Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра математики

**Магістерська робота робота**
освітній ступінь – магістр

на тему: **«РОЗРАХУНОК ВЕРХНЬОЇ МЕЖІ ВИХІДНОЇ ВАЛЕНТНОСТІ В МОДИФІКОВАНІЙ ТРАНСПОРТНІЙ ЗАДАЧІ»**

Виконав: студент 2-го року навчання
освітньої програми «Прикладна математика»,
спеціальності 113 Прикладна математика

Панасюк Роман Олегович

Керівник: Швай Н.О. доцент

Рецензент: Шаповал Н.В. доцент

Магістерська робота захищена
з оцінкою

_____

Секретар
ЕК_____
(підпис)
«_____»
_____ 20____р.

Київ - 2025

# Зміст

# Вступ

**Актуальність.** Моя власна мотивація, як автора даної роботи, щодо напрямку досліджень, народилася в контексті комерційного досвіду у сфері соціальних мереж, але крім цього вона є цінною у таких сферах як розподіл логістичних маршрутів, розподілення хмарних розрахунків, балансування навантаження в енергетичних мережах, а також телекомунікаційних мережах.

**Об'єкт дослідження.** Транспортна задача з додатковими структурними обмеженнями на можливість доставки товару від конкретного виробника, конкретному споживачу, а також обмеженнями на об'єм поставок товару.

**Предмет дослідження.** Процес мінімізації загального ліміту на об'єм перевезень від одного виробника враховуючи матрицю допустимості, а також обмеження на об'єм перевезень між конкретною парою виробник-споживач.

**Мета дослідження.** Метою даної роботи є розробка математичної моделі та алгоритмічного розв'язку для модифікованої транспортної задачі, з додатковими обмеженнями на максимальну кількість перевезень між конкретною парою виробник-споживач, та матрицею допустимості, що визначає структуру сполучень виробників та споживачів, в якій нам необхідно мінімізувати ліміт $L$, який визначає максимальну кількість поставок від кожного виробника усім споживачам.

**Завдання дослідження:**

1. Проаналізувати класичну транспортну задачу та її модифікації з додатковими обмеженнями.
2. Сформулювати математичну модель задачі з обмеженням на кількість поставок ($L$) і з урахуванням структури сполучень.
3. Розробити метод, що дозволяв би перевіряти чи є можливою доставка при фіксованому значенні $L$.
4. Реалізувати програмний алгоритм пошуку мінімального значення $L$, що дозволяє, виконуючи усі наявні обмеження, задовольнити попит усіх споживачів.
5. Реалізація експериментів, для перевірки ефективності запропонованого підходу, та аналізу його результатів.
6. Фінальний аналіз, та оцінка впливу структури сполученості, та обсягів попиту на ефективність виконання.

**Наукова новизна.** Набір досліджуваних в межах даної роботи додаткових обмежень в транспортній задачі не був досліджений раніше, до того ж суть задачі полягає в мінімізації загального обмеження на виробників, що в комбінації

утворює унікальну задачу. Розробка алгоритмічного розв'язку дозволить зробити перший крок в дослідженні подібних задач, а також створить підгрунтя для наступних робіт.

**Практичне значення.** Розв'язок даної задачі може бути використаний для балансування навантаження для бізнесу в онлайн сервісах, а також інших сферах. Її структура є легко розширюваною, та може бути вбудованою в рекомендаційну систему онлайн платформи, чи у систему планування постачань.

**Метод дослідження.** В рамках роботи використано наступні методи: мережевий аналіз, параметричний пошук, інкрементальне моделювання, а також експериментальний аналіз синтетичних даних.

**Структура роботи**. Спершу, для отримання широкого контексту, було розглянуто наявні джерела за дотичними темами, далі було розглянуто відомі методи для вирішення класичних варіацій задачі, після чого було запроновано алгоритм та кілька його варіацій. Фіналом роботи стало проведення експериментів на синтетичних даних.

# 1 Аналіз наукових джерел

## 1.1 Історія становлення транспортної задачі

Транспортна задача є однією з найдавніших з існуючих моделей оптимізації, що виникла як відповідь на практичні виклики, пов'язані з необхідністю ефективного розподілу ресурсів і зменшення витрат на перевезення вантажів, для якої перші концептуальні формулювання з'явилися, завдовго до того, коли були розроблені сучасні інструменти математичного аналізу чи теорії оптимізації.

Французький математик Гаспар Монж, який в той час був ще й військовим інженером, у 1781 році представив світу свою роботу, в якій сформулював задачу, що стосувалася пошуку найбільш ефективного способу транспортування ґрунту від кар'єрів, де він видобувався, до будівельних майданчиків. Ця задача фактично виникла через військові потреби того часу.

Основна ідея цієї задачі полягала в знаходженні найбільш ефективного плану переміщення маси. Формально потрібно було встановити відповідність між місцями видобутку та пунктами призначення, при цьому додатково потрібно було мінімізувати загальні витрати, які були пропорційні відстані, на яку транспортувалася одиниця вантажу. В рамках цієї постановки задачі вантаж не міг бути розділеним на частини, тобто кожна одиниця маси повинна була переміщуватися лише цілком, що і лягло в основу створення детермінованої моделі перевезення.

Попри свою революційність, задача Монжа виявилася складною для аналітичного розв'язання у загальному вигляді, через відсутність на той момент ефективних методів, що могли б її вирішити, і це обмежувало її застосування протягом майже двох століть, а її математичне вирішення стало можливим лише в XX столітті — з розвитком лінійного програмування, двоїстості, а також завдяки внеску інших дослідників, наприклад в контексті розв'язку задачі Канторовича.

Підсумовуючи, задача Монжа стала не лише початковою ланкою в еволюції транспортної задачі, а й продовжує бути предметом дослідження і сьогодні — зокрема, у формалізації моделей оптимального транспорту на безперервних просторах, у штучному інтелекті, комп'ютерному зорі, фінансовому моделюванні та багатьох інших сферах.

Одним із перших дослідників, які підійшли до транспортної задачі з прикладного та алгоритмічного боку, став радянський вчений Олексій Миколайович Толстой, який у 1930 році опублікував наукову працю в рамках видання "Планування транспорту за участі Народного комісаріату транспорту СРСР". Ним було запропоновано кілька практичних підходів розв'язання транспортної задачі, а прикладним застосуванням цих підходів було перевезення вантажів залізничною

мережею Радянського Союзу.

Зокрема, він дослідив два важливі приклади:

В одній з варіацій задачі, що включав два джерела, пункти призначення впорядковувалися за різницею відстаней до кожного з джерел. У такому випадку перше джерело забезпечувало потреби споживачів з початку списку, допоки його запаси не вичерпувалися, а решту потреб покривалося другим джерелом. Підхід був цінний через незалежність розподілу від конкретних обсягів попиту та пропозиції, через що цей підхід міг використовуватися протягом довгого часу без потреби у зміні чи оновленню плану перевезень.

Дргугою варіацією була так звана задача на кільцевій залізничній лінії, в рамках якої всі джерела і споживачі були розташовані вздовж одного кільцевого маршруту. Саме для такого формулювання було вперше застосовано критерій від'ємного циклу, як ознаку оптимальності розв'язку, що дозволяло швидко знаходити оптимальні маршрути, при цьому не використовуючи прямий підхід перебору варіантів.

Зрештою, Толстой об'єднав обидва підходи в єдиний еврістичний алгоритм, який був застосований до реального кейсу планування перевезень залізничною мережею в СРСР. Задача, яку він розв'язував, складалася з 10 джерел та 68 пунктів призначення, а також існувала додаткова умова у вигляді 155 існуючих шляхів між ними, які фактично визначалися тогочасною залізничною мережею.

Ще одним науковцем, який займався першими кроками в теорії транспортних задач, був Леонід Віталійович Канторович. Це був радянським математиком, що став лауреатом Нобелівської Премії. У одній зі своїх робіт, у 1939 році ним було запропоновано використати методи лінійного програмування для задач розподілу ресурсів з додатковими обмеженнями. Сформована Канторовичем математична модель, в якій було додано обмеження кількості ресурсів, мали розподілитися між кількома процесами, таким чином, щоб максимізувати потенційний корисний ефект. [1]

Незалежно від Канторовича, американський математик і співробітник Бюро перепису США, Френк Гічкок, у 1941 році вперше навів формалізований опис траспортної задачі. У свої роботі він розглянув питання розподілу товару між групою постачальників та групою споживачів, з урахуванням вартості перевезень між кожною парою постачальник-споживач. Ця модель містила в собі такі змінні, як кількість товару, яку необхідно перевезти а також обмеження по запасах виробників. Ще одним із нововведень була матрична інтерпретація задачі, яке в наші часи вважається стандартним форматом у транспортних задачах. [2]

Під час Другої світової війни у США гостро стояла потреба в оптимізації логістичних процесів для забезпечення військ. Одним з провідних спеціалістів

цієї галузі став Тьяллінг Купманс, що працював у системі військово-економічного планування, та активно досліджував потоки у складних мережах постачання. По завершенню війни, йому вдалося ознайомитися з працями Канторовича, і саме це допомогло популяризувати їх в західному світі, що заклало основу для розвитку "лінійного програмування".

Поява симплекс-методу стала кульмінацією в розвитку задач оптимізації, через те, що він був першим універсальним методом їх розв'язання. Його розробив Джорж Данциг у 1947 році. Його найбільшою перевагою була можливість знаходження оптимального значення цільової функції, при наявності великої кількості обмежень. Поява симплекс методу дозволило формалізувати транспортну задачу, як задачу лінійного програмування, з наявною спеціальною структурою та відповідно розв'язувати її за допомогою існуючих інструментів оптимізації. Згодом було запропоновано спеціальні методи саме для транспортної задачі, такі як метод північно-західного кута, метод мінімальної вартості та метод потенціалів. Ці методи враховували структуру задачі і дозволяли значно пришвидшити процес пошуку розв'язку порівняно з використанням загального симплекс-методу. Завдяки цим досягненням, активно почали розвиватися модифікації транспортних задач: задача перевалки, задача з обмеженнями на постачання, багаторівневі задачі транспортування, та багато інших. Наслідком розвитку цих задач було формування окремої підгалузі у теорії оптимізації - оптимізація транспортних систем, яка активно мала застосування в економіці, логістиці, інформаційних мережах та інших. Згодом, наявна сіткова структура, що собою формує двочастковий граф постачальників та споживачів, спровокувала розвиток методів, що враховували саме ці особливості. Такий підхід дозволив поглянути на задачу не лише з боку правильності розв'язку, а й з боку мінімізації обчислювальних затрат. Широкого практичного застосування набув метод Форда-Фалкерсона, що дозволяє знайти потік у мережі, та побудувати план оптимального розподілу ресурсів. [3,4,5,6]

Зростання потужностей обчислювальних систем, стало причиною нового імпульсу розвитку транспортної задачі, через що, її застосування виходить за межі логістики, в такі сфери як: енергетика, штучний інтелект, урбаністика, охорона здоров'я.

