НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра математичного моделювання та аналізу даних

«До захисту допущено»
В.о. завідувача кафедри
_____ Іван ТЕРЕЩЕНКО
«___» _____ 2025 р.

Дипломна робота

на здобуття ступеня бакалавра

зі спеціальності: 113 Прикладна математика
на тему: **«Оптимальне траснспортування на графах і задача Бекмана»**

Виконав: студент 4 курсу, групи ФІ-12
Бобров Андрій Олексійович _____

Керівник: ст. досл., к.т.н., доц.
Хайдуров Владислав Володимирович _____

Консультант: канд. фіз.-мат. н., снс.
Рябов Георгій Валентинович _____

Рецензент: доц., канд. фіз.-мат. н.,
доцент кафедри ММЗІ
Ніщенко Ірина Іванівна _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Київ — 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра математичного моделювання та аналізу даних

Рівень вищої освіти — перший (бакалаврський)
Спеціальність (освітня програма) — 113 Прикладна математика,
ОПП «Математичні методи моделювання, розпізнавання образів та комп'ютерного зору»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Іван ТЕРЕЩЕНКО
«__» _____ 2025 р.

ЗАВДАННЯ
на дипломну роботу

Студент: Бобров Андрій Олексійович

1. Тема роботи: «*Оптимальне трасnспортування на графах і задача Бекмана*»,

керівник: ст. досл., к.т.н., доц. Хайдуров Владислав Володимирович,

затверджені наказом по університету №1761-с від «26» 05 2025 р.

2. Термін подання студентом роботи: «__» _____ 2025 р.

3. Вихідні дані до роботи: послідовність кроків алгоритму побудови оптимального транспортування на графах.

4. Зміст роботи:

1) Провести огляд джерел за тематикою дослідження;

2) Довести еквівалентність задач на графі;

3) Перевірити результати експериментально.

5. Перелік ілюстративного матеріалу: презентація доповіді

6. Дата видачі завдання: 10 вересня 2024 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання	Примітка
1	Узгодження теми роботи із науковим керівником	01-15 вересня 2024 р.	Виконано
2	Огляд опублікованих джерел за тематикою оптимального транспортування	Вересень-жовтень 2024 р.	Виконано
3	Огляд опублікованих джерел за тематикою задачі Бекмана і її аналогів	Жовтень-грудень 2024 р.	Виконано
4	Отримання узагальнень на випадок не орієнтованих невід'ємно зважених графів	Січень-лютий 2025 р.	Виконано
5	Доведення еквівалентності задач і побудова процедури перетворення потоку на траспортний план	Лютий-Березень 2025 р.	Виконано
6	Розробка програмного забезпечення мовою Python	Квітень 2025 р.	Виконано
7	Оформлення дипломної роботи	Травень 2025 р.	Виконано

Студент _____ Бобров А.О.

Керівник _____ Хайдуров В. В.

РЕФЕРАТ

Кваліфікаційна робота містить: 44 сторінки, 2 рисунки, 2 таблиці, 9 джерел.

У цій роботі було досліджено зв'язок задачі мінімального потоку (задачі Бекмана) із задачею оптимального транспортування для випадку дискретного простору (зв'язного невід'ємно зваженого графа).

В ході дослідження було доведено еквівалентність задачі мінімального потоку та задачі оптимального транспортування на графі. Було запропоновано алгоритм побудови оптимального транспортного плану із оптимального потоку та доведено коректність алгоритму. Було наведено реалізацію алгоритму мовою Python. Також, було проведено порівняльний аналіз результатів роботи алгоритму з іншими методами побудови оптимального транспортного плану.

ГРАФ, ПОТІК НА ГРАФІ, ОПТИМАЛЬНИЙ ТРАНСПОРТНИЙ ПЛАН

ABSTRACT

The qualification work contains: 44 pages, 2 figures, 2 tables, and 9 citations.

This work investigates the relationship between the minimum flow problem (the Beckmann problem) and the optimal transport problem in the case of a discrete space (a connected non-negatively weighted graph).

In the course of the research, the equivalence of the minimum flow problem and the optimal transport problem on a graph was proven. An algorithm for constructing an optimal transport plan from an optimal flow was proposed, and the correctness of the algorithm was proven. An implementation of the algorithm in Python was presented. A comparative analysis of the algorithm's results with other methods of constructing the optimal transport plan was also conducted.

GRAPH, FLOW ON A GRAPH, OPTIMAL TRANSPORTATION PLAN

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

$\mathbb{R}_+$	—	множина невід'ємних дійсних чисел.
δ_A	—	міра Дірака множини A.
C_{uv}	—	шлях мінімальної ваги між вершинами u, v.
$\mathbf{n}$	—	вектор зовнішньої нормалі.
∂A	—	межа множини A.
dH^n	—	n-вимірна міра Хаусдорфа.
$d\lambda^n$	—	n-вимірна міра Лебега.
$\mu[A]$	—	значення міри μ на множині A.
$\Pi(\mu, \nu)$	—	множина каплінгів мір μ та ν.
$\otimes$	—	добуток Кронекера.
$L^1(\mu)$	—	множина абсолютно інтегровних функцій відносно міри μ.
$\|\varphi\|_{\mathrm{Lip}}$	—	скорочене позначення сталої Ліпшиця
φ^c	—	c-перетворення функції $\varphi : X \to \mathbb{R}$
φ^{cc}	—	cc-перетворення функції $\varphi : X \to \mathbb{R}$

ВСТУП

Актуальність дослідження. Актуальність даного дослідження зумовлена широким використанням графа для моделювання зв'язків між містами і побудови транспортних маршрутів в логістиці. В загальному випадку побудова оптимальних транспортних маршрутів шляхом розв'язання лінійних задач не є оптимальною за часом. Мотивацією для запропонованого в цій роботі підходу є те, що побудова оптимального потоку є доволі швидкою процедурою і вимагає менше пам'яті.

Метою дослідження є побудова і реалізація алгоритму перетворення потоку на графі на план транспортування. Для досягнення мети необхідно вирішити такі завдання:

1) провести огляд опублікованих джерел за тематикою дослідження;

2) довести еквівалентність поставлених задач;

3) отримати процедуру перетворення оптимального потоку на оптимальний транспортний план;

4) перевірити одержані результати експериментально.

Об'єктом дослідження є логістично-економічні процеси.

Предметом дослідження є математичні методи, графові моделі і алгоритми дослідження логістично-економічних процесів.

При розв'язанні поставлених задач використовувались такі *методи дослідження*: методи дискретної математики (теорія графів), функціонального аналізу (теорії міри) та комп'ютерного моделювання (проведення обчислювальних експериментів).

Практичне значення полягає в прискоренні побудови оптимального транспортного плану на графі.

1 ЗАДАЧА ОПТИМАЛЬНОГО ТРАНСПОРТУВАННЯ ТА ЗАДАЧА ПОШУКУ ОПТИМАЛЬНОГО ПОТОКУ

В першому розділі розглянуто поняття задачі оптимального транспортування та поняття задачі оптимального потоку для неперервного випадку. Розглянуто два підходи до формулювання задачі оптимального транспортування, і зв'язок між ними.

1.1 Задача оптимального транспортування

Сформулюємо основні поняття теорії оптимального транспортування.

Означення 1.1 (Образ міри при відображенні)**.** Нехай задана міра μ на вимірному просторі X і вимірне відображення $T : X \to Y$. **Образом міри μ при відображені T** називається міра ν на Y, для якої справедливо:

$$\forall A \subset Y, A \text{ – вимірна} : \nu[A] = \mu\left[T^{-1}(A)\right].$$

Позначають: $\nu = T\#\mu$.

