m-math_2025_012

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Факультет математики та інформатики
Кафедра математичного моделювання

Оптимізація потоку в мережі: Реалізація алгоритмів у веб-додатку

Кваліфікаційна робота
Рівень вищої освіти – друга(магістр)

Виконав:
Студент 6 курсу, групи 601
спеціальності 122 – Комп'ютерні науки

Росошанський Олег Мирославович

Керівник:
доктор фізико-математичних наук,
доцент Клевчук І.І.

До захисту допущено
на засіданні кафедри
протокол № 5 від 26 листопада 2024 р..
Зав. кафедрою _____ проф. Черевко І.М.

Чернівці – 2024

АНОТАЦІЯ

Веб-додаток, побудований на основі фреймворку Angular та мови TypeScript, використовує проштовхування передпотоку, пошуку заданого потоку з мінімальною ціною. Ці алгоритми використовуються для розв'язання задач, пов'язаних з потоками в мережах, і є важливими для оптимізації транспортних систем, розподілу ресурсів та інших подібних задач.

Використання фреймворку Angular дозволяє побудувати потужний та ефективний веб-додаток. Angular має багатий набір функцій та компонентів, що допомагають розробникам створювати зручний і легко масштабований інтерфейс користувача. TypeScript, у свою чергу, надає переваги статичного типізації, поліпшеного синтаксису та більшого контролю над розробкою програмного коду.

Завдяки поєднанню Angular і TypeScript, веб-додаток забезпечує продуктивну та надійну роботу, а також простоту розробки та підтримки.

Ключові слова: Веб-додаток, Angular, TypeScript, алгоритми потоків, проштовхування передпотоку, максимальний потік, Алгоритм Форда-Фалкерсона.

ANNOTATION

A web application built using the Angular framework and TypeScript language implements push-relabel and minimum-cost flow algorithms. These algorithms are used to solve network flow problems and are crucial for optimizing transportation systems, resource allocation, and similar tasks.

Using the Angular framework facilitates the creation of a powerful and efficient web application. Angular offers a rich set of features and components that help developers design a user-friendly and easily scalable interface. TypeScript, in turn, provides the benefits of static typing, enhanced syntax, and greater control over the development process.

Thanks to the combination of Angular and TypeScript, the web application ensures productive and reliable performance, as well as ease of development and maintenance.

Keywords: Web application, Angular, TypeScript, flow algorithms, preflow pushing, maximum flow, Ford-Fulkerson Algorithm.

The qualification work contains the results of one's own research. The use of ideas, results, and texts of scientific research by other authors must be referenced to the appropriate source.

Кваліфікаційна робота містить результати власних досліджень.Використання ідей, результатів і текстів наукових досліджень інших авторів мають посилання на відповідне джерело.

_____ О.М.Росошанський

Зміст

ВСТУП

Теорія графів є розділом дискретної математики, що досліджує властивості зліченних множин з відношеннями між їхніми елементами. Вона використовує геометричний підхід для аналізу об'єктів. У 1936 році Денеш Кеніг ввів поняття "граф", але вже у 1736 році Леонард Ейлер поставив задачу про Кенігсберзькі мости, що була першим внеском у теорію графів. Сучасна теорія графів надає зручний інструментарій для моделювання структурних властивостей систем та відношень між об'єктами. Вона знаходить застосування в математиці, інформатиці, програмуванні, хімії, економіці, логістиці, комунікаційних та транспортних системах.

Основна увага даної роботи приділяється задачі пошуку максимального потоку у мережі, а також алгоритмам його пошуку та їхній реалізації у вигляді веб-додатку. Розглядаються різні алгоритми, які дозволяють знаходити максимальний потік у мережі, і вказуються їх переваги та недоліки.

Розроблена програма, яка реалізує алгоритми пошуку максимального потоку у мережі. Пояснюється структура програми, основні функції та їх функціональні можливості. Також наводяться додатки з програмним кодом та посиланнями на використані джерела, що використовуються в роботі.

РОЗДІЛ 1. ОСНОВИ ТЕОРІЇ ГРАФІВ

1.1. Поняття графа

Граф - абстрактний математичний об'єкт, що складається з вершин та ребер, які з'єднують пари вершин. Наприклад, можна представити множину залізничних вокзалів як множину вершин, а залізничні сполучення між ними - як множину ребер. Залежно від контексту використання, графи можуть мати різні властивості, такі як напрямленість, обмеження на кількість ребер, вага та додаткова інформація про вершини і ребра. Графи є важливими для математики та інформатики, оскільки багато різних структур можна зобразити за їх допомогою. Наприклад, структуру Вікіпедії можна зобразити як орієнтований граф, де статті є вершинами, а гіперпосилання - ребрами. Графи є загальним інструментом, який можна застосовувати для вирішення різноманітних задач у транспортних мережах, комп'ютерних системах, будівництві, молекулярному моделюванні тощо. Теорія графів є предметом дослідження та містить багато невирішених проблем і гіпотез.