## 1.2 Класична транспортна задача

### 1.2.1 Постановка задачі

Класична транспортна задача формалізує процес мінімізації витрат на перевезення однорідного чи взаємозамінного вантажу від виробників до споживачів, а її метою є визначення оптимальний розподілу продукції від постачальників до споживачів, виконавши усі наявні додаткові обмеження. [8]

Нехай маємо $m$ виробників $A_1, A_2, \ldots, A_m$, у кожному з яких є запас вантажу обсягом $a_1, a_2, \ldots, a_m$ одиниць відповідно. Також розглянемо $n$ споживаічів $B_1, B_2, \ldots, B_n$, кожен з яких потребує відповідно $b_1, b_2, \ldots, b_n$ одиниць вантажу. Припускаємо, що загальний обсяг постачання дорівнює сукупному попиту:

$$\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j,$$

тобто задача є *збалансованою*.

Нехай $c_{ij}$ — вартість перевезення одиниці вантажу від $A_i$ до $B_j$. Ці вартості утворюють матрицю:

$$C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix}$$

І тепер, нашим завданням є заходження такого транспортуючого плану $X = [x_{ij}]$, який задовольнить весь попит у пунктах призначення, при цьому ще й загальні витрати на транспортування будуть мінімальними.

Позначимо $x_{ij}$ — кількість вантажу, що транспортується з $A_i$ до $B_j$. Розв'язок повинен задовольняти наступні умови:

- загальний обсяг вантажу, який відправляється з кожного пункту $A_i$, дорівнює запасу $a_i$:
$$\sum_{j=1}^{n} x_{ij} = a_i, \quad \forall i = 1, \ldots, m;$$
- загальна кількість вантажу, яку отримує кожен пункт $B_j$, дорівнює попиту $b_j$:
$$\sum_{i=1}^{m} x_{ij} = b_j, \quad \forall j = 1, \ldots, n;$$
- усі змінні невід'ємні:
$$x_{ij} \geq 0, \quad \forall i, j.$$

Метою є мінімізація загальної вартості перевезень:

$$C = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \to \min .$$

Варто зазначити, що описана модель є задачею лінійного програмування, хоча транспортна задача має кілька особливостей:

- всі коефіцієнти при змінних у системі обмежень дорівнюють одиниці;
- з $m + n$ обмежень лише $m + n - 1$ є лінійно незалежними враховуючи зазначену вище умову балансу;
- отже, кількість базисних змінних дорівнює $m + n - 1$, а кількість вільних змінних відповідає: $mn - (m + n - 1) = (m - 1)(n - 1)$

**Умови допустимості та оптимальності класичної транспортної задачі**

У процесі розв'язку транспортної задачі, слід відокремлювати поняття допустимого та оптимального планів перевезень, в межах заданої системи.

**Табличне представлення транспортної задачі**

Таблична форма подання транспортної задачі є однією з найбільш зручних форм, враховуючи чітку та виражену сітчасту структуру. Така форма широко застосовується в теоретичному аналізі, а також на практиці, через її інтуїтивність, та простоту сприйняття, адже вона дуже наочно відображає взаємозв'язки між постачальниками та споживачами. Також аналогічним чином можна представити і інші сутності, такі як вартість перевезень між парою виробник-постачальник, а також фінальний план транспортування. [8]

У класичній постановці:

- Рядки таблиці відповідають постачальникам $A_1, A_2, \ldots, A_m$;
- Стовпці — споживачам$B_1, B_2, \ldots, B_n$;
- У клітинках таблиці розміщується вартість перевезення $c_{ij}$ одиниці вантажу від $A_i$ до $B_j$;
- У правому стовпці зазначається обсяг запасів $a_i$ для кожного постачальника;
- У нижньому рядку — потреба $b_j$ кожного споживача.

Таблиця вихідних даних має наступну загальну форму:

| | $B_1$ | $B_2$ | $B_3$ | **Запас** $a_i$ |
|---|---|---|---|---|
| $A_1$ | $c_{11}$ | $c_{12}$ | $c_{13}$ | $a_1$ |
| $A_2$ | $c_{21}$ | $c_{22}$ | $c_{23}$ | $a_2$ |
| $A_3$ | $c_{31}$ | $c_{32}$ | $c_{33}$ | $a_3$ |
| **Попит** $b_j$ | $b_1$ | $b_2$ | $b_3$ | |

Табл. 1: Загальне табличне представлення транспортної задачі

На прикладі нижче, розглянемо конкретні числові значення вартостей перевезень, запасів та попиту:

| | $B_1$ | $B_2$ | $B_3$ | **Запас** $a_i$ |
|---|---|---|---|---|
| $A_1$ | 2 | 4 | 5 | 20 |
| $A_2$ | 3 | 1 | 7 | 30 |
| $A_3$ | 4 | 6 | 2 | 25 |
| **Попит** $b_j$ | 25 | 15 | 35 | |

Табл. 2: Приклад табличного представлення вихідних даних

Така таблиця є базою для формування допустимого плану транспортування, що задовольняє умови балансу запасів і попиту, та мінімізує загальні витрати на перевезення.

**Умови допустимості.** План транспортування $X = [x_{ij}]$ вважається **допустимим**, за умови, якщо він задовольняє такі умови:

1. **Баланс постачання:**

$$\sum_{j=1}^{n} x_{ij} = a_i, \quad \forall i = 1, 2, \ldots, m,$$

   де $a_i$ — обсяг продукції у $i$-му пункті відправлення.

2. **Баланс попиту:**

$$\sum_{i=1}^{m} x_{ij} = b_j, \quad \forall j = 1, 2, \ldots, n,$$

   де $b_j$ — потреба $j$-го пункту призначення.

3. **Невід'ємність:**

$$x_{ij} \geq 0, \quad \forall i, j.$$

При виконанні умова балансу:

$$\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j,$$

задача називатиметься *збалансованою* або *закритою*. В іншому випадку вона може бути приведеною до збалансованої за допомогою додавання штучного пункту постачання або споживання з нульовими витратами на транспортування.

**Умови оптимальності.** Допустимий план вважається **оптимальним**, якщо він мінімізує загальну вартість перевезень:

$$C = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \to \min,$$

де $c_{ij}$ — вартість перевезення одиниці вантажу з $A_i$ до $B_j$.

Оптимальність плану може бути визначеною за допомогою *методу потенціалів*. Для цього вводяться потенціали $u_i$ для виробників та $v_j$ для споживачів. Для усіх базисних клітинок $(i, j)$, де $x_{ij} > 0$, виконується:

$$u_i + v_j = c_{ij}.$$

Після обчислення потенціалів для всіх не базисних клітинок перевіряється умова:

$$u_i + v_j \leq c_{ij}, \quad \forall (i, j) \notin \text{базис}.$$

Якщо така умова виконується для всіх не базисних клітинок, то план є оптимальним. Якщо ж існує хоча б одна клітинка, для якої $u_i + v_j < c_{ij}$, — це означає, що план може бути покращеним, і він не є оптимальним. Даний підхід ґрунтується на принципах двоїстості задачач лінійного програмування, при застосуванні яких є можливим ефективна реалізація пошуку оптимального рішення.

### 1.2.2 Матриця допустимості

Стандартна постановка задачі за замовченням припускає можливість транспортування вантажу від будь якого якого виробника, будь якому споживачу, але в практичних випадках, які обумовлюються реальним життям, така повна сполученість, може не реалізовуватися. Причин може бути безліч: технічні обмеження, відсутність транспортних шляхів або юридичні правила. Для формалізації таких обмежень, стандартна модель розширюється бінарною матрицею, що відома як матриця допустимих сполучень, або матриця зв'язності.

Позначимо її як $R = [r_{ij}]$, де кожен елемент визначає, чи можлива доставка від постачальника $A_i$ до споживача $B_j$:

$$r_{ij} = \begin{cases} 1, & \text{якщо } A_i \text{ може доставляти до } B_j, \\ \\ 0, & \text{якщо сполучення відсутнє.} \end{cases}$$

Така матриця формулює додаткові обмеження на змінні $x_{ij}$ у транспортній задачі: якщо $r_{ij} = 0$, то $x_{ij} = 0$. Отже, вона фіксує можливу структуру перевезень і перетворює загальну модель на задачу в двочастковому графі.

Приклад матриці допустимості для трьох постачальників і чотирьох споживачів подано нижче:

| | $B_1$ | $B_2$ | $B_3$ | $B_4$ |
|---|---|---|---|---|
| $A_1$ | 1 | 0 | 1 | 1 |
| $A_2$ | 0 | 1 | 1 | 0 |
| $A_3$ | 1 | 1 | 0 | 1 |

Табл. 3: Матриця допустимості $R = [r_{ij}]$

Згідно з матрицею допустимості, маємо такі сполучення між постачальниками та споживачами:

- Постачальник $A_1$ може доставляти до $B_1$, $B_3$ та $B_4$, але не має зв'язку з $B_2$.
- Постачальник $A_2$ має зв'язок лише з $B_2$ та $B_3$, а доставка до $B_1$ і $B_4$ заборонена.
- Постачальник $A_3$ може обслуговувати $B_1$, $B_2$ та $B_4$, проте не має доступу до $B_3$.

Отже, матриця допустимості $R$ задає структуру дозволених перевезень. Дана матриця буде розглянута в наступних розділах, в контексті одного з класичних обмежень транспортної задачі.

### 1.2.3 Графова інтерпретація транспортної задачі

Одним із ключових та фундаментальних підходів до аналітичного та алгоритмічного розв'язання транспортної задачі є інтерпретація транспортної задачі як графа. Завдяки можливості розглянути модель як задачу на графі, формалізація задачі суттєво спрощується. Також нам відкривається широкий спектр методів з теорії графів, а також мережевих потоків.