Означення 1.2. Нехай дано міру μ на множині X і міру ν на множині Y. **Каплінгом** мір μ та ν називається міра π на $X \times Y$ така, що для довільних вимірних множин $A \subset X$ та $B \subset Y$ виконується

$$\pi[A \times Y] = \mu[A]; \tag{1.1}$$

$$\pi[X \times B] = \nu[B]. \tag{1.2}$$

Множину усіх каплінгів мір μ та ν будемо позначати $\Pi(\mu, \nu)$.

1.1.1 Задача Монжа

Гаспар Монж у своїй роботі 1781 року [6] сформулював задачу оптимального перенесення купи ґрунту X в яму Y того ж об'єму. Перенесення визначається за допомогою деякої функції $T : X \to Y$, що не формально позначає в яку точку ями $y \in Y$ треба перевезти ґрунт з точки $x \in X$.

Формулювання задачі Монжа має наступний вигляд:

Означення 1.3 (Задача Монжа)**.** Дано розподіли μ на X та ν на Y і деяка функція ціни $c : X \times Y \to \mathbb{R}_+ \cup \{+\infty\}$, яка визначає вартість перевезення ґрунту з точки $x \in X$ в точку $y \in Y$. Необхідно знайти функцію $T^* : X \to Y$ таку, що $T^*\#\mu = \nu$ і функціонал

$$\int_X c\left(x, T^*(x)\right)\, d\mu(x),$$

досягає мінімуму.

Основною проблемою такого формулювання є той факт, що маса деякої точки $x \in X$ не може бути розділена між двома місцями в ямі $y_1, y_2 \in Y$.

1.1.2 Задача Канторовича

У 1942 Леонід Канторович сформулював задачу транспортування товару до споживачів [4].

У формалізації Канторовича ми маємо дві множини X та Y, що відповідно є множинами «товару» та «споживачів»; щільності «виробництва товару» і «споживання» відповідно рівні μ та ν. Задача полягає в пошуку оптимального плане перевезення $\pi \in \Pi(\mu, \nu)$ усього «товару» до «споживачів», за умови, що ціна перевезення одиниці товару з точки $x \in X$ до споживача у точці $y \in Y$ рівна $c(x, y)$.

Означення 1.4 (Задача Канторовича)**.** Дано розподіли μ на X та ν на Y і деяка функція ціни $c : X \times Y \to \mathbb{R}_+ \cup \{+\infty\}$. Необхідно знайти міру $\pi \in \Pi(\mu, \nu)$, яка мінімізує функціонал

$$\int_{X\times Y} c(x, y)\, d\pi.$$

На сьогоднішній день формулювання вище має назву *задачі Монжа-Канторовича.*

1.1.3 Дуальність

Задача Канторовича має дуальне формулювання.

Визначимо множину:

$$\Phi_c = \{(\varphi, \psi) \in L^1(\mu) \times L^1(\nu) : \forall (x, y) \in X \times Y : \varphi(x) + \psi(y) \leq c(x, y)\},$$

і визначимо функціонал $J(\varphi, \psi)$ як:

$$J(\varphi, \psi) = \int_X \varphi\, d\mu + \int_Y \psi\, d\nu.$$

Тоді справедлива наступна теорема.

Теорема 1.1 (Дуальність Канторовича [9])**.** *Нехай X та Y — дві компактні множини з мірами μ та ν, відповідно. Нехай $c : X \times Y \to \mathbb{R}_+ \cup \{+\infty\}$ — напівнеперервна знизу функція ціни. Тоді:*

$$\inf_{\pi\in\Pi(\mu,\nu)} \int_{X\times Y} c(x, y)\, d\pi(x, y) = \sup_{(\varphi,\psi)\in\Phi_c} J(\varphi, \psi).$$

Ця теорема є фундаментом для подальшого дослідження як методів розв'язку задачі Канторовича, так і зв'язку з іншими задачами оптимізації.

Зокрема, важливим наслідком теореми 1.1 є наступна теорема.

Теорема 1.2 (Дуальність Канторовича-Рубінштейна [9])**.** *Нехай (X, ρ) — компактний метричний простір і задані дві міри μ та ν на X.*

Для функції ціни $c(x,y) = \rho(x,y)$ *виконується:*

$$\begin{aligned} \inf_{\pi\in\Pi(\mu,\nu)} \int_{X\times Y} \rho(x,y)\, d\pi(x,y) = \\ = \sup_{\varphi\in L^1(\mu-\nu)} \left\{ \int_X \varphi\, d(\mu-\nu) : \|\varphi\|_{Lip} \le 1 \right\}. \end{aligned} \tag{1.3}$$

1.2 Задача пошуку мінімального потоку

Нехай дано деяку достатньо гладенька і зв'язну область $\Omega \subset \mathbb{R}^2$, яку можна інтерпретувати як деяке «місто». На множині Ω визначені дві ймовірнісні міри μ, ν — відповідно міра локального попиту та локальної пропозиції. В цих позначеннях можна вважати, що міра $(\mu-\nu)$ є локальною мірою надлишкового попиту.

Будемо вважати, що трафік товару моделюється як деяке векторне поле $\mathbf{Y} : \Omega \to \mathbb{R}^2$. Таким чином напрям $\mathbf{Y}(x)$ визначає напрям руху товару в точці $x \in \Omega$, а $|\mathbf{Y}(x)|$ визначає інтенсивність цього руху.

Природньо вважати, що виконується аналог закону збереження маси: кількісно витік споживачів із довільної області $K \subset \Omega$ рівний надлишковому попиту в цій області:

$$\int_{\partial K} \mathbf{Y}\cdot\mathbf{n}\, dH = (\mu-\nu)(K).$$

Локально це еквівалентно рівності в слабкому сенсі:

$$\operatorname{div}(\mathbf{Y}) = (\mu-\nu), \tag{1.4}$$

тобто

$$\int_{\partial K} \mathbf{Y}\cdot\mathbf{n}\, dH = \iint_K \operatorname{div}(Y)\, d\lambda^2 = \iint_K d(\mu-\nu)$$

Також будемо вважати наше місто ізольованим:

$$\mathbf{Y}\cdot\mathbf{n} = 0 \text{ на } \partial\Omega. \tag{1.5}$$

Означення 1.5 (Задача пошуку мінімального потоку [1; 3])**.** Нехай дано дві міри μ та ν на множині Ω і деяка неспадна функція $g : \mathbb{R}_+ \to \mathbb{R}_+ \cup \{+\infty\}$. Необхідно знайти таке векторне поле $\mathbf{Y}^*$, що задовольняє умовам (1.4)–(1.5) і мінімізує функціонал:

$$\int_\Omega g\left(|\mathbf{Y}(\omega)|\right)\, d\lambda^2. \tag{1.6}$$

Векторне поле $\mathbf{Y}^*$, на якому досягається мінімум (1.6), називають **мінімальним потоком**.

В цій дещо спрощеній моделі ми вважаємо, що ціна транспортування одиниці товару не залежить від інтенсивності (нема заторів). Також у якості функції g оберемо функцію $g(t) = t$ для $t \in \mathbb{R}_+$; в багатьох випадках це доволі природній вибір.

Тоді задача пошуку мінімального потоку приймає вигляд

$$\int_\Omega |\mathbf{Y}(x)|\, d\lambda^2 \to \min\,.$$

Неформально ми шукаємо такий потік, сумарна інтенсивність якого є мінімальною.

1.3 Зв'язок задачі Канторовича з пошуком мінімального потоку

Задача пошуку мінімального потоку 1.5 полягає у знаходженні шляхів, якими треба транспортувати товар. З іншого боку, задача Канторовича 1.4 полягає в знаходженні способів перевезення одиниці товару до споживача.

Природним чином постає питання про зв'язок цих задач.

