stage structure enhance the model's ability to address the dynamic nature of disaster response. However,

3.3 Algorithmic solution to the problem:

As model (10) is essentially an integer programming model where the coupling constraint is a complex constraint, which leads to the model cannot be solved in polynomial time. Therefore, we need to decompose the model into two sub-model by introducing Lagrangian relaxation approach so that it will be easier to carry out. Specifically, the original model is divided into a min-cost flow problem which is mentioned in **3.1** and a time-dependent problem. In this assignment, we will elaborate the first model by applying the successive shortest path algorithm.

The successive shortest path algorithm is a classic method for solving the minimum cost flow problem, a fundamental optimization problem in network flow theory. This algorithm is based on the idea of iteratively finding and augmenting shortest paths in the residual network. The paths are shortest in terms of the reduced cost, which is the cost relative to the current flow. This ensures that the algorithm always makes progress towards the optimal solution. The algorithm terminates when the desired flow value is reached or when no more flow can be sent, i.e., when there is no path from the source to the sink in the residual network. At this point, the current flow is a minimum cost flow.

The algorithm is stated in **Reference [1]** as below:

Algorithm 3 Successive shortest path algorithm for min-cost flow problem.

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

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

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

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

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

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

Let's elaborate on the key steps of the algorithm and provide additional insights:

1. Initialization:

The algorithm starts by taking a variable x as a feasible flow between any origin-destination (OD) pair. This flow x has the minimum delivery cost among all feasible flows with the same flow value.

2. Termination check:

The algorithm will terminate if:

- the flow value of x reaches v (the desired flow value)
- there is no minimum cost path in the residual network (V, A(x), C(x), U(x), D).

If the algorithm does not terminate, it calculates the shortest path with the maximum flow using a label-correcting algorithm, and then proceeds to Step 3.

The functions A(x), C(x), U(x) in the residual network are defined as follows:

- A(x): This represents the set of links in the residual network. It includes a link (i,j) if and only if (i,j) is a link in the original network A and the flow x_{ij} on (i,j) is less than its upper bound u_{ij}. It also includes a reverse link (j,i) if and only if (j,i) is a link in A and the flow x_{ij} is greater than zero.
- C(x): This represents the cost function in the residual network. If (i,j) is a link in A and the flow x_{ij} is less than its upper bound u_{ij}, then the cost of link (i,j) in the residual network is the same as its cost c_{ij} in the original network. If (j,i) is a link in A and the flow x_{ji} is greater than zero, then the cost of the reverse link (j,i) in the residual network is the negative of the cost c_{ji} of link (j,i) in the original network.
- U(x): This represents the capacity function in the residual network. If (i,j) is a link in A and the flow x_{ij} is less than its upper bound u_{ij}, then the capacity $U_{ij}(x)$ of link (i,j) in the residual network is the same as its upper bound u_{ij} in the original network. If (j,i) is a link in A and the flow x_{ji} is greater than zero, then the capacity $U_{ji}(x)$ of the reverse link (j,i) in the residual network is the same as the flow x_{ji} on link (j,i) in the original network.

3. Flow augmentation:

The algorithm increases the flow along the minimum cost path. If the increased flow value does not exceed v, the algorithm goes back to Step 2.

Time Complexity:

The time complexity of the algorithm is $O(n^3 * m)$ for a network with n nodes and m edges. This makes it efficient for small to medium-sized problems, but other algorithms may be more efficient for large-scale problems. The algorithm is also flexible and can be easily adapted to handle additional constraints or variations of the minimum cost flow problem. For example, it can be modified to handle the case where the nodes, rather than the arcs, have costs and capacities, or the case where the goal is to maximize profit rather than minimize cost. The algorithm is also robust and can handle networks with complex topologies and a wide range of cost and capacity values.

Assumptions and Limitations:

However, like all network flow algorithms, it assumes that the network is connected, i.e., there is a path from the source to the sink, and that the capacities and costs are non-negative. If these assumptions are not met, the algorithm may not work correctly or may produce incorrect results. Therefore, it's important to check these assumptions before applying the algorithm.

Comparison with Network Simplex Algorithm:

The *Network Simplex Algorithm* is a network flow algorithm that works directly with the network simplex data structure. It maintains optimality by performing pivot operations on the basis of the simplex algorithm.

Comparison:

- Efficiency: Network simplex algorithm is often more efficient in practice, especially for large networks, due to its more sophisticated data structures and pivot operations.
- Sparse Graphs: successive shortest path algorithm may perform well on sparse graphs where finding augmenting paths is relatively fast, but it can become inefficient on dense graphs.

- Implementation: successive shortest path algorithm is conceptually simpler and easier to implement, making it a good choice for educational purposes or simpler applications.
- Worst-Case: The worst-case complexity of successive shortest path algorithm can be higher than Network simplex algorithm, but in practice, the performance may depend on the specific characteristics of the problem instance.

To sum up, the choice depends on the specific characteristics of the problem instance, implementation considerations, and the desired trade-off between simplicity and efficiency. Network simplex algorithm is often preferred for large and dense networks, while successive shortest path algorithm might be more suitable for simpler applications or educational purposes.

Conclusion:

Despite these limitations, the successive shortest path algorithm is a powerful tool for solving the minimum cost flow problem and has many applications in areas such as transportation, logistics, supply chain management, and telecommunication. It is also a fundamental algorithm in network optimization and is a key component of more complex algorithms and models in this field.

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] HackerEarth, "Maximum flow Tutorials & Notes", Algorithms, 2016. https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/tutorial/

[3] GeeksforGeeks, "Max Flow Problem Introduction", 2023 https://www.geeksforgeeks.org/max-flow-problem-introduction/

[4] Brilliant, "Ford-Fulkerson Algorithm", Brilliant Math & Science Wiki, (n.d.). https://brilliant.org/wiki/ford-fulkerson-algorithm/

[5] A. Shapiro, D. Dentcheva, and A. Ruszczynski, "Lectures on stochastic programming: modeling and theory." SIAM, 2021.