У такій інтерпретації транспортна задача представляється як орієнтований двочастковий граф $G = (V, E)$, де множина вершин $V$ складається з двох підмножин: $A = \{A_1, A_2, \ldots, A_m\}$ — вершини, що відповідають постачальникам, та $B = \{B_1, B_2, \ldots, B_n\}$ — вершини, що відповідають споживачам. Дуги графа $(A_i, B_j) \in E$ в свою чергу будуть фактично відображати матрицю допустимості, яка вказує на можливість транспортування продукції від постачальника $A_i$ до споживача $B_j$.

Також кожній дузі графа приписується вагове значення $c_{ij}$, що відповідатиме вартості транспортування одиниці вантажу через конкретну дугу, а також змінна потоку $x_{ij}$, яка буде вказувати на кількість вантажу, що фактично транспортується цією дугою. Підсумовуючи, транспортна задача зводитиметься до знаходження такого потоку на графі, який задовольняв би обмеження на вхід і вихід у вершинах, тобто задовольнявся би баланс між постачанням та попитом, а також і мінімізувалися б сумарні витрати:

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

$$\text{за умов:} \quad \sum_{j=1}^{n} x_{ij} = a_i, \quad \sum_{i=1}^{m} x_{ij} = b_j, \quad x_{ij} \geq 0$$

Двочасткова структура графа транспортної задачі дозволяє нам використовувати методи, для задач максимального потоку, задач мінімального потоку з мінімізацією вартості перевезень, алгоритмів на основі евристики для пошуку шляху в мережі. Такими алгоритмами є: алгоритм Форда-Фалкерсона та його модифікації, алгоритм Беллмана-Форда, а також так званий метод потенціалів.

Ще однією перевагою графового представлення є можливість візуалізації самої задачі, аналізу та різноманітних властивостей задачі. До прикладу, можна встановлювати наявні обмеження, що знижують ефективність розподілу, або ідентифікувати цикли, що використовуються для покращення плану в методі потенціалів. Загалом, графове представлення є універсальним, адже на його основі можливо реалізувати різні варіації транспортної задачі, таких як, обмеження пропускної здатності, обмеження часовими вікнами доставки, варіація з динамічними потоками, або обмеження на кількість постачань.

Отже, з допомогою графової інтерпретації не лише спрощується математична постановка транспортної задачі, а й відкриваються додаткові методи реалізації в обчислювальних системах сучасного світу.

Нижче наведемо приклад графового представлення.

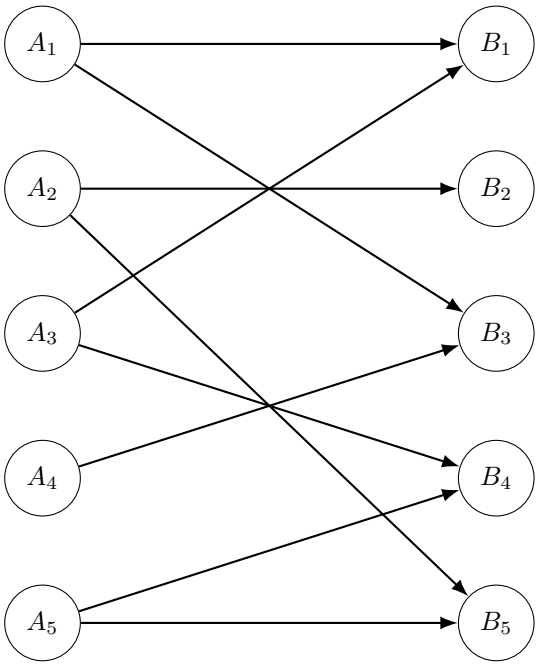



**Рисунок:** Графова модель транспортної задачі. Вершини $A_i$ відповідають постачальникам, вершини $B_j$ — споживачам. Орієнтовані дуги показують допустимі напрямки перевезень.

## 1.3 Задача мінімізації витрат на перевезення

Задача мінімізації витрат на перевезння, також відома як задача Гічкока–Купманса, полягає у виборі такого плану перевезень, який мінімізує загальні витрати, задовольнивши обмеження обсягів постачання, а також необхідного попиту. Френк Гічкок у 1941 році вперше сформулював матричну модель цієї задачі, в термінах лінійного програмування. Згодом був розроблений метод "Крокуючого каменя який мав на меті замінити симплекс метод, а в 1963 році була створена адаптація класичного симплекс-методу саме для транспортної задачі. Пізніше Ф.Кляйн доповнив зазначені вище підходи для вирішення задачі мінімальних грошових потоків, висвітливши їх використання у проблемах транспортування. Згодом Вонг Квак подав вдосконалений підхід, в якому комбінувалися цілі на базі goal programmin, а С. Ахуджа у 1986 імплементував алгоритм для мінімаксної транспортної задачі, націлену зменшити найбільші одиничні витрати. В тому ж році Дж. Каррін підсвітив проблему заборонених маршрутів у транспортній моделі, в той час як М. Салтан та Солтан із Гоялом у 1988 році вивчали початкові базисні рішення, а також виродження у траспортній моделі. Пізніше Аршам із Ханом, а згодом Кірк зі Статтіром, реалізували simplex-подібні та циклічні методи побудови початкового розв'язку, та його вдосконалення, що власне заклало підгрунтя сучасних методів розв'язання СМТР [9].

## 1.4 Задача мінімізації витрат на перевезення зі змішаними обмеженнями

На основі потреб реального світу, виникає необхідність у введенні додаткових, змішаних умов, крім обмеження на постачання та попит в задачі мінімізації витрат на перевезення. Це може бути обумовлено потребами планування виробництва, управління запасами чи навіть інвестиційним аналізом. Складність таких задач полягає у проблематиці їх точного розв'язання. Цікавим феноменом є принцип "more for less"(MFL), який полягає у нижчій кількості загальних витрат, при збільшенню кількості вантажу, що транспортується. На відміну від класичного СМТР, основою задачею MFL є мінімізація загальних витрат, а не максимізація обсягу перевезень. Вперше MFL було змодельовано у роботах арнза та Клінгмана у 1971[9].

## 1.5 Задача з мінімізації максимального часу перевезення

Задача мінімізації максимального часу перевезення, відома як Bottleneck Transportation Problem, (BTP) додає до моделі транспортної задачі параметри у вигляді часу доставки $t_{ij}$ для кожного маршруту та прагне мінімізувати найбільший серед них.

Даний підхід є критично важливим для вантажів, що схильні до швидкого псування, військової логістики та надзвичайних перевезень, де задля забезпечення підвищеної надійності оптимізується найбільший за тривалістю інтервал доставки. Вперше ВТР була досліджена Гаммером у 1969 році (time TP). Надалі Гарфінкель і Рау (1971), Шварц (1971), Шарма та Своруп (1977–1978), Сешан і Тікекар (1980), Кханна з колегами (1983) та Ісерманн (1984) поглибили аналіз цієї моделі, а у 1991 році Вардараджан представив оптимальний алгоритм для випадку $2 \times n$. [9]

## 1.6 Багатокритеріальна транспортна задача

Багатокритеріальна транспортна задача Multi-objective Transportation Problem (MOTP), є розширенням стандартного формулювання транспортної задачі, введенням $K$ цільових функцій

$$F^k(x) = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij}^k \, x_{ij}, \quad k = 1, \ldots, K,$$

які на практиці не рідко перебувають у конфлікті. З метою знаходження Парето-оптимальних розв'язків можна застосувати різні методи: лексикографічне цільове програмування, генетичні алгоритми[4] та утилітарні функції, інтерактивні алгоритми, нечітке програмування. Завдання полягає у формуванні множини ефективних рішень і виборі компромісного варіанта, який найбільш оптимально збалансовуватиме всі наявні критерії [9].

## 1.7 Загальна форма задачі MILP та метод розв'язання Branch-and-Cut

Змішане цілочисельне лінійне програмування, що звучить англійською як Mixed Integer Linear Programming, (MILP) — це такий тип задачі, що поєднує дійсні змінні та цілочисельні, за наявної цільової функції при чому всі обмеження лишаються лінійними [4].

**Математичне формулювання задачі MILP:**

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^\top x \\ \text{за умов:} \quad & Ax \leq b, \quad \text{(лінійні нерівності)} \\ & x_j \in \mathbb{Z}, \qquad\qquad \forall j \in \mathcal{I}, \\ & x_j \in \mathbb{R}, \qquad\qquad \forall j \notin \mathcal{I}, \end{aligned}$$

де:

- $x \in \mathbb{R}^n$ — вектор змінних;
- $c \in \mathbb{R}^n$ — вектор коефіцієнтів цільової функції;
- $A \in \mathbb{R}^{m \times n}$ — матриця коефіцієнтів лівих частин обмежень;
- $b \in \mathbb{R}^m$ — вектор правих частин обмежень;
- $\mathcal{I} \subseteq \{1, 2, \ldots, n\}$ — індекси змінних, які повинні бути цілочисельними.

Цей клас задач включає в себе, як звичайне лінійне програмування (LP), так і повністю цілочисельне (ILP), та є дуже потужним інструментом для моделювання транспортних, логістичних і виробничих задач.

**Алгоритм розв'язання: Branch-and-Cut**

Ефективним сучасним підходом до розв'язання задач MILP є алгоритм *Branch-and-Cut*, який поєднує:

- техніку **гілок і меж** (Branch-and-Bound),
- додавання **відсічних площин** (Cutting Planes).

**Суть підходу.** Branch-and-Cut будує дерево пошуку, вузли якого — часткові задачі з додатковими обмеженнями відтинаючи, нерелевантні області, вирішуючи LP-релаксації на кожному вузлі, що дозволяє поступово знаходити оптимальне цілочисельне рішення.

**Покроковий опис алгоритму Branch-and-Cut**

1. **Розв'язання LP-релаксації:**

   $$\min\{c^\top x : Ax \leq b\}$$

   без урахування цілочисельних обмежень. Якщо отримане рішення $x^*$ — цілочисельне, задача розв'язана.

2. **Перевірка цілочисельності:** якщо деякі $x_j^*$ не належать $\mathbb{Z}$ для $j \in \mathcal{I}$, виконується гілкування по дробових змінних. Для цього створюються дві нові задачі з додатковими обмеженнями:

   $$x_j \leq \lfloor x_j^* \rfloor \quad \text{та} \quad x_j \geq \lceil x_j^* \rceil.$$

3. **Додавання відсічних площин (cutting planes):** у разі виявлення, що розв'язок $x^*$ порушує специфічну властивість цілочисельних розв'язків, до задачі додається нове лінійне обмеження:

$$\alpha^\top x \leq \beta, \quad \text{де} \quad \alpha^\top x^* > \beta.$$

Такі площини виключають дробові розв'язки без виключення допустимих цілочисельних.