Теорема 1.3 (Еквівалентність задачі пошуку мінімального потоку і задачі Монжа-Канторовича [8])**.** *Нехай μ, ν – дві ймовірнісні міри на Ω,*

$\rho(x,y) = |x-y|_2$. *Тоді:*

$$\inf_{\pi\in\Pi(\mu,\nu)} \int_{\Omega^2} \rho(x,y)\, d\pi(x,y) = \inf\left\{ \int_{\Omega} |\mathbf{Y}(x)|\ d\lambda^2 : \mathbf{Y} \text{ } \textit{задовольняє} \text{ } (1.4)-(1.5) \right\}.$$

Отже, постає питання про те, яким чином ми можемо отримати оптимальний план транспортування для задачі Канторовича 1.4, використовуючи деякий мінімальний потік, і навпаки.

Висновки до розділу 1

В цьому розділу було розглянуто основні означення теорії оптимального транспортування і визначення задачі мінімального потоку.

Було розглянуто випадок, коли ціна в задачі оптимального транспортування задається евклідовою метрикою. В цьому випадку можна побудувати еквівалентність між задачами оптимального транспортування та мінімального потоку.

Наступний розділ буде присвячено дослідженню еквівалентності цих задач у випадку графа і метрики, індукованої вагами цього графа.

2 ОПТИМАЛЬНЕ ТРАНСПОРТУВАННЯ І МІНІМАЛЬНИЙ ПОТІК НА ГРАФІ

В цьому розділі будуть сформульовані аналоги задачі Канторовича (1.4) та задачі пошуку мінімального потоку (1.5) для графа.

Буде доведено еквівалентність цих задач. Також буде одержано процедуру побудови оптимального плану транспортування з мінімального потоку.

2.1 Задача Канторовича на скінченній множині

Спочатку наведемо визначення каплінгу дискретних мір. В межах цієї роботи будемо вважати, що міра на множині з $n < \infty$ елементів задається вектором з невід'ємними компонентами.

Означення 2.1 (Каплінг дискретних мір)**.** Нехай дано дві дискретні міри $\mathbf{a} \in \mathbb{R}_+^n$ та $\mathbf{b} \in \mathbb{R}_+^m$. Через $\mathbb{1}_n$ позначимо вектор-стовпчик одиниць.

Каплінгом двох дискретних мір називається матриця $\mathbf{P} \in \mathbb{R}_+^{n \times m}$ така, що

$$\mathbf{P}\mathbb{1}_m = \mathbf{a};$$
$$\mathbf{P}^\top \mathbb{1}_n = \mathbf{b}$$

Множину усіх каплінгів двох дискретних мір $\mathbf{a}, \mathbf{b}$ будемо позначати $\mathbf{U}(\mathbf{a}, \mathbf{b})$.

Тепер наведемо означення задачі Канторовича для випадоку двох скінченних множин.

Означення 2.2 (Лінійна програма Канторовича [7])**.** Нехай дано дві скінченні множини X та Y, причому $|X| = n, |Y| = m$. Також нехай дано дві дискретні міри $\mathbf{a} \in \mathbb{R}_+^n$ та $\mathbf{b} \in \mathbb{R}_+^m$ і матриця цін $\mathbf{C} \in \mathbb{R}_+^{n \times m}$.

Необхідно зайти матрицю $\mathbf{P}^* \in \mathbf{U}(\mathbf{a}, \mathbf{b})$ таку, що

$$\sum_{i,j} \mathbf{P}_{ij} \cdot \mathbf{C}_{ij} \to \min .$$

Означення вище є частковим випадком (1.4), коли обидві міри μ та ν є сумами мір Дірака.

Таким чином для означення вище також справедливі аналоги дуальних теорем (1.1–1.2).

2.2 Задача пошуку мінімального потоку на графі

Надалі ми будемо працювати із зваженими графами $G = (V, E, w)$, де V – скінченна множина вершин графа, $E \subset V \times V$ – множина неорієнтованих ребер графа, тобто $(u, v) \in E \iff (v, u) \in E$, $w : E \to \mathbb{R}_+ \cup \{+\infty\}$ – функція ваги ребра; без втрати загальності вважаємо вершини графу V пронумерованими від 1 до $n := |V|$.

Визначимо потік на графі.

Означення 2.3 (Потік на графі)**. Потоком на графі** $G = (V, E, w)$ будемо називати довільне відображення $s : E \to \mathbb{R}_+$.

Без втрати загальності можемо вважати, що довільна функція $f : V \to \mathbb{R}$ є деяким вектором $\mathbf{f} \in \mathbb{R}^{|V|}$, для якого виконується $\forall v \in V : f(v) = \mathbf{f}_v$.

На відміну від неперервного випадку, в якому $\mathbf{f}$ є векторним полем, потік на графі має тільки «інтенсивність»; напрям руху визначається ребром, що є аргументом потоку.

2.2.1 Диференціальні оператори на графі

Сформулюємо аналоги диференціальних операторів для графа. Такий підхід дозволить природним шляхом узагальнити поняття задачі

мінімального потоку на випадок графу.

Означення 2.4 (Оператор градієнта)**.** Для довільної функції вершини графа $\mathbf{f} \in \mathbb{R}^{|V|}$ визначимо оператор градієнта $\nabla : \mathbb{R}^{|V|} \to \mathbb{R}^{|E|}$ як:

$$\forall (i,j) \in E : (\nabla \mathbf{f})_{ij} := \mathbf{f}_i - \mathbf{f}_j.$$

Означення 2.5 (Оператор дивергенції)**.** Для довільного потоку $\mathbf{s} \in \mathbb{R}^{|E|}$ визначимо оператор дивергенції $\operatorname{div} : \mathbb{R}^{|E|} \to \mathbb{R}^{|V|}$ як:

$$\forall i \in V : \operatorname{div}(s)_i := \sum_{j:(i,j)\in E} \mathbf{s}_{ij} - \mathbf{s}_{ji}.$$

Наступне твердження доводить зв'язок між визначеннями вище.

Твердження 2.1 (Коректність визначення)**.** *Нехай дано граф $G = (V, E, w)$ і дві функції $f : V \to \mathbb{R}$ та $g : E \to \mathbb{R}$. Для довільних двох функцій $f^1, f^2 : I \to \mathbb{R}$ визначимо скалярне множення $\langle f^1, f^2 \rangle := \sum_{i\in I} f_i^1 \cdot f_i^2$.*

Тоді справедлива рівність:

$$\langle \nabla f, g \rangle = \langle f, \operatorname{div}(g) \rangle.$$

Доведення. Розкриємо скалярний добуток $\langle \nabla f, g \rangle$:

$$\begin{aligned}\langle \nabla f, g \rangle &= \sum_{e\in E} (\nabla f)_e \cdot g_e = \sum_{v\in V} \sum_{u:(v,u)\in E} (\nabla f)_{uv} \cdot g_{uv} = \\ &= \sum_{v\in V} \sum_{u:(v,u)\in E} (f_u - f_v) \cdot g_{uv} = \sum_{v\in V} \sum_{u:(v,u)\in E} f_u \cdot g_{uv} - \\ &- \sum_{v\in V} \sum_{u:(v,u)\in E} f_v \cdot g_{uv}.\end{aligned}$$

Тепер змінимо порядок підсумовування першої суми. Отримаємо:

$$\sum_{v\in V}\sum_{u:(v,u)\in E} f_v \cdot g_{vu} - \sum_{v\in V}\sum_{u:(v,u)\in E} f_v \cdot g_{uv} =$$
$$= \sum_{v\in V} f_v \sum_{u:(v,u)\in E} (g_{vu} - g_{uv}) = \sum_{v\in V} f_v \cdot \operatorname{div}(g) = \langle f, \operatorname{div}(g)\rangle$$

□

Таким чином, наведені вище означення мають властивості, притаманну неперервним операторам градієнта і дивергенції: оператор дивергенції є спряженим оператором до оператора градієнта відносно рівномірного розподілу на графі.

2.2.2 Потік на графі

Розглянемо зв'язний граф $G = (V, E, w)$, який моделює деяке місто. Природньо вважати, що ваги ребер моделюють відстань між районами міста, зокрема $\forall e \in E : w(e) \geq 0$.

На множині вершин визначимо дискретні міри $\mathbf{a}$ та $\mathbf{b}$, що, аналогічно до секції 1.2, є відповідно мірами локального попиту та пропозиції.

Визначимо аналог закону збереження маси (1.4): для довільної вершини $v \in V$:

$$\operatorname{div}(\mathbf{s})_v = \mathbf{a}_v - \mathbf{b}_v. \tag{2.1}$$

Оператор дивергенції 2.5 визначає кількість «товару», що виходить з вершини. Таким чином (2.1) накладає наступну умову на потік: кількість трафіку з вершини має бути рівною надлишковому попиту в цій вершині.

Тепер визначимо задачу пошуку мінімального потоку на графі.

Означення 2.6 (Задача пошуку мінімального потоку на графі)**.** Нехай дано неорієнтований зв'язний граф з невід'ємними вагами

$G = (V, E, w)$. Необхідно знайти такий потік $s : E \to \mathbb{R}_+$, що функціонал

$$\sum_{e \in E} \mathbf{s}_e \cdot w_e, \tag{2.2}$$

досягає мінімуму і виконується умова збереження маси (2.1).

Потік, що мініміхує функціонал (2.2) будемо називати **мінімальним потоком на графі**.

Таке формулювання співпадає з неперервним аналогом задачі мінімального потоку 1.5: ми шукаємо потік, який мінімізує сумарну інтенсивність.

2.3 Еквівалентність задачі оптимального транспортування і задачі пошуку мінімального потоку для графа

В цій секції буде доведено основний результат роботи.

Теорема 2.1 (Еквівалентність задачі оптимального транспортування і мінімального потоку для графа)**.** *Нехай дано зв'язний невід'ємно зважений граф* $G = (V, E, w)$ *і дві міри* $\mathbf{a}, \mathbf{b}$*. Визначимо матрицю цін* $\mathbf{C}$ *наступним чином*

$$\mathbf{C}_{ij} = \textit{мінімальна вага шляху між вершинами } i, j$$

, де вага шляху є сумою ваг усіх ребер цього шляху.

Тоді виконується рівність

$$\min_{\mathbf{P} \in \mathbf{U}(\mathbf{a},\mathbf{b})} \sum_{i,j} \mathbf{P}_{ij} \cdot \mathbf{C}_{ij} = \min \left\{ \sum_{e \in E} \mathbf{s}_e \cdot w_e : \mathbf{s}_e \textit{ задовольняє } 2.1 \right\}. \tag{2.3}$$

Доведення. Для доведення рівності покажемо, що для довільного плану транспортування можна отримати потік, що задовольняє умові (2.1), і з довільного адекватного потоку можна отримати план транспортування з однаковими цінами. Перше покаже, що оптимальна

ціна транспортування не менша за ціну оптимального потоку; друге покаже, що ціна оптимального потоку не менша за ціну оптимального транспортування.

Позначимо через $\mathbf{P} \in \mathbf{U}(\mathbf{a}, \mathbf{b})$ деякий транспортний план, а через C_{uv} – множину ребер, що сполучають вершини $u, v \in V$ і таку, що сума

$$\sum_{e \in C_{uv}} w_e$$

є мінімальною. Тобто, C_{uv} – шлях мінімальної ваги між $u, v \in V$; у випадку існування декількох шляхів мінімальної ваги між вершинами $u, v \in V$ обираємо один згідно деякої детермінованої процедуру (алгоритм Дейкстри тощо). Покладемо $\mathbf{s} = \mathbf{0}$. Наступна процедура перетворить $\mathbf{s}$ в потік, що задовольняє (2.1):

1) Для кожної пари вершин $(u, v) \in V^2$, якщо значення $\mathbf{P}_{uv}$ більше 0, знайти один шлях мінімальної довжини C_{uv};

2) Для кожного ребра $e \in C_{uv}$ збільшити значення $\mathbf{s}_e$ на величину $\mathbf{P}_e$.

Визначимо ціну такого потоку:

$$\sum_{e \in E} \mathbf{s}_e \cdot w_e = \sum_{(u,v) \in E} \mathbf{s}_{uv} \cdot w_{uv}.$$

Для цього розпишемо $\mathbf{s}_{uv}$. Отримаємо:

$$\sum_{e \in E} \mathbf{s}_e \cdot w_e = \sum_{(u,v) \in E} w_{uv} \cdot \left(\sum_{(i,j) \in V^2 : (u,v) \in C_{ij}} \mathbf{P}_{ij} \right).$$

Змінимо порядок підсумовування в правій частині останньої рівності:

$$\sum_{(u,v) \in E} w_{uv} \cdot \left(\sum_{(i,j) \in V^2 : (u,v) \in C_{ij}} \mathbf{P}_{ij} \right) = \sum_{(i,j) \in V^2} \mathbf{P}_{ij} \cdot \left(\sum_{(u,v) \in E : (u,v) \in C_{ij}} w_{uv} \right).$$

Помітимо, що

$$\mathbf{C}_{ij} = \sum_{(u,v)\in E:(u,v)\in C_{ij}} w_{uv}.$$

за означенням матриці $\mathbf{C}$. Таким чином

$$\sum_{e\in E} \mathbf{s}_e \cdot w_e = \sum_{(i,j)\in V^2} \mathbf{P}_{ij} \cdot \left(\sum_{(u,v)\in E:(u,v)\in C_{ij}} w_{uv} \right) = \sum_{(i,j)\in V^2} \mathbf{P}_{ij} \cdot \mathbf{C}_{ij}.$$

Звідки

$$\min_{\mathbf{P}\in\mathbf{U}(\mathbf{a},\mathbf{b})} \sum_{i,j} \mathbf{P}_{ij} \cdot \mathbf{C}_{ij} \geq \min \left\{ \sum_{e\in E} \mathbf{s}_e \cdot w_e : \mathbf{s}_e \text{ задовольняє } (2.1) \right\}.$$

Доведемо зворотню нерівність. Для цього побудуємо транспортний план з потоку.

Нехай $\mathbf{s}$ – потік, що задовольняє умові (2.1). Визначимо орієнтований граф $G_s = (V_s, E_s)$, де $V_s = \{v \in V : \exists u \in V : \mathbf{s}_{(u,v)} \geq 0 \vee \mathbf{s}_{(v,u)} \geq 0\}$, $E_s = \{e \in E : \mathbf{s}_e \geq 0\}$. Тобто граф G_s – це граф, що складається з усіх вершин і ребер, які мають невід'ємне значення потоку $\mathbf{s}$.

Визначимо множини джерел S та стоків D як

$$S = \{v \in V : \mathbf{a}_v - \mathbf{b}_v < 0\};$$
$$D = \{v \in V : \mathbf{a}_v - \mathbf{b}_v > 0\}.$$

Нехай $\mathbf{s}$ – деякий потік, що задовольняє (2.1). Визначимо нульову матрицю $\mathbf{P}$ розміру $n \times n$.

1) Поки $\mathbf{s} \neq \mathbf{0}$;

2) Для всіх вершин $s \in S$, для всіх вершин $d \in D$;

3) Знайти найкоротший шлях C_{sd} графа G_s, де $\forall e \in C_{sd} : w_e > 0$;

4) Визначити значення $\delta_{sd} := \min_{e\in C_{sd}} \mathbf{s}_e$;

5) Додати δ_{sd} до $\mathbf{P}_{sd}$;

6) Для всіх $e \in C_{sd}$ зменшити $\mathbf{s}_e$ на δ_{sd}.