4. **Продовження пошуку:** кожна з підзадач обробляється рекурсивно — або гілкуванням, або додаванням нових площин.

5. **Обрізання (pruning):** якщо нижня межа (оцінка LP-релаксації) підзадачі перевищує найкращу відому верхню межу (значення цілочисельного розв'язку), вузол дерева відкидається.

6. **Завершення:** після обробки всіх вузлів вибирається найкраще допустиме рішення серед знайдених.

**Внутрішні методи для LP-релаксацій**

Для розв'язання LP-релаксацій на кожному кроці зазвичай застосовуються:

- **Симплекс-метод (dual simplex)** — ефективний при додаванні нових обмежень;
- **Метод внутрішніх точок (interior point)** — менш чутливий до структури задачі, але менш ефективний у MILP.

**Переваги та практичне застосування**

Алгоритм Branch-and-Cut дозволяє з високою точністю знаходити глобальні оптимальні рішення для широкого класу дискретно-контрольованих систем. Він застосовується в сучасних розв'язувачах, таких як `CBC`, `Gurobi`, `CPLEX`, `SCIP`, і підтримується інтерфейсами `PuLP`, `OR-Tools`, тощо.

## 1.8 Методи розв'язання транспортної задачі у графовій інтерпретації

### 1.8.1 Алгоритм Форда–Фалкерсона

Алгоритм Форда–Фалкерсона призначений для знаходження максимально можливого потоку від джерела $s$ до стоку $t$ в орієнтованому графі $G = (V, E)$ з пропускними здатностями $c_{uv}$. Його ключові кроки:

1. Ініціалізувати нульовий потік $f(u, v) = 0$ на всіх ребрах.
2. Побудувати залишкову мережу $G_f$ з ємностями $c_f(u, v) = c(u, v) - f(u, v)$ та $c_f(v, u) = f(u, v)$.
3. Поки в $G_f$ існує шлях доповнення $P$ від $s$ до $t$:
   - Знайти $\Delta = \min_{(u,v)\in P} c_f(u, v)$.
   - Для кожного ребра $(u, v) \in P$ збільшити $f(u, v)$ на $\Delta$ та зменшити $f(v, u)$ на $\Delta$.
   - Оновити залишкові ємності $c_f$.
4. Після завершення ітерацій $f$ є максимальним потоком.

Недоліки: час роботи залежить від способу пошуку шляху доповнення і може бути експоненційним без додаткових обмежень (для гарантованого поліноміального часу зазвичай використовують модифікації, наприклад Edmonds–Karp).

### 1.8.2 Алгоритм Беллмана–Форда

Алгоритм Беллмана–Форда дозволяє знайти найкоротші відстані $d(v)$ від фіксованої початкової вершини $s$ до всіх інших вершин графа $G = (V, E)$, у якому ваги ребер $w(u, v)$ можуть бути від'ємними (за умови відсутності від'ємних циклів). Основна ідея полягає в багаторазовому «релаксуванні» ребер, тобто поступовому поліпшенні оцінок відстаней.

1. **Ініціалізація.**

   $$d(s) := 0, \quad d(v) := +\infty \quad \forall\, v \in V \setminus \{s\}.$$

   Це означає, що на початку вважаємо відстань до всіх вершин, окрім джерела $s$, невідомою (безкінечною).

2. **Релаксування ребер.** Повторюємо наступний крок рівно $|V| - 1$ разів (де $|V|$ — кількість вершин):

   для кожного ребра $(u, v) \in E$ : якщо $d(u)+w(u, v) < d(v)$, то $d(v) := d(u)+w(u, v)$.

   Кожна ітерація забезпечує, що усі шляхи з at most $k$ ребер знайдені після $k$ кроків. Після $|V| - 1$ ітерацій гарантовано враховані всі шляхи без циклів.

3. **Перевірка від'ємних циклів.** Для виявлення циклів з від'ємним сумарним ваговим значенням виконуємо ще одне релаксування всіх ребер:

   для кожного $(u, v) \in E$ : якщо $d(u)+w(u, v) < d(v)$, то існує від'ємний цикл.

   Наявність такого оновлення свідчить про те, що можна зменшити відстань нескінченно, тобто граф містить від'ємний цикл.

**Коментарі.**

- Число ітерацій $|V|-1$ обране тому, що найкоротший шлях у графі без циклів містить щонайбільше $|V|-1$ ребер.
- Релаксування — це операція оновлення оцінки відстані через перевірку шляху через поточне ребро.
- Складність алгоритму $O(|V|\cdot|E|)$, що робить його повільнішим за Dijkstra у випадку тільки невід'ємних ваг, але є універсальним для від'ємних значень.
- У контексті транспортної задачі метод Беллмана–Форда застосовують для обчислення потенціалів у методі MODI, щоб уникнути від'ємних циклів у залишковій мережі.

Застосування: у методі потенціалів транспортної задачі використовують відстані Беллмана–Форда для перерахунку потенціалів $u_i, v_j$ та перевірки від'ємних циклів.

### 1.8.3 Метод потенціалів

Метод потенціалів служить для оцінки оптимальності і покращення початкового базисного рішення транспортної задачі:

1. Нехай є початковий базисний план з $m+n-1$ заповненими клітинками $(i,j)$ і невикористані небазисні клітинки.
2. Ввести потенціали $u_i$ для рядків та $v_j$ для стовпців так, щоб для кожної базисної клітинки виконувалося

   $$u_i + v_j = c_{ij}.$$

   Прийняти $u_1 = 0$ і послідовно обчислити інші потенціали.
3. Для кожної небазисної клітинки $(i,j)$ обчислити оцінку

   $$\Delta_{ij} = u_i + v_j - c_{ij}.$$

4. Якщо $\Delta_{ij} \le 0$ для всіх небазисних клітинок, план є оптимальним. Інакше:
   - Обрати клітинку з $\Delta_{ij} > 0$.
   - Сформувати замкнений цикл через базисні клітинки, по черзі додаючи та віднімаючи одиницю потоку.
   - Оновити план, перемістивши потік по циклу на мінімальну кількість у від'ємних клітинках.

5. Повторювати оцінку та корекцію, доки всі $\Delta_{ij} \leq 0$.

Цей метод забезпечує гарантоване поліпшення початкового плану та веде до оптимального рішення транспортної задачі.

### 1.8.4 Алгоритм Shortest Augmenting Path (SAP)

Нехай задано орієнтований граф $G = (V, E)$ з пропускною здатністю $c(u, v)$ на кожному ребрі $(u, v) \in E$, джерелом $s \in V$ та стоком $t \in V$.

Мета — знайти допустимий потік $f$, який:

- задовольняє обмеження пропускної здатності:

$$0 \leq f(u, v) \leq c(u, v), \quad \forall (u, v) \in E;$$

- виконує закон збереження потоку:

$$\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w), \quad \forall v \in V \setminus \{s, t\};$$

- максимізує загальний потік з джерела:

$$|f| = \sum_{v \in V} f(s, v).$$

**Крок 1. Ініціалізація**

Задати $f(u, v) := 0$ для всіх $(u, v) \in E$. Побудувати початкову залишкову мережу $G_f$, у якій:

- кожне ребро $(u, v)$ має залишкову здатність $c_f(u, v) = c(u, v) - f(u, v)$;
- якщо $f(u, v) > 0$, то додається зворотне ребро $(v, u)$ з $c_f(v, u) = f(u, v)$.

**Крок 2. Пошук найкоротшого доповнюючого шляху**

Використати алгоритм BFS (див. підрозділ нижче) для пошуку найкоротшого шляху $P$ від $s$ до $t$ у залишковій мережі $G_f$. Критерій мінімальності — найменша кількість ребер у шляху.

Якщо такого шляху не існує — перейти до Кроку 6.

**Крок 3. Обчислення приросту потоку**

Обчислити величину приросту потоку по шляху $P$:

$$\delta = \min_{(u,v) \in P} c_f(u, v),$$

де $c_f(u, v)$ — залишкова здатність ребра $(u, v)$.

**Крок 4. Збільшення потоку**

Оновити значення потоку $f$ на шляху $P$:

$$f(u,v) := f(u,v) + \delta, \quad \forall (u,v) \in P;$$

$$f(v,u) := f(v,u) - \delta, \quad \text{(зворотній потік)}.$$

**Крок 5. Оновлення залишкової мережі**

На основі оновленого потоку перебудувати залишкову мережу $G_f$ та повернутись до Кроку 2.

**Крок 6. Завершення**

Коли неможливо знайти доповнюючий шлях у залишковій мережі, алгоритм завершується. Поточне значення потоку $f$ є максимальним. Повернути:

$$|f| = \sum_{v \in V} f(s,v).$$

**Алгоритм пошуку в ширину (BFS)**

Алгоритм BFS (Breadth-First Search) — це стратегія обходу графа, яка забезпечує знаходження шляху з найменшою кількістю ребер. У задачі пошуку доповнюючого шляху він працює наступним чином:

1. Ініціалізувати чергу $Q := \{s\}$ та відмітити $s$ як відвіданий.
2. Поки черга не порожня:
   - Витягнути вершину $u$ з черги.
   - Для кожного сусіда $v$ вершини $u$, якщо $c_f(u,v) > 0$ і $v$ не був відвіданий:
     - відмітити $v$ як відвіданий;
     - зберегти батьківське посилання $parent[v] := u$;
     - додати $v$ до черги.
3. Якщо $t$ досягнуто — за допомогою масиву $parent$ відновити шлях $P$ від $s$ до $t$.
4. Якщо $t$ не досягнуто — шляху не існує.

Таким чином, BFS гарантує знаходження найкоротшого шляху (в кількості ребер), що робить його ключовим компонентом алгоритму SAP.

### 1.8.5 Алгоритм Edmonds–Karp

Алгоритм Edmonds–Karp є спеціалізованим варіантом загального методу Форда–Фалкерсона для розв'язання задачі максимального потоку в орієнтованій мережі. Основною відмінністю є використання пошуку в ширину (BFS) для знаходження доповнюючих шляхів у залишковій мережі, що гарантує поліноміальну складність алгоритму.

**Вхідні дані:**

- Орієнтований граф $G = (V, E)$;
- Пропускна здатність кожного ребра $c(u, v) \geq 0$;
- Джерело потоку $s \in V$;
- Стік потоку $t \in V$.