Визначимо ціну такого транспортного плану. Будемо вважати, що під

час роботи алгоритму на кроці 3 було виявлено k шляхів; позначимо k-й шлях C^k. Таким чином, ціна транспортування рівна

$$\sum_k \delta_k \cdot \sum_{(i,j)\in C^k} w_{ij} = \sum_k \sum_{(i,j)\in C^k} (\delta_k \cdot w_{ij}).$$

Змінимо порядок підсумовування:

$$\sum_{(i,j)\in E_s} \sum_{k:(i,j)\in C^k} (\delta_k \cdot w_{ij}) = \sum_{(i,j)\in E_s} w_{ij} \left(\sum_{k:(i,j)\in C^k} \delta_k \right).$$

Помітимо, що для довільного ребра $(i,j) \in E_s$ вираз

$$\sum_{k:(i,j)\in C^k} \delta_k, \tag{2.4}$$

підраховує $\mathbf{s}_{ij}$. Дійсно, процедура описана вище працює доки потік $\mathbf{s}$ не вичерпається. Таким чином, вираз (2.4) є свого роду «розшаруванням» значення $\mathbf{s}_{ij}$. Звідки

$$\sum_{(i,j)\in E_s} \sum_{k:(i,j)\in C^k} (\delta_k \cdot w_{ij}) = \sum_{(i,j)\in E_s} w_{ij} \left(\sum_{k:(i,j)\in C^k} \delta_k \right) =$$

$$= \sum_{(i,j)\in E_s} w_{ij} \cdot \mathbf{s}_{ij}.$$

Отже

$$\sum_{i,j} \mathbf{P}_{ij} \cdot \mathbf{C}_{ij} = \sum_k \delta_k \cdot \sum_{(i,j)\in C^k} w_{ij} = \sum_{(i,j)\in E_s} w_{ij} \cdot \mathbf{s}_{ij}.$$

Це доводить нерівність

$$\min_{\mathbf{P}\in\mathbf{U}(\mathbf{a},\mathbf{b})} \sum_{i,j} \mathbf{P}_{ij} \cdot \mathbf{C}_{ij} \le \min \left\{ \sum_{e\in E} \mathbf{s}_e \cdot w_e : \mathbf{s}_e \text{ задовольняє } (2.1) \right\}.$$

звідки отримуємо твердження теореми. □

Теорему 2.1 можна розглянути з іншого боку: нехай кожна вершина $v \in V$ – це частинка з деякою масою $\mathbf{a}_v - \mathbf{b}_v$, а $e \in E$ – «елементарний

об'єм». В такому разі пошук оптимального плану це визначення руху індивідуальної частинки $v \in V$ по «всьому об'єму», а визначення оптимального потоку – це визначення загальної маси частинок, що проходить по конкретному «елементарному об'єму» – ребру $e \in E$.

Подібні підходи відомі в гідродинаміці (і теорії поля загалом) як *лагранжевий* та *ейлерів* підхід відповідно. Обидва підходи є еквівалентними і пов'язані рівнянням неперервності, що де-факто є аналогом рівняння збереження маси, аналогічно до (2.1).

Висновки до розділу 2

У цьому розділі було надано визначення аналогів диверенціальних операторів, за допомогою яких було розроблено формулювання задачі Канторовича (1.4) та задачі мінімального потоку (1.5) на випадок графів.

Було конструктивно доведено еквівалентність задач на графу і явно описана процедура перетворення транспортного плану на потік і навпаки.

Наступний розділ буде присвячено формулюванню понять, що дозволяють чисельно перевірити коректність та безпосередньо перевірці результатів поточного розділу.

3 ПОРІВНЯННЯ РЕЗУЛЬТАТІВ

У цьому розділі наведено короткі теоретичні відмості про алгоритмічні розв'язки задачі оптимального траснспортування та мінімального потоку. Наведено порівняння резлутатів знаходження оптимального транспортування на пряму і використовуючи мінімальний потік. Наведено аналіз отриманих результатів.

3.1 Алгоритмічне підґрунтя

Коротко розглянемо теоретичні відомості, що дозволять чисельно порівняти два підходи отримання оптимального транспортного плану: розв'язок лінійной задачі і перетворення мінімального потоку

3.1.1 Задача Канторовича

Наведемо спосіб перетворити дискретну задачу Канторовича (2.2) у задачу лінійного програмування У стандартній формі.

Твердження 3.1 (Задача Канторовича як задача лінійного програмування [7])**.** *Нехай* $\mathbf{C}$ *– матриця цін,* $\mathbf{a}, \mathbf{b}$ *– дві дискретні міри та* $\mathbf{I}_n$ *– одинична матриця* $n \times n$*. Визначимо матрицю*

$$\mathbf{A} = \begin{bmatrix} \mathbb{1}_n^\top \otimes \mathbf{I}_m \\ \\ \mathbf{I}_n \otimes \mathbb{1}_m^\top \end{bmatrix}.$$

Визначимо вектор $\mathbf{c}$*, як вектор, отриманий шляхом з'єднання колонок*

матриці $\mathbf{C}$ *у вектор-стовпчик та вектор* $\mathbf{k}$ *як*

$$\mathbf{k} = \begin{bmatrix} \mathbf{a} \\ \\ \mathbf{b} \end{bmatrix}.$$

Тоді

$$\min_{\mathbf{P} \in U(\mathbf{a},\mathbf{b})} \sum_{i,j} \mathbf{P}_{ij} \cdot \mathbf{C}_{ij} = \min_{\substack{\mathbf{p} \in \mathbb{R}_+^{nm} \\ \mathbf{Ap}=\mathbf{k}}} \mathbf{c}^\top \mathbf{p}. \tag{3.1}$$

Відповідно пошук оптимального плану транспортування можна звести до розв'язання задачі лінійного програмування (3.1)

3.1.2 Задача пошуку мінімального потоку

Визначимо **залишковий граф** (*англ.* Residual Network).

Означення 3.1 (Залишковий граф)**.** Для графа $G = (V, E, w)$ і потоку $\mathbf{s}$ визначимо **залишковий граф** $G^s = (V, E^s, w^s)$, де $E^s = \{e \in E : \mathbf{s}_e > 0\}$ і

$$w^s_{u,v} = \begin{cases} w_{u,v}, & \text{якщо } (u,v) \in E \\ -w_{v,u}, & \text{якщо } (v,u) \in E \end{cases}.$$

Наступне твердження є фундаментом алгоритму усунення циклів (*англ.* Cycle Cancelling algorithm) [5], який буде використаний в якості алгоритму пошуку мінімального потоку на графі.

Теорема 3.1 (Критерій мінімальності потоку [2])**.** *Потік* $\mathbf{s}$ *на графі* G *мінімальним тоді і лише тоді, коли граф* G^s *не має циклів від'ємної ціни.*

Отримавши оптимальний потік, ми можемо отримати оптимальний план, використавши процедуру, наведену в доведенні теореми 2.1.

3.2 Порівняння результатів

Для порівняння роботи алгоритмів розглянемо декілька графів із «складною» системою ребер. Гарним прикладом таких графів є так звані **повні графи**, що мають усі можливі ребра. Ваги ребер, розподіли $\mathbf{a}$ та $\mathbf{b}$ будуть зазначені окремо в таблицях, щоб не засмічувати рисунки. Далі через K_n будемо позначати повний граф на n вершинах

Спочатку розглянемо повний граф K_5 на 5 вершинах з відповідними

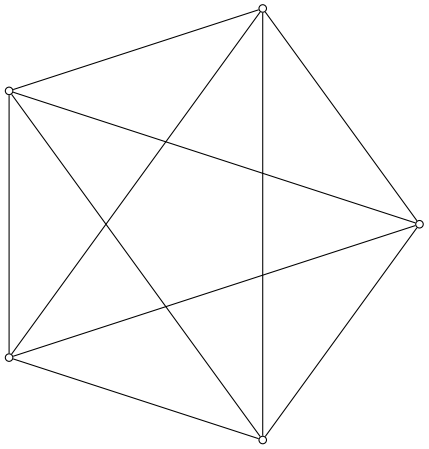

Рисунок 3.1 – Повний граф на 5 вершинах

розподілами $\mathbf{a}, \mathbf{b}$ (таблиця 3.1) та вагами

v	**1**	**2**	**3**	**4**	**5**
a	6	4	10	8	9
b	4	2	14	9	8

Таблиця 3.1 – Таблиця значень розподілів $\mathbf{a}, \mathbf{b}$ на вершинах графа K_5

$$\mathbf{C} = \begin{pmatrix} 0 & 7 & 6 & 10 & 5 \\ 7 & 0 & 7 & 3 & 8 \\ 8 & 7 & 0 & 5 & 2 \\ 10 & 3 & 5 & 0 & 9 \\ 5 & 8 & 2 & 9 & 0 \end{pmatrix}.$$

Розв'язавши лінійну задачу (3.1), ми отримаємо наступний оптимальний план

$$\mathbf{P}_T^* = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix},$$

з ціною транспортування 24.

Порівняємо результат, застосувавши алгоритм усунення циклів і перетворивши оптимальний потік на транспортний план. Отримаємо

$$\mathbf{P}_F^* = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix},$$

з ціною 24.

Таким чином обидва алгоритми отримали плани, ціна яких рівна 24.

Проведемо порівняння підходів на більшому графі K_7. Розподіли

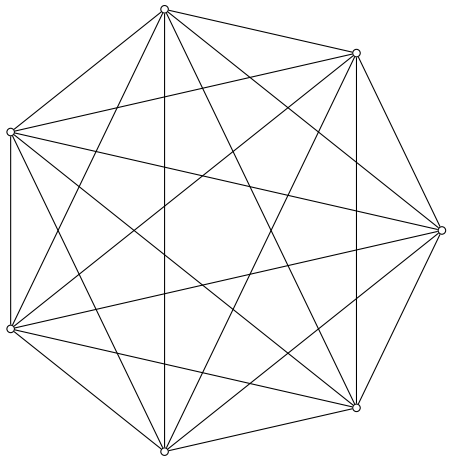

Рисунок 3.2 – Повний граф на 7 вершинах

$\mathbf{a}, \mathbf{b}$ подані в таблиці 3.2. Матрицю цін, розподіли $\mathbf{a}, \mathbf{b}$ зробимо менш випадковими. Це дозволить алгоритмам знайти різні оптимальні розв'язки.

$$\mathbf{C} = \begin{pmatrix} 0 & 2 & 1 & 2 & 1 & 2 & 1 \\ 2 & 0 & 2 & 1 & 2 & 1 & 2 \\ 1 & 2 & 0 & 2 & 1 & 2 & 1 \\ 2 & 1 & 2 & 0 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 0 & 2 & 1 \\ 2 & 1 & 2 & 1 & 2 & 0 & 2 \\ 1 & 2 & 1 & 2 & 1 & 2 & 0 \end{pmatrix}.$$

Розв'язання лінійної задачі дає такий оптимальний транспортний план

v	**1**	**2**	**3**	**4**	**5**	**6**	**7**
a	1	0	1	1	0	1	0
b	0	1	0	1	1	0	1

Таблиця 3.2 – Таблиця значень розподілів $\mathbf{a}, \mathbf{b}$ на вершинах графу K_7

$$\mathbf{P}_T^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Ціна транспортування вище рівна 3. В свою чергу, за допомогою алгоритму усунення циклів було побудовано план

$$\mathbf{P}_F^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Ціна транспортування також 3. Хоча план транспортування відрізняється, легко перевірити, що він дійсно переносить **a** на **b**.

Висновки до розділу 3

В цьому розділі було розглянуто алгоритмічне підґрунтя для розв'язання задач оптимального транспортування і мінімального потоку.

Було отримано результати, що свідчать про коректність процедури. В першому випадку плани траснспортування співпадають, але в другому ми наочно бачимо різницю. Тим не менш, як було доведено і перевірено, оптимальна ціна транспортування лишалась однаковою.

ВИСНОВКИ

У цій роботі було досліджено математичну модель задачі побудови оптимального плану транспортування і задачі побудови мінімального потоку.

У параграфі 1.1 було розглянуто неперевне формулювання задачі Монжа-Канторовича (1.4) тоді як параграф 1.2 було присвячено задачі пошуку мінімального потоку.

У розділі 2 було розглянуто дискретні формулювання цих задач. В якості природнього дискретного аналогу метричного простору було обрано зв'язний не орієнтований граф з невід'ємними вагами. Було розроблено аналоги диференціальних операторів для графу, а наведені аналоги були використані при формулюванні задачі пошуку мінімального потоку.

Також було доведено еквівалентність поставлених задач на графі у теоремі 2.1.

Розділ 3 було присвяченно питанню чисельної перевірки одержаних теоретичних результатів. Було наведено твердження, що дозволило розв'язати задачу Монжа-Канторовича на графі як задачу лінійного програмування і посилання на алгоритм, для розв'язання задачі мінімального потоку.

Процедура з доведення теореми 2.1 була запрограмована мовою Python і наведена в додатку. Коректність роботи процедури була перевірена на двох прикладах. Проведені чисельні експерименти показали, що побудовані двома способами плани є оптимальними, хоча можуть відрізнятись.

ПЕРЕЛІК ПОСИЛАНЬ

1. *Beckmann M.* A Continuous Model of Transportation // Econometrica. — 1952. — Т. 20, № 4. — С. 643—660. — URL: https://onlinelibrary.wiley.com/doi/abs/0012-9682(195210)20:4<643:ACMOT>2.0.CO;2-G.
2. *Busacker R.*, *Saaty T.* Finite Graphs and Networks: An Introduction with Applications. — McGraw-Hill, 1965. — (International series in pure and applied mathematics). — ISBN 9780070093058. — URL: https://books.google.com.ua/books?id=j0NDAAAAIAAJ.
3. *Carlier G.*, *Santambrogio F.* A Variational Model for Urban Planning with Traffic Congestion // ESAIM: COCV. — 2005. — Т. 11, № 4. — С. 595—613. — URL: http://cvgmt.sns.it/paper/72/ ; cvgmt preprint.
4. *Kantorovich L.* On the Translocation of Masses // Journal of Mathematical Sciences. — 2006. — Бер. — Т. 133. — DOI: 10.1007/s10958-006-0049-2.
5. *Klein M.* A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems // Management Science. — 1967. — Т. 14, № 3. — С. 205—220. — ISSN 00251909, 15265501. — URL: http://www.jstor.org/stable/2628396.
6. *Monge G.* Mémoire sur la théorie des déblais et des remblais. — De l'Imprimerie Royale, 1781.
7. *Peyré G.*, *Cuturi M.* Computational Optimal Transport // Foundations and Trends in Machine Learning. — 2019. — Т. 11, № 5/6. — С. 355—607.
8. *Santambrogio F.* Optimal Transport for Applied Mathematicians: Calculus of Variations, PDEs, and Modeling. — Springer International Publishing, 2016. — (Progress in Nonlinear Differential Equations and Their Applications). — ISBN 9783319365817. — URL: https://books.google.com.ua/books?id=2KTXtAEACAAJ.

9. *Villani C.*, *Society A. M.* Topics in Optimal Transportation. — American Mathematical Society, 2003. — (Graduate studies in mathematics). — ISBN 9781470418045. — URL: https://books.google.com.ua/books?id=MyPjjgEACAAJ.