**Вихід:**

Максимальний потік $f$ із $s$ до $t$, що задовольняє:

$$0 \leq f(u, v) \leq c(u, v), \quad \forall (u, v) \in E,$$
$$\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w), \quad \forall v \in V \setminus \{s, t\}.$$

**Кроки алгоритму:**

1. Ініціалізувати потік:

$$f(u, v) := 0 \quad \forall (u, v) \in E.$$

2. Побудувати початкову залишкову мережу $G_f$, у якій:

$c_f(u, v) := c(u, v) - f(u, v)$, та додати зворотні ребра з $c_f(v, u) := f(u, v)$.

3. Повторювати, доки існує доповнюючий шлях $P$ з $s$ до $t$ у $G_f$:

   (а) Знаходити найкоротший шлях $P$ з $s$ до $t$ у залишковій мережі $G_f$ за допомогою BFS (Breadth-First Search);

   (б) Обчислити приріст потоку:

$$\delta := \min_{(u,v) \in P} c_f(u, v).$$

(в) Збільшити потік уздовж шляху:

$$f(u,v) := f(u,v) + \delta, \quad f(v,u) := f(v,u) - \delta, \quad \forall (u,v) \in P.$$

(г) Оновити залишкову мережу відповідно до нового значення потоку.

4. Коли доповнюючих шляхів не залишилось, поточне значення $f$ є максимальним потоком. Повернути:

$$|f| = \sum_{v \in V} f(s,v).$$

**Складність алгоритму:**

Алгоритм Edmonds–Karp має поліноміальну складність:

$$\mathcal{O}(V \cdot E^2),$$

оскільки кожне ребро може бути насичене принаймні після $O(V)$ проходів BFS.

## 1.9 Підсумки, та визначення прогалин в літературі

Загалом, транспортна задача в силу кількості сфер в яких її можна застосувати, дає простір для накладення великої кількості різних додаткових умов, що створюють унікальні варіації даної задачі. В сучасних задачах мало дослідженими є варіації, що обмежують постачальників, у комбінацією із матрицею досутупу, що підкреслює дослідиницьку цінність даної роботи!

# 2 Розв'язок досліджуваної задачі

## 2.1 Постановка задачі

Нехай $A_i$, $i = 1, \ldots, N$, — виробники, а $B_j$, $j = 1, \ldots, M$, — споживачі з потребами $b_j$. Позначимо змінні

$$x_{ij} \;=\; \text{кількість товару, відправленого від } A_i \text{ до } B_j.$$

Нехай також задана матриця допустимості $R = [r_{ij}]$, де

$$r_{ij} = \begin{cases} 1, & \text{якщо між } A_i \text{ і } B_j \text{ існує сполучення,} \\ 0, & \text{інакше.} \end{cases}$$

Вводимо два додаткових обмеження:

1. Кожен виробник може відправити максимум $K$ одиниць кожному конкретному споживачу:

$$x_{ij} \;\le\; K\, r_{ij}, \quad \forall\, i = 1, \ldots, N,\; j = 1, \ldots, M.$$

2. Сумарно кожен виробник може відправити не більше $L$ одиниць:

$$\sum_{j=1}^{M} x_{ij} \;\le\; L, \quad \forall\, i = 1, \ldots, N.$$

При цьому необхідно задовольнити потребу кожного споживача:

$$\sum_{i=1}^{N} x_{ij} \;=\; b_j, \quad \forall\, j = 1, \ldots, M,$$

а також

$$x_{ij} \;\ge\; 0.$$

Метою задачі є мінімізація загального ліміту $L$:

$$\min_{x_{ij},\, L} \; L$$

що підлягає наведеним вище обмеженням.

## 2.2 Умови розв'язності задачі

Для того щоб задача мала допустиме розв'язання (тобто існувала матриця постачань $X = [x_{ij}]$, яка задовольняє всі обмеження), необхідно виконання наступних умов:

1. **Глобальна ресурсна достатність:**

$$\sum_{j=1}^{M} b_j \leq N \cdot L.$$

   Сумарна потреба всіх споживачів не повинна перевищувати загальний доступний обсяг постачання з боку виробників, обмежений величиною $L$ на кожного.

2. **Зв'язність попиту:**

$$\forall\, j \in \{1, \ldots, M\}, \quad \sum_{i=1}^{N} r_{ij} \geq 1.$$

   Кожен споживач повинен бути з'єднаний хоча б з одним виробником. Інакше постачання до нього неможливе за жодного сценарію.

3. **Локальна пропускна спроможність:**

$$\forall\, j \in \{1, \ldots, M\}, \quad \sum_{i=1}^{N} K \cdot r_{ij} \geq b_j.$$

   Навіть при максимальному обсязі $K$ з кожного доступного джерела, сума потенційних постачань повинна бути достатньою для покриття попиту $b_j$.

4. **Сумісність системи обмежень:** має існувати хоча б одна множина невід'ємних змінних $x_{ij}$, для якої виконуються всі такі умови:

$$\begin{aligned}
&x_{ij} = 0 \quad \text{якщо } r_{ij} = 0, \\
&0 \leq x_{ij} \leq K, \\
&\sum_{j=1}^{M} x_{ij} \leq L \quad \forall i = 1, \ldots, N, \\
&\sum_{i=1}^{N} x_{ij} = b_j \quad \forall j = 1, \ldots, M.
\end{aligned}$$

   Несумісність цієї системи означає, що жоден допустимий розподіл постачань не задовольняє всі вимоги задачі.

У випадку, коли хоча б одна з умов не виконується, задача може бути нерозв'язною в повному обсязі. У такому випадку ми будемо розглядати задачу пошуку найбільшої допустимої підмножини споживачів, для якої існує задовільне рішення при фіксованому значенні $L$.

## 2.3 Алгоритмічне вирішення задачі

### Покроковий опис реалізованих алгоритмів

Для вирішення поставленої задачі були реалізовані три підходи: два графові (Edmonds–Karp, Preflow Push) і один оптимізаційний на основі змішаного цілочисельного програмування (MILP). Усі методи модифіковано для знаходження мінімального допустимого значення $L$ при заданих обмеженнях.

**1. Edmonds–Karp (максимальний потік + двійковий пошук по $L$)**

1. Задати діапазон пошуку:

$$L_{\min} = \left\lceil \frac{\sum_j b_j}{N} \right\rceil, \quad L_{\max} = \sum_j b_j.$$

2. Поки $L_{\min} \leq L_{\max}$:
   (а) Встановити $L = \left\lfloor \frac{L_{\min}+L_{\max}}{2} \right\rfloor$;
   (б) Побудувати орієнтований граф:
   - дуги $s \to A_i$ з місткістю $L$;
   - дуги $A_i \to B_j$ з місткістю $K$ (якщо $r_{ij} = 1$);
   - дуги $B_j \to t$ з місткістю $b_j$.

   (в) Виконати алгоритм Edmonds–Karp:
   - за допомогою BFS знайти найкоротший шлях $s \to t$;
   - пропустити по ньому потік;
   - повторювати до вичерпання шляхів.

   (г) Якщо потік дорівнює $\sum_j b_j$, оновити $L^* = L$ і $L_{\max} = L - 1$;
   (д) Інакше $L_{\min} = L + 1$.
3. Після завершення двійкового пошуку отримане $L^*$ є оптимальним. Будується матриця постачань $x_{ij}$ на основі остаточного потоку.

**2. Preflow–Push (максимальний потік + двійковий пошук по $L$)**

1. Аналогічно, задати діапазон для двійкового пошуку по $L$:

$$L_{\min} = \left\lceil \frac{\sum_j b_j}{N} \right\rceil, \quad L_{\max} = \sum_j b_j.$$

2. Поки $L_{\min} \leq L_{\max}$:

   (а) Побудувати граф з обмеженнями $L$, $K$ та $b_j$ на відповідних дугах;

   (б) Виконати алгоритм Preflow–Push:

   - ініціалізувати потік з джерела $s$ до всіх суміжних вершин;
   - підтримувати надлишковий потік у вершинах та префлоу в графі;
   - застосовувати операції *push* (передача надлишку по дугах) і *relabel* (зміна висоти вершини) доти, доки існує надлишковий потік не у стоку;
   - завершити, коли весь потік досягне стоку $t$ або більше неможливо просунути потік.

   (в) Якщо досягнутий потік дорівнює $\sum_j b_j$, оновити $L^* = L$ і $L_{\max} = L - 1$;

   (г) Інакше $L_{\min} = L + 1$.

3. Після завершення пошуку $L^*$ є мінімальним допустимим лімітом. Потік між вузлами $A_i$ і $B_j$ формує матрицю постачань.

**3. MILP (PuLP + CBC або Google OR-Tools + SCIP)**

1. Оголосити змінні:

   - $x_{ij} \in \mathbb{Z}_{\geq 0}$ — кількість товару від $A_i$ до $B_j$;
   - $L \in \mathbb{Z}_{\geq 0}$ — змінна, яку потрібно мінімізувати.

2. Задати обмеження:

   - $\sum_j x_{ij} \leq L$ — обмеження на загальний обсяг відправки від кожного виробника;
   - $x_{ij} \leq K \cdot r_{ij}$ — обмеження на постачання та допустимість маршруту;
   - $\sum_i x_{ij} = b_j$ — забезпечення потреб кожного споживача.

3. Побудувати цільову функцію:

$$\min L$$

4. Передати задачу до солвера:

- **CBC (через PuLP)** або **SCIP (через OR-Tools)**;
- Використовується метод **Branch-and-Cut**:
  (а) спочатку виконується LP-релаксація;
  (б) якщо розв'язок дробовий — виконується гілкування (branching);
  (в) додаються лінійні обмеження (cutting planes);
  (г) розв'язок оновлюється до отримання найменшого цілочисельного $L$.

5. В результаті отримується оптимальний $L^*$ та відповідні значення $x_{ij}$.

**Підсумок і подальше використання**

Усі три алгоритмічні підходи — Edmonds–Karp, Preflow Push та MILP (через CBC / OR-Tools) — було реалізовано в рамках програмної частини дослідження. Кожен з методів модифіковано відповідно до специфіки задачі: введено обмеження $K$ на одиничні постачання, побудовано систему для перевірки допустимості $L$, а також реалізовано двійковий пошук або оптимізацію $L$ як змінної.

У фінальній частині дослідження планується провести порівняльний аналіз цих методів за такими критеріями:

- час виконання для різних розмірів задачі кількість виробників і споживачів;
- якість отриманих розв'язків;
- чутливість до структури вхідних даних, таких як щільність матриці $R$.