ДОДАТОК А ТЕКСТ ПРОГРАМИ

А.1 Програма 1

Програмний код на мовою Python для побудови оптимального плану транспортування на не орієнтованому невід'ємно визначеному графі $G = (V, E, w)$.

```
import numpy as np
import collections

# --- Reusable Edge Class ---
class Edge:
    def __init__(self, to, capacity, cost, reverse_edge_idx):
        self.to = to
        self.capacity = capacity  # Original capacity
        self.cost = cost          # Original cost per unit of flow
        self.reverse_edge_idx = reverse_edge_idx # Index in graph[to] of the reverse edge
        self.flow = 0.0

# --- Edmonds-Karp for Initial Feasible Flow ---
def _bfs_for_edmonds_karp(graph, s, t, num_vertices, parent_nodes, parent_edges):
    """
    Performs BFS to find an augmenting path in the residual graph (capacities only).
    Fills parent_nodes and parent_edges to reconstruct the path.
    Returns True if a path is found, False otherwise.
    """
    for i in range(num_vertices): # Reset parents
        parent_nodes[i] = -1
        parent_edges[i] = None

    queue = collections.deque()
    queue.append(s)

    visited = [False] * num_vertices
    visited[s] = True

    while queue:
```

```
        u = queue.popleft()
        if u == t:
            return True # Path found

        for edge in graph[u]:
            residual_capacity = edge.capacity - edge.flow
            if not visited[edge.to] and residual_capacity > 1e-9: # Check for usable capacity
                visited[edge.to] = True
                parent_nodes[edge.to] = u
                parent_edges[edge.to] = edge
                queue.append(edge.to)
    return False

def _find_initial_feasible_flow_edmonds_karp(graph, s, t, num_vertices, required_flow):
    """
    Finds an initial feasible flow up to required_flow using Edmonds-Karp.
    Modifies flows on the Edge objects in 'graph'.
    Returns the total flow achieved.
    """
    current_total_flow = 0.0
    parent_nodes = [-1] * num_vertices
    parent_edges = [None] * num_vertices # Stores the Edge object leading to a node

    while current_total_flow < required_flow - 1e-9:
        if not _bfs_for_edmonds_karp(graph, s, t, num_vertices, parent_nodes, parent_edges):
            break # No more augmenting paths

        path_bottleneck = float('inf')
        curr = t
        while curr != s:
            edge_to_curr = parent_edges[curr]
            path_bottleneck = min(path_bottleneck, edge_to_curr.capacity - edge_to_curr.flow)
            curr = parent_nodes[curr]

        flow_to_send_on_path = min(path_bottleneck, required_flow - current_total_flow)
        if flow_to_send_on_path < 1e-9:
            break # Bottleneck is too small or demand nearly met

        v = t
```

```
70          while v != s:
71              u = parent_nodes[v]
72              edge_uv = parent_edges[v] # Edge u -> v
73
74              edge_uv.flow += flow_to_send_on_path
75              # Update flow on reverse edge v -> u
76              graph[v][edge_uv.reverse_edge_idx].flow -= flow_to_send_on_path
77              v = u
78
79          current_total_flow += flow_to_send_on_path
80
81      return current_total_flow
82
83  # --- Bellman-Ford for Negative Cycle Detection ---
84  def _find_negative_cycle_bellman_ford(graph, num_vertices):
85      """
86      Finds a negative cost cycle in the residual graph using Bellman-Ford.
87      The graph contains Edge objects with current flows and original costs.
88      Residual capacities and costs are derived from these.
89      Returns (list_of_cycle_edges, bottleneck_capacity, cycle_cost) or (None, 0, 0).
90      """
91      distance = [0.0] * num_vertices # Initialize distances to 0 to find any neg cycle
92      predecessor_node = [-1] * num_vertices
93      predecessor_edge_obj = [None] * num_vertices # Stores the Edge object from graph
94
95      # Relax edges V-1 times
96      for _ in range(num_vertices - 1):
97          for u_node in range(num_vertices):
98              for edge in graph[u_node]: # edge is an Edge object from u_node to edge.to
99                  residual_capacity = edge.capacity - edge.flow
100                 if residual_capacity > 1e-9:
101                     # Cost of using this residual edge is edge.cost
102                     if distance[edge.to] > distance[u_node] + edge.cost:
103                         distance[edge.to] = distance[u_node] + edge.cost
104                         predecessor_node[edge.to] = u_node
105                         predecessor_edge_obj[edge.to] = edge
106
107     # Check for negative cycles (V-th iteration)
108     node_on_cycle_candidate = -1
```

```
    for u_node in range(num_vertices):
        for edge in graph[u_node]:
            residual_capacity = edge.capacity - edge.flow
            if residual_capacity > 1e-9:
                if distance[edge.to] > distance[u_node] + edge.cost + 1e-9:
                    # Negative cycle detected (or reachable from one)
                    # To ensure 'start_node_for_reconstruction' is actually on the cycle:
                    predecessor_node[edge.to] = u_node # Update for Vth iter relaxation
                    predecessor_edge_obj[edge.to] = edge

                    node_on_cycle_candidate = edge.to
                    for _ in range(num_vertices):
                        if predecessor_node[node_on_cycle_candidate] == -1:
                            node_on_cycle_candidate = -1 # Error or no cycle via this path
                            break
                        node_on_cycle_candidate = predecessor_node[node_on_cycle_candidate]

                    if node_on_cycle_candidate != -1:
                        break # Found a starting point for cycle reconstruction
        if node_on_cycle_candidate != -1:
            break

    if node_on_cycle_candidate == -1:
        return None, 0, 0 # No negative cycle found

    # Reconstruct the cycle starting from node_on_cycle_candidate
    cycle_edges = []
    visited_in_reconstruction = [False] * num_vertices
    curr = node_on_cycle_candidate

    while not visited_in_reconstruction[curr]:
        visited_in_reconstruction[curr] = True
        # The edge used to reach curr is predecessor_edge_obj[curr]
        # Its source is predecessor_node[curr]
        if predecessor_edge_obj[curr] is None: return None, 0, 0 # Should not happen here

        curr = predecessor_node[curr] # Move to the actual start of path that forms cycle.

    # Now curr is node_on_cycle_candidate and has been visited.
```

```
148         # Trace again to collect edges.
149         path_to_form_cycle = []
150         temp_curr = curr # curr is the start of the cycle (node_on_cycle_candidate after N backtracks)
151 
152         cycle_cost = 0.0
153         cycle_bottleneck = float('inf')
154 
155         for _ in range(num_vertices + 1): # Max V edges in a simple cycle
156             edge_leading_to_temp_curr = predecessor_edge_obj[temp_curr]
157             if edge_leading_to_temp_curr is None: return None, 0, 0 # Should not happen
158 
159             path_to_form_cycle.append(edge_leading_to_temp_curr)
160             cycle_cost += edge_leading_to_temp_curr.cost
161             cycle_bottleneck = min(cycle_bottleneck,
162                                    edge_leading_to_temp_curr.capacity - edge_leading_to_temp_curr.flow)
163 
164             temp_curr = predecessor_node[temp_curr]
165             if temp_curr == curr: # We have completed the cycle
166                 break
167         else: # Did not return to start, something is wrong
168             return None, 0, 0
169 
170         path_to_form_cycle.reverse() # To get edges in cycle order
171 
172         if cycle_cost < -1e-9 and cycle_bottleneck > 1e-9: # Ensure it's a valid, usable negative cycle
173             return path_to_form_cycle, cycle_bottleneck, cycle_cost
174         else: # Numerically not a useful negative cycle, or error in reconstruction
175             return None, 0, 0
176 
177 
178     # --- Main Cycle Canceling Algorithm Wrapper ---
179     def min_cost_cycle_canceling(
180         original_graph_edges,
181         node_supplies,
182         node_demands,
183         num_original_nodes
184     ):
185         if len(node_supplies) != num_original_nodes:
186             raise ValueError(f"Length of node_supplies must equal num_original_nodes.")
```

```
187     if len(node_demands) != num_original_nodes:
188         raise ValueError(f"Length of node_demands must equal num_original_nodes.")
189 
190     _supplies_info = []
191     _total_supply = 0.0