Для цього буде згенеровано серію тестових прикладів за допомогою окремого модуля генерації даних, а результати — представлені у вигляді таблиць і графіків.

## 2.4 Складність алгоритму

Нехай

$$R = \{(i, j) \mid r_{ij} = 1\}, \quad |R| \le NM.$$

Тоді у побудованому графі:

$$V = N + M + 2, \quad E = N + M + |R|.$$

Перевірка розв'язності (Edmonds–Karp) дає

$$O\big(V\,E^2\big) = O\big((N + M)\,(N + M + |R|)^2\big).$$

Бінарний пошук по $L$ на діапазоні до сумарного попиту $B = \sum_j b_j$ додає множник

$$O(\log B).$$

$$\boxed{O\big(\log B \;\times\; (N + M)\,(N + M + |R|)^2\big)}$$

В граничному випадку $|R| = NM$ це дає

$$O\big(\log B \;\times\; (N + M)\,(NM)^2\big).$$

# 3 Програмна реалізація

## 3.1 Вибір програмних інструментів для реалізації алгоритму

Для реалізації програмної частини дослідження було використано мову програмування **Python**, яка завдяки своїй зручності, для здійснення математичних досліджень, а також наявності великої кількості бібліотек є підходящою платформою для моделювання задачі, що розглядається в рамках даної роботи. Усі обчислювальні компоненти реалізовано у вигляді класів з модульною архітектурою, що забезпечує повторне використання і прозорість розв'язків.

Ключовим інструментом побудови математичної моделі, як задачі на графі є бібліотека **NetworkX**, що використовувалася для побудови орієнтованого графа потоку. В моделі кожен виробник і споживач подається як вузол, а логічні та кількісні обмеження — як дуги з відповідною пропускною здатністю.

Крім того, для альтернативного формулювання задачі було використано бібліотеки **Google OR-Tools** і **PuLP**, які дозволяють записати транспортну модель у вигляді задачі змішаного цілочисельного лінійного програмування (MILP). У рамках фреймворку OR-Tools розв'язок здійснювався за допомогою внутрішнього MILP-розв'язувача `SCIP`, тоді як PuLP взаємодіє з відкритим розв'язувачем `CBC`.

Для роботи із розрахунками, використовувалися бібліотеки **NumPy** та **Pandas**.

Візуалізація результатів здійснювалася за допомогою бібліотеки **Matplotlib** для графів, крім цього деякі таблиці були виведені в текстовому форматі завдяки відповідному методу класа, який відповідав за виведення результатів.

Уся розробка здійснювалась в інтерактивному хмарному середовищі **Google Colab**, що дало змогу поєднувати код, вивід, візуалізації та опис алгоритмів в одному документі. Це спростило процес розробки, тестування, документування та презентації результатів, а також це забезпечило додаткову безпеку, через відсутність залежності від конкретного фізичного носія.

## 3.2 Генерація вхідних даних

Для цілей тестування та порівняння алгоритмів було реалізовано спеціальний клас `DataGenerator`, який автоматично формує вхідні дані для задачі з урахуванням її обмежень.

Основні функціональні можливості генератора:

- Кількість виробників $N$ та кількість споживачів $M$ задаються як параметри;

- Матриця сполученості $R \in \{0,1\}^{N \times M}$ генерується випадковим чином з імовірністю наявності з'єднання (параметр $connection\_prob$), але з гарантією, що кожен споживач має хоча б одного постачальника;
- Вектор попиту $\vec{b} = (b_1, b_2, \ldots, b_M)$ може бути як рівномірно заданий (усі $b_j$ однакові), так і випадково згенерований в заданому діапазоні;
- Підтримується параметр `seed` для відтворюваності результатів при повторному запуску;
- Генерація реалізована в окремому методі `generate()`, який повертає пару об'єктів: матрицю $R$ та вектор попиту $b$.

Цей модуль дає змогу створювати як одиничні контрольні приклади, так і серії задач для статистичного порівняння методів при різних конфігураціях параметрів (щільність з'єднань, розподіл попиту, масштаб задачі).

## 3.3 Реалізація пошуку мінімального $L$

Основна логіка розв'язання задачі реалізована в класі `TransportSolver`, який побудований як універсальний модуль для роботи з різними алгоритмічними підходами. Клас дозволяє перевірити наявність розв'язку, знайти мінімальне значення $L$, побудувати матрицю постачань $x_{ij}$, а також аналізувати часткові розв'язні підграфи у випадку, коли повна задача є нерозв'язною.

Ключові особливості:

- Вхідними параметрами є:
  - вектор попиту $\vec{b}$;
  - матриця сполученості $R$;
  - параметр $K$ — максимальна кількість одиниць, яку виробник може відправити одному споживачу.
- Побудова графа потоку:
  - створюється орієнтований граф з джерелом $s$ та стоком $t$;
  - дуги $s \to A_i$ мають місткість $L$, що підбирається;
  - дуги $A_i \to B_j$ мають місткість $K$ (з урахуванням $r_{ij}$);
  - дуги $B_j \to t$ мають місткість $b_j$.

- Реалізовано пошук мінімального допустимого $L$ за допомогою двійкового пошуку в межах:

$$L_{\min} = \left\lceil \frac{\sum_j b_j}{N} \right\rceil, \quad L_{\max} = \sum_j b_j$$

  Для кожного значення $L$ перевіряється, чи можливо досягти повного потоку.

- Передбачено підтримку кількох алгоритмів для обчислення максимального потоку: `Edmonds-Karp`, `Preflow Push`, а також двох MILP-орієнтованих рішень — `CBC (PuLP)` і `SCIP (OR-Tools)`.

- Якщо повна задача не має розв'язку, клас знаходить найбільший допустимий підграф споживачів, який може бути задоволений при фіксованому $L$.

- Повертається словник із детальною інформацією про:
  - наявність розв'язку;
  - мінімальне значення $L$;
  - матрицю постачань;
  - список задоволених / незадоволених споживачів;
  - час виконання.

Таким чином, `TransportSolver` є гнучким інструментом для аналізу задачі при різних конфігураціях вхідних даних та методів розв'язання, і дозволяє легко проводити як одиничні експерименти, так і автоматизовані порівняльні тести, що і буде використано для побудови прикладів, які власне і будуть в наступному блоці.

## 3.4 Приклади

**Приклад 1**

Для демонстрації роботи алгоритму було згенеровано вхідні дані з використанням класу `DataGenerator`, який забезпечує контроль над структурою задачі. Було використано наступні параметри:

- Кількість виробників: $N = 5$;
- Кількість споживачів: $M = 5$;
- Обмеження на максимальну кількість одиниць, яку один виробник може відправити одному споживачеві: $K = 1$;
- Діапазон попиту: $d_j \in [1, 1]$ для всіх $j$, тобто попит є фіксованим і однаковим для кожного споживача;
- Ймовірність існування з'єднання між $A_i$ та $B_j$: $p = 0.5$.

На основі цих параметрів було згенеровано:

- Вектор попиту $b = (1, 1, 1, 1, 1)$;
- Матрицю сполученості $R \in \{0, 1\}^{5\times5}$, у якій кожен елемент $r_{ij}$ набуває значення 1 з імовірністю 0.5, а також гарантовано, що кожен споживач має хоча б одного постачальника.


Тепер розв'яжемо згенеровану задачу з використанням методу Shortest Augmenting Path (SAP). Нижче наведено результати виконання обчислень для заданої конфігурації вхідних даних.

```
==========  РЕЗУЛЬТАТ  ==========

Розв'язність повної задачі: ✅

Мінімальне L: 2

Матриця постачань:
     B_0  B_1  B_2  B_3  B_4
A_0    1    0    1    0    0
A_1    0    0    0    0    1
A_2    0    0    0    0    0
A_3    0    0    0    0    0
A_4    0    1    0    1    0

⏱ Час виконання алгоритму: 0.0180 секунд
```

Отже, задачу було розв'язано, було знайдено мінімальне L та матрицю постачань при заданій матриці сполученості.

**Приклад 2**

- Кількість виробників: $N = 10$;
- Кількість споживачів: $M = 10$;
- Обмеження на максимальну кількість одиниць, яку один виробник може відправити одному споживачеві: $K = 1$;
- Діапазон попиту: $d_j \in [2, 2]$ для всіх $j$, тобто кожен споживач має фіксований попит у 2 одиниці;
- Ймовірність існування з'єднання між $A_i$ та $B_j$: $p = 0{,}3$;

На основі цих параметрів було згенеровано:

- Вектор попиту $b = (2, 2, 2, 2, 2, 2, 2, 2, 2, 2)$;
- Матрицю сполученості $R \in \{0, 1\}^{10\times10}$, у якій кожен елемент $r_{ij}$ набуває значення 1 з імовірністю 0.3, причому гарантовано, що кожен споживач має хоча б одного постачальника.

📌 Матриця з'єднаності (R):

| | B_0 | B_1 | B_2 | B_3 | B_4 | B_5 | B_6 | B_7 | B_8 | B_9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A_0: | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| A_1: | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| A_2: | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| A_3: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A_4: | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| A_5: | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| A_6: | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| A_7: | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| A_8: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| A_9: | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

📌 Вектор попитів (demand):

B_0: 2
B_1: 2
B_2: 2
B_3: 2
B_4: 2
B_5: 2
B_6: 2
B_7: 2
B_8: 2
B_9: 2


Тепер розв'яжемо згенеровану задачу з використанням методу Shortest Augmenting Path (SAP).

Для даної матриці не вийде вирішити задачу в повному формулюванні, адже деякі споживачі не будуть повністю задоволені, тому сформулюємо підграф, на якому розвязок існуватиме.