192     for i in range(num_original_nodes):
193         if node_supplies[i] < -1e-9: raise ValueError(f"Supply at node {i} cannot be negative.")
194         if node_supplies[i] > 1e-9:
195             _supplies_info.append((i, node_supplies[i]))
196             _total_supply += node_supplies[i]
197 
198     _demands_info = []
199     _total_demand = 0.0
200     for i in range(num_original_nodes):
201         if node_demands[i] < -1e-9: raise ValueError(f"Demand at node {i} cannot be negative.")
202         if node_demands[i] > 1e-9:
203             _demands_info.append((i, node_demands[i]))
204             _total_demand += node_demands[i]
205 
206 
207     super_source_idx = num_original_nodes
208     super_sink_idx = num_original_nodes + 1
209     num_transformed_nodes = num_original_nodes + 2
210 
211     # Build the graph structure for flow algorithms
212     # This graph will be modified by initial flow and cycle canceling
213     _graph = [[] for _ in range(num_transformed_nodes)]
214     # Keep track of original edges to report flow on them
215     _original_edge_references = []
216 
217     def _add_edge_pair_to_graph(u, v, cap, cost, is_original=False):
218         fwd_edge = Edge(v, cap, cost, len(_graph[v]))
219         bwd_edge = Edge(u, 0, -cost, len(_graph[u]))
220         _graph[u].append(fwd_edge)
221         _graph[v].append(bwd_edge)
222         if is_original:
223             _original_edge_references.append({'u': u, 'v': v, 'edge_obj': fwd_edge,
224                                               'original_capacity': cap, 'original_cost': cost})
225 
```

```
    for u, v, cap, cost_val in original_graph_edges:
        if not (0 <= u < num_original_nodes and 0 <= v < num_original_nodes):
            raise ValueError(f"Edge ({u},{v}) has out-of-bounds nodes for original graph.")
        _add_edge_pair_to_graph(u, v, cap, cost_val, is_original=True)

    for s_node, s_amount in _supplies_info:
        _add_edge_pair_to_graph(super_source_idx, s_node, s_amount, 0)
    for d_node, d_amount in _demands_info:
        _add_edge_pair_to_graph(d_node, super_sink_idx, d_amount, 0)

    # 1. Find Initial Feasible Flow
    initial_flow_achieved = _find_initial_feasible_flow_edmonds_karp(
        _graph, super_source_idx, super_sink_idx, num_transformed_nodes, _total_supply
    )


    # Calculate initial cost based on the feasible flow
    current_min_cost = 0.0
    for item in _original_edge_references: # Only sum costs for original edges
        current_min_cost += item['edge_obj'].flow * item['original_cost']

    # 2. Iteratively Cancel Negative Cycles
    iteration_count = 0
    max_iterations = num_transformed_nodes * num_transformed_nodes * len(original_graph_edges)

    while iteration_count < max_iterations : # Add a safety break for complex cases
        iteration_count += 1
        cycle_edge_list, bottleneck_delta, cycle_c = \
        _find_negative_cycle_bellman_ford(_graph, num_transformed_nodes)

        if not cycle_edge_list:
            break # No more negative cycles, current flow is optimal

        # Augment flow along the cycle
        for edge_in_cycle in cycle_edge_list:
            edge_in_cycle.flow += bottleneck_delta
            _graph[edge_in_cycle.to][edge_in_cycle.reverse_edge_idx].flow -= bottleneck_delta

        # Update cost: cycle_c is the sum of costs, it's negative
```

```
265             current_min_cost += bottleneck_delta * cycle_c
266
267
268         if iteration_count >= max_iterations and cycle_edge_list:
269             print("Warning (CC): Reached max iterations, cycle canceling might be slow or stuck.")
270
271         # Prepare final flow distribution for reporting (on original edges)
272         final_flow_distribution_original = []
273         for item in _original_edge_references:
274             edge = item['edge_obj']
275             if edge.flow > 1e-9:
276                 final_flow_distribution_original.append(
277                     (item['u'], item['v'], edge.flow, item['original_capacity'], item['original_cost'])
278                 )
279
280         return current_min_cost, initial_flow_achieved, final_flow_distribution_original
281
282     def flow_decomposition_to_transportation_plan(f, S, D):
283         """
284         Decomposes a network flow into a Kantorovich transportation plan.
285         Handles transshipment nodes.
286         """
287         # Create a mutable copy of the flow for updates
288         residual_flow = collections.defaultdict(lambda: collections.defaultdict(int))
289         for u in f:
290             for v in f[u]:
291                 if f[u][v] > 0:
292                     residual_flow[u][v] = f[u][v]
293
294         transportation_plan = []
295         num_nodes = len(S)
296
297         while True:
298             source = -1
299             # Find a node with a net positive outflow in the residual graph
300             for i in range(num_nodes):
301                 out_flow = sum(residual_flow[i].values())
302                 in_flow = sum(residual_flow[j][i] for j in range(num_nodes))
303                 if out_flow > in_flow:
```

```
                source = i
                break

        if source == -1:
            break  # No more flow to decompose

        # Find a path from the source to a sink using DFS
        stack = [(source, [source])]
        path_found = None

        # Keep track of visited nodes to avoid cycles in the current path search
        visited_in_dfs = {source}

        while stack:
            u, path = stack.pop()

            # A sink for a path is a node that is a net demander in the overall problem
            if S[u] - D[u] < 0:
                path_found = path
                break

            for v in sorted(list(residual_flow[u].keys())):
                if residual_flow[u][v] > 0 and v not in visited_in_dfs:
                    visited_in_dfs.add(v)
                    stack.append((v, path + [v]))

        if path_found:
            path = path_found

            # Determine bottleneck capacity of the found path
            bottleneck = float('inf')
            for i in range(len(path) - 1):
                u_p, v_p = path[i], path[i+1]
                bottleneck = min(bottleneck, residual_flow[u_p][v_p])

            transportation_plan.append((path, bottleneck))

            # Update the residual flow by subtracting the bottleneck amount
            for i in range(len(path) - 1):
```

```
                u, v = path[i], path[i+1]
                residual_flow[u][v] -= bottleneck
        else:
            # If no path is found from any source, we are done
            break

    return transportation_plan



# --- Example Usage ---
if __name__ == '__main__':
# --- Test Cycle Canceling Algorithm ---
    n = 7

    # Supply and demand arrays
    supplies = np.array([1, 2, 3, 4, 5, 6, 7])
    demands = supplies[::-1]

    # Initialize cost matrix
    costMatrix = np.array([
        [0,2,1,2,1,2,1],
        [2,0,2,1,2,1,2],
        [1,2,0,2,1,2,1],
        [2,1,2,0,2,1,2],
        [1,2,1,2,0,2,1],
        [2,1,2,1,2,0,2],
        [1,2,1,2,1,2,0],
    ])


    # Initialize and populate edge list
    edges = []
    for i in range(n):
        for j in range(i + 1, n):
            edges.append(
                (i, j, float("inf"), costMatrix[i, j])
            )
            edges.append(
```

```
                (j, i, float("inf"), costMatrix[i, j])
            )

    # Retrieve minimal flow
    cc_cost, cc_flow_moved, cc_flow_details = \
        min_cost_cycle_canceling(
            edges,
            supplies,
            demands,
            n
        )

    # Cast flow to a proper format
    flow_dict_of_dicts = collections.defaultdict(lambda: collections.defaultdict(int))
    for u, v, f, _, __ in cc_flow_details:
        flow_dict_of_dicts[u][v] = f
        print((u, v), "->", f)

    # Run the decomposition
    plan = flow_decomposition_to_transportation_plan(flow_dict_of_dicts, supplies, demands)

    # --- Display the Results ---
    print("Decomposition of the Feasible Flow:")
    print("-" * 40)
    total_decomposed_flow = 0
    for path, amount in plan:
        total_decomposed_flow += amount
        print(f"  Path: {' -> '.join(map(str, path))}, Amount: {amount}")
    print("-" * 40)
    print(f"Total flow decomposed: {total_decomposed_flow}")

    # The sum of net supplies should match the total decomposed flow
    total_net_supply = sum(max(0, s_i - d_i) for s_i, d_i in zip(supplies, demands))
    print(f"Total net supply to be shipped: {total_net_supply}")
```