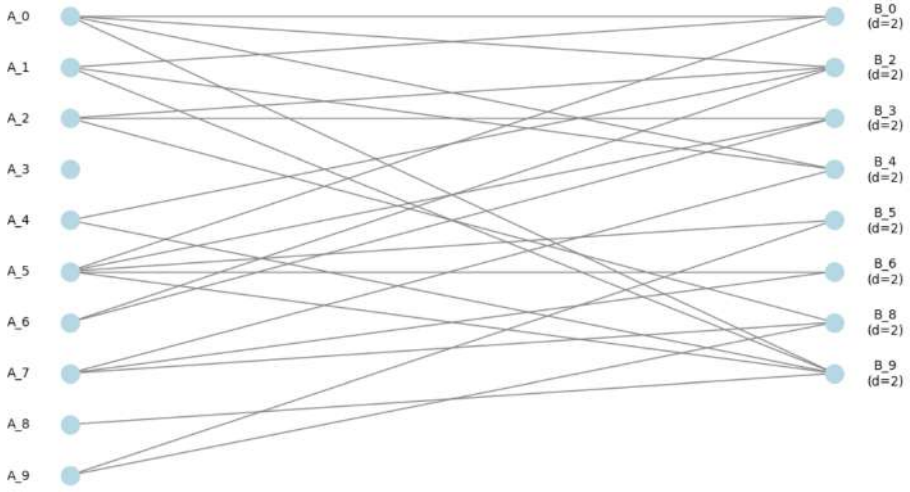
📌 Матриця з'єднаності (R):

| | B_0 | B_2 | B_3 | B_4 | B_5 | B_6 | B_8 | B_9 |
|---|---|---|---|---|---|---|---|---|
| A_0: | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| A_1: | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| A_2: | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| A_3: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A_4: | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| A_5: | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| A_6: | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| A_7: | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| A_8: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| A_9: | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

📌 Вектор попитів (demand):

B_0: 2
B_2: 2
B_3: 2
B_4: 2
B_5: 2
B_6: 2
B_8: 2
B_9: 2

Графове зображення матриця сполученості




Нижче наведено результати виконання обчислень для заданої конфігурації вхідних даних.

========== РЕЗУЛЬТАТ ==========

Розв'язність повної задачі: ❌
Знайдено підграф на якому можемо розв'язати: ✅

Мінімальне L: 2

Матриця постачань:

| | B_0 | B_2 | B_3 | B_4 | B_5 | B_6 | B_8 | B_9 |
|---|---|---|---|---|---|---|---|---|
| A_0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| A_1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| A_2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| A_3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A_4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| A_5 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| A_6 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| A_7 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| A_8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| A_9 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

Задоволений підграф споживачів: [0, 2, 3, 4, 5, 6, 8, 9]
Невдоволені споживачі: [1, 7]

⏱ Час виконання алгоритму: 0.8536 секунд

**Порівняння модифікованих методів при різних розмірах матриць**

З метою аналізу ефективності реалізованих алгоритмів було проведено експериментальне порівняння методів для різних масштабів задачі. Основна мета полягає у вивченні того, як змінюється час виконання, стабільність та поведінка кожного з методів при зростанні кількості виробників і споживачів.

Порівняння здійснюється для наступних підходів:

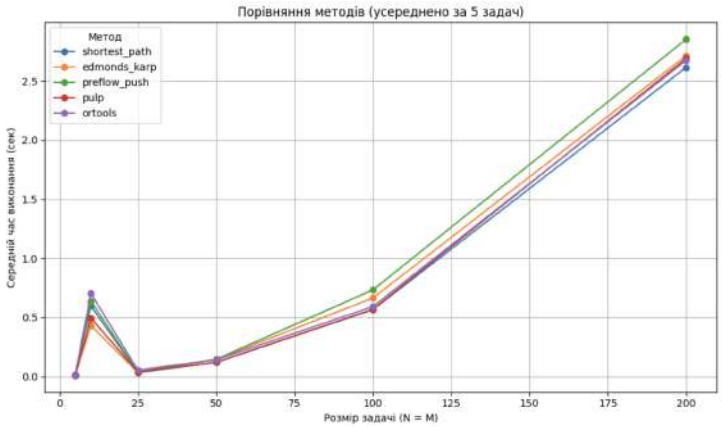
- Edmonds–Karp;
- Preflow Push;
- Shortest Augmenting Path (SAP);
- MILP (CBC через `PuLP`);
- MILP (SCIP через `OR-Tools`).

Щоб забезпечити об'єктивність оцінки, для кожної розмірності задачі ($N = M \in \{5, 10, 25, 50, 100, 200\}$) і для кожного методу було згенеровано по 5 випадкових прикладів. Результати усереднювались за кожною конфігурацією, що дозволяє зменшити вплив випадковості структури з'єднань або попиту на загальну оцінку продуктивності.

Далі буде наведено відповідні графіки та таблиці з порівняльними результатами.

Усереднені результати:

| | Розмір (N=M) | Метод | Середній час (сек) | Середнє L_min | Розв'язано повністю (%) |
|---|---|---|---|---|---|
| 0 | 5 | shortest_path | 0.0158 | 1.5 | 40% |
| 1 | 5 | edmonds_karp | 0.0113 | 1.5 | 40% |
| 2 | 5 | preflow_push | 0.0167 | 1.5 | 40% |
| 3 | 5 | pulp | 0.0129 | 1.5 | 40% |
| 4 | 5 | ortools | 0.0131 | 1.5 | 40% |
| 5 | 10 | shortest_path | 0.5964 | 2.4 | 40% |
| 6 | 10 | edmonds_karp | 0.4292 | 2.4 | 40% |
| 7 | 10 | preflow_push | 0.6373 | 2.4 | 40% |
| 8 | 10 | pulp | 0.4949 | 2.4 | 40% |
| 9 | 10 | ortools | 0.7021 | 2.4 | 40% |
| 10 | 25 | shortest_path | 0.0344 | 2.2 | 100% |
| 11 | 25 | edmonds_karp | 0.0365 | 2.2 | 100% |
| 12 | 25 | preflow_push | 0.0385 | 2.2 | 100% |
| 13 | 25 | pulp | 0.0373 | 2.2 | 100% |
| 14 | 25 | ortools | 0.0554 | 2.2 | 100% |
| 15 | 50 | shortest_path | 0.1199 | 2.2 | 100% |
| 16 | 50 | edmonds_karp | 0.1443 | 2.2 | 100% |
| 17 | 50 | preflow_push | 0.1456 | 2.2 | 100% |
| 18 | 50 | pulp | 0.1200 | 2.2 | 100% |
| 19 | 50 | ortools | 0.1415 | 2.2 | 100% |
| 20 | 100 | shortest_path | 0.5653 | 2.0 | 100% |
| 21 | 100 | edmonds_karp | 0.6644 | 2.0 | 100% |
| 22 | 100 | preflow_push | 0.7326 | 2.0 | 100% |
| 23 | 100 | pulp | 0.5686 | 2.0 | 100% |
| 24 | 100 | ortools | 0.5912 | 2.0 | 100% |
| 25 | 200 | shortest_path | 2.6154 | 2.0 | 100% |
| 26 | 200 | edmonds_karp | 2.7131 | 2.0 | 100% |
| 27 | 200 | preflow_push | 2.6530 | 2.0 | 100% |
| 28 | 200 | pulp | 2.6907 | 2.0 | 100% |
| 29 | 200 | ortools | 2.6745 | 2.0 | 100% |




Бачимо, що навіть при невеликих розмірах, час пошкуку розв'язку суттєво збільшується у випадках коли неможливо знайти необхідний розв'язок, що бачимо при розмірності 10, коли лише 4/10 прикладів були розв'язані повністю. Також цікаво, що при збільшенні розмірності не зменшується відсоток розмірів, що свідичить про те, що при заданій потребі, збільшення кількості виробників фактично збільшує ймовірність того, що хоча б якийсь виробник зможе задовльнити конкретного споживача. Насправді це вишлядає, як те, що потребує додаткового дослідження.

На наявних прикладах при великій розмірності краще себе проявив модифіковай метод Preflow Push тоді як на маленьких розмірах матриці сполученості найкразе себе показав OR-tool

# 4 Висновки

В результаті даної роботи вдалося розробити та імплементувати алгоритм для пошуку верхньої межі вихідної валентності з метою урівняння конкуренції в транспортній задачі з додатковими обмеженнями у вигляді метриці допустимості, а також лімітованої пропускної здатності кожного зі сполучень між виробником та споживачем. Розглянутий набір додаткових обмежень не був дослідженим раніше, тому сама розробка алгоритму представляє цінність. В процесі дослідження вдалося реалізувати 5 варіацій алгоритму, які були сформовані використанням різних допоміжних алгоритмів на одному з кроків загального алгоритму. В подальших дослідженнях бачу сенс сконцентруватися на дослідженні випадків, коли ми маємо більшу кількість споживачів ніж було розглянуто в прикладах вище, а саме по 1000, 10000, 100000 споживачів та виробників. Очікую що в таких випадках, буде доцільно застосувати більш комплексні алгоритми для забезпечення ефективного часу виконання алгоритму та його обчислювальної складності. Враховуючи отримані результати, вважаю за доцільне продовжити дослідження для випадку високої розрідженості матриці допустимості.

# Література


[1] Kantorovich L. V. Mathematical Methods of Organizing and Planning Production // Management Science. – Vol. 6, No. 4. – P. 366–422. – 1960.

[2] Hitchcock F. L. The Distribution of a Product from Several Sources to Numerous Localities // Journal of Mathematics and Physics. – Vol. 20, No. 1–4. – P. 224–230. – 1941.

[3] Taha H. A. Operations Research: An Introduction. – 8th ed. – Englewood Cliffs: Prentice Hall. – 1997.

[4] Klingman D., Russell R. The Transportation Problem with Mixed Constraints // Operational Research Quarterly (1970–1977). – Vol. 25, No. 3. – P. 447. – 1974.

[5] Ringuest J. L., Rinks D. B. Interactive Solutions for the Linear Multiobjective Transportation Problem // European Journal of Operational Research. – Vol. 32, No. 1. – P. 96–106. – 1987.

[6] Lau B., Cole S. R., Gange S. J. Competing Risk Regression Models for Epidemiologic Data // American Journal of Epidemiology. – Vol. 170, No. 2. – P. 244–256. – 2009.

[7] Luenberger D. G., Ye Y. Transportation and Network Flow Problems / In: Linear and Nonlinear Programming. – New York: Springer. – P. 145–171. – 2008.

[8] Іксанов О., Шевченко В. Транспортна задача, її властивості та методи розв'язування. – [Монографія]. – 2010.

[9] Chauhan R., Bhat A. A., Sulaiman F. B. A., Tarray T. A. Recent Trends in Transportation Problem: A Review Paper // International Journal of Modern Mathematical Sciences. – Vol. 21, No. 1. – P. 20–43. – 2023. URL: `http://www.ModernScientificPress.com/Journals/ijmms.aspx`

[10] Priyadharshini T., Manikandan K. M. Application of Graph Theory to Find Optimal Path and Minimized Cost for the Transportation Problem // International Journal of Mathematics Trends and Technology. – Vol. 67, No. 10. – P. 61–69. – 2019.

[11] Bisen S. K. Application of Graph Theory in Transportation Networks // International Journal of Scientific Research and Management. – Vol. 5, No. 7. – 2017.

[12] Guze S. Graph Theory Approach to Transportation Systems Design and Optimization // TransNav: International Journal on Marine Navigation and Safety of Sea Transportation. – Vol. 8, No. 4. – P. 571–578. – 2014.

[13] Google for Developers. OR-Tools. – URL: https://developers.google.com/optimization

[14] NetworkX. NetworkX documentation. – URL: https://networkx.org/

# Додаток А

```python
class DataGenerator:

    def __init__(self, N, M, demand_range=(1, 5), connection_prob=0.5, seed=None, equal_demand=False):

        self.N = N
        self.M = M
        self.demand_range = demand_range
        self.connection_prob = connection_prob
        self.seed = seed
        self.equal_demand = equal_demand
        self.logger = logging.getLogger("DataGenerator")

        if seed is not None:
            np.random.seed(seed)
            self.logger.info(f"Seed встановлено: {seed}")

    def generate_connectivity_matrix(self):

        R = np.random.rand(self.N, self.M) < self.connection_prob
        R = R.astype(int)

        # Гарантуємо, що кожен споживач має хоча б одного постачальника
        for j in range(self.M):
            if R[:, j].sum() == 0:
                i = np.random.randint(0, self.N)
                R[i, j] = 1

        self.logger.info("Згенеровано матрицю сполученості R")
        return R

    def generate_demand_vector(self):
        low, high = self.demand_range

        if self.equal_demand:
            value = np.random.randint(low, high + 1)
            demand = np.full(self.M, value)
            self.logger.info(f"Згенеровано рівномірний попит: усі споживачі мають {value}")
        else:
            demand = np.random.randint(low, high + 1, size=self.M)
            self.logger.info("Згенеровано вектор попиту (різний для кожного споживача)")
        return demand

    def generate(self):
        R = self.generate_connectivity_matrix()
        demand = self.generate_demand_vector()
        return demand, R

```

```python
class ProblemVisualizer:
    def __init__(self, R, demand, K, consumer_indices=None):
        self.R = R
        self.demand = demand
        self.K = K
        self.N, self.M = R.shape
        self.consumer_indices = consumer_indices or list(range(self.M))  # глобальні індекси
        self.G = nx.Graph()
        self.logger = logging.getLogger("ProblemVisualizer")
        self.logger.info("Ініціалізовано модуль візуалізації")

    def build_graph(self):
        self.logger.info("Побудова графа з'єднань...")

        for i in range(self.N):
            self.G.add_node(f"A_{i}", bipartite=0)
        for j in range(self.M):
            global_j = self.consumer_indices[j]
            self.G.add_node(f"B_{global_j}\n(d={self.demand[j]})", bipartite=1)
        for i in range(self.N):
            for j in range(self.M):
                if self.R[i, j] == 1:
                    global_j = self.consumer_indices[j]
                    self.G.add_edge(f"A_{i}", f"B_{global_j}\n(d={self.demand[j]})")
        self.logger.info("Граф побудовано")

    def display_ascii_matrix(self):
        self.logger.info("Виведення матриці з'єднаності у вирівняній ASCII-таблиці")
        col_width = 5
        row_label_width = 5

        header = " " * row_label_width + "".join(f"{f'B_{self.consumer_indices[j]}':>{col_width}}" for j in range(self.M))
        print("📌 Матриця з'єднаності (R):")
        print(header)

        for i in range(self.N):
            row_label = f"A_{i}:".ljust(row_label_width)
            row_values = "".join(f"{self.R[i, j]:>{col_width}}" for j in range(self.M))
            print(row_label + row_values)

    def display_demand_and_K(self):
        self.logger.info("Виведення параметрів попиту та обмеження K")
        print("\n📌 Вектор попитів (demand):")
        for j in range(self.M):
            print(f"  B_{self.consumer_indices[j]}: {self.demand[j]}")
        print(f"\n📌 Обмеження K: {self.K} одиниць на одного споживача від одного виробника")

    def draw(self, figsize=(10, 6), node_size=200):
        self.logger.info("Початок візуалізації задачі")

        self.display_ascii_matrix()
        self.display_demand_and_K()
        self.build_graph()

        left_nodes = [n for n in self.G.nodes if n.startswith("A_")]
        right_nodes = [n for n in self.G.nodes if n.startswith("B_")]

        # Основні координати вузлів
        pos = {}
        pos.update((node, (0, -i)) for i, node in enumerate(left_nodes))
        pos.update((node, (3, -i)) for i, node in enumerate(right_nodes))

        # Позиції підписів вузлів з зсувом
        label_pos = {}
        for i, node in enumerate(left_nodes):
            label_pos[node] = (-0.2, -i)
        for i, node in enumerate(right_nodes):
            label_pos[node] = (3.2, -i)

        plt.figure(figsize=figsize)
        nx.draw(self.G, pos, with_labels=False, node_size=node_size,
                node_color="lightblue", edge_color="gray")

        nx.draw_networkx_labels(self.G, label_pos, font_size=10)

        plt.title("Графове зображення матриця сполученості")
        plt.axis("off")
        plt.show()
        self.logger.info("Візуалізація завершена")
```

```python
class TransportSolver:

    def __init__(self, demand_vector, connectivity_matrix, max_single_supply):
        self.b = np.array(demand_vector)
        self.R = np.array(connectivity_matrix)
        self.K = max_single_supply
        self.N, self.M = self.R.shape
        self.logger = logging.getLogger("TransportSolver")
        self.logger.info(f"Ініціалізовано задачу з {self.N} виробниками та {self.M} споживачами")

    def build_graph(self, L, b=None, R=None):
        b = b if b is not None else self.b
        R = R if R is not None else self.R
        G = nx.DiGraph()
        G.add_node("s")
        G.add_node("t")

        for i in range(self.N):
            G.add_edge("s", f"A_{i}", capacity=L)

        for j in range(len(b)):
            G.add_edge(f"B_{j}", "t", capacity=b[j])

        for i in range(self.N):
            for j in range(len(b)):
                if R[i, j] == 1:
                    G.add_edge(f"A_{i}", f"B_{j}", capacity=self.K)

        return G

    def is_feasible(self, L, b=None, R=None):
        b = b if b is not None else self.b
        R = R if R is not None else self.R
        G = self.build_graph(L, b, R)
        flow_value, _ = nx.maximum_flow(G, "s", "t")
        feasible = flow_value == b.sum()
        self.logger.debug(f"Перевірка при L={L}: {'так' if feasible else 'ні'} (flow={flow_value}, demand={b.sum()})")
        return feasible

    def find_min_L(self):
        total_demand = self.b.sum()
        left = (total_demand + self.N - 1) // self.N
        right = total_demand
        best_L = right
        self.logger.info("Початок пошуку мінімального L...")

        while left <= right:
            mid = (left + right) // 2
            if self.is_feasible(mid):
                best_L = mid
                right = mid - 1
            else:
                left = mid + 1

        self.logger.info(f"Знайдено мінімальне L: {best_L}")
        return best_L

    def find_min_L_for_subgraph(self, b_sub, R_sub):
        total_demand = b_sub.sum()
        left = (total_demand + self.N - 1) // self.N
        right = total_demand
        best_L = right
        self.logger.info("Пошук мінімального L для підграфа...")

        while left <= right:
            mid = (left + right) // 2
            if self.is_feasible(mid, b=b_sub, R=R_sub):
                best_L = mid
                right = mid - 1
            else:
                left = mid + 1

        self.logger.info(f"Мінімальне L для підграфа: {best_L}")
        return best_L
```

```python
    def find_largest_feasible_subgraph(self, L, flow_func=None):
        best_subset = []
        best_size = 0

        for r in range(1, self.M + 1):
            for subset in combinations(range(self.M), r):
                sub_b = self.b[list(subset)]
                sub_R = self.R[:, list(subset)]
                G = self.build_graph(L, sub_b, sub_R)
                flow_value, _ = nx.maximum_flow(G, "s", "t", flow_func=flow_func)
                if flow_value == sub_b.sum() and r > best_size:
                    best_subset = list(subset)
                    best_size = r

        return best_subset

    def solve_with(self, method="shortest_path"):
        self.logger.info(f"=== Старт розв'язання задачі (метод: {method}) ===")
        start_time = time.time()

        flow_func = {
            "edmonds_karp": nx.algorithms.flow.edmonds_karp,
            "preflow_push": nx.algorithms.flow.preflow_push,
            "shortest_path": nx.algorithms.flow.shortest_augmenting_path
        }.get(method, nx.algorithms.flow.shortest_augmenting_path)

        def is_feasible_with_method(L, b=None, R=None):
            b = b if b is not None else self.b
            R = R if R is not None else self.R
            G = self.build_graph(L, b, R)
            flow_value, _ = nx.maximum_flow(G, "s", "t", flow_func=flow_func)
            return flow_value == b.sum()

        if is_feasible_with_method(self.b.sum()):
            L_min = self.find_min_L()
            matrix = self.get_supply_matrix(L_min, flow_func=flow_func)
            exec_time = time.time() - start_time
            return {
                "method": method,
                "full_problem_solvable": True,
                "subgraph_used": False,
                "L_min": L_min,
                "supply_matrix": matrix,
                "feasible_subgraph": list(range(self.M)),
                "unsatisfied_consumers": [],
                "execution_time": exec_time
            }

        sub_indices = self.find_largest_feasible_subgraph(self.b.sum(), flow_func=flow_func)
        if not sub_indices:
            exec_time = time.time() - start_time
            return {
                "method": method,
                "full_problem_solvable": False,
                "subgraph_used": False,
                "L_min": None,
                "supply_matrix": None,
                "feasible_subgraph": [],
                "unsatisfied_consumers": list(range(self.M)),
                "execution_time": exec_time
            }

        sub_b = self.b[sub_indices]
        sub_R = self.R[:, sub_indices]
        L_min = self.find_min_L_for_subgraph(sub_b, sub_R)
        matrix = self.get_supply_matrix(L_min, sub_b, sub_R, consumer_indices=sub_indices, flow_func=flow_func)
        exec_time = time.time() - start_time

        return {
            "method": method,
            "full_problem_solvable": False,
            "subgraph_used": True,
            "L_min": L_min,
            "supply_matrix": matrix,
            "feasible_subgraph": sub_indices,
            "unsatisfied_consumers": list(set(range(self.M)) - set(sub_indices)),
            "execution_time": exec_time
        }

    def solve(self):
        return self.solve_with(method="shortest_path")
```