Теорія графів є широко використовуваним математичним інструментом з численними застосуваннями у різних областях. Наприклад, в геоінформаційних системах (ГІС) графи використовуються для моделювання дорожньої мережі, з'єднань між будинками та інженерними мережами. Будинки, споруди та квартали вважаються вершинами графа, а дороги, лінії електропередачі та інші з'єднання - ребрами. Застосовуючи різні алгоритми теорії графів до таких моделей, можна здійснювати обчислення, які допомагають знайти найкоротший шлях, планувати оптимальні маршрути чи знаходити найближчі продуктові магазини. Теорія графів залишається активним напрямком досліджень, з численними відкритими проблемами та гіпотезами, що потребують подальшого розвитку та вивчення [1].

1.2. Основні означення та властивості

Теорія графів є важливою частиною математичного апарату інформатики та кібернетики, і вона використовується для формулювання багатьох задач, пов'язаних з дискретними об'єктами. Ці задачі виникають в різних областях, таких як проектування інтегральних схем, керування, автоматичні системи, економіка, статистика, розклади і дискретна оптимізація [1].

Термін "граф" був вперше використаний у книзі відомого угорського математика Д. Кеніга у 1936 році, але перші задачі теорії графів пов'язані з іменем Л. Ейлера (XVIII ст.).

Простий граф складається з множини вершин V і множини ребер E, де V - непорожня скінченна множина елементів, які є вершинами, а E - множина невпорядкованих пар різних елементів з V, які є ребрами.

Ребро {u,v} з'єднує вершини u і v. У простому графі пару вершин може з'єднувати не більше одного ребра, оскільки E - множина. Іноді виникає потреба розглядати графи, в яких дві вершини можуть бути з'єднані більш ніж одним ребром, і для цього використовується поняття мультиграфа. Мультиграф складається з множини вершин V і сім'ї невпорядкованих пар різних елементів з V, які є ребрами. У мультиграфі елементи в сім'ї (ребра) можуть повторюватись, і ребра, які з'єднують одну й ту саму пару вершин, називаються кратними або паралельними ребрами.

Далі узагальнення полягає в тому, що, крім кратних ребер, граф може містити петлі, тобто ребра, які з'єднують вершину з самою собою. Псевдограф складається з множини вершин V і сім'ї невпорядкованих пар вершин, які не обов'язково є різними.

Теорія графів вивчає властивості та взаємозв'язки графічних структур, де вершини представляють об'єкти, а ребра вказують на зв'язки між цими об'єктами. Вона дозволяє аналізувати складні системи та проблеми шляхом моделювання їх у вигляді графів. Такі графові моделі допомагають розкрити

особливості структури даних і знайти оптимальні рішення в різних областях, включаючи транспортні мережі, соціальні мережі, маршрутизацію даних, розкладання задач, оптимальне планування та багато інших.

Термінологія та поняття, що використовуються в теорії графів, стали основою для розвитку багатьох алгоритмів та методів оптимізації. Наприклад, пошук найкоротших шляхів у графі або визначення ефективного маршруту в мережі. Теорія графів також знайшла застосування в розв'язанні задач розкладання, де необхідно знайти оптимальний розподіл ресурсів або графічну модель процесу прийняття рішень.

Зрозуміння та застосування теорії графів є невід'ємною частиною сучасних наукових досліджень і практичних застосувань у багатьох галузях. Вона допомагає розуміти складні зв'язки та виявляти оптимальні рішення в різних сферах діяльності людей.

Три вищезгаданих типи графів, які були описані раніше, відносяться до неорієнтованих графів. Псевдограф є найбільш загальним типом неорієнтованого графа, оскільки він може містити як петлі, так і кратні ребра. Мультиграф є неорієнтованим графом, який дозволяє наявність кратних ребер, але не має петель. Нарешті, простий граф є неорієнтованим графом без кратних ребер і петель.

Помімо неорієнтованих графів, існують також орієнтовані графи. Орієнтований граф складається зі множини вершин V і множини упорядкованих пар елементів з V, які називаються дугами або орієнтованими ребрами. Дуга вигляду (v, v) є петлею. У графічних представленнях, дуги зображаються у вигляді стрілок.

Орієнтований мультиграф є парою (V, E), де V є скінченною непорожньою множиною вершин, а E є сім'єю упорядкованих пар елементів з V. У цьому випадку, дуги (елементи E) можуть повторюватись, і такі дуги називаються кратними. Зазначимо, що кратні дуги з'єднують одну і ту саму пару вершин і мають однаковий напрям.

Дві вершини u та v в неорієнтованому графі G вважаються суміжними,

якщо {u, v} є ребром з множини E. Якщо e = {u, v} є ребром, то вершини u та v вважаються його кінцями. Два ребра називаються суміжними, якщо вони мають спільний кінець. Вершину v і ребро e вважають інцидентними, якщо v є кінцем ребра e. Слід зазначити, що суміжність є зв'язком між однорідними елементами графа, тоді як інцидентність є зв'язком між різнорідними елементами графа.

Степінь вершини в неорієнтованому графі визначається як кількість ребер, які зв'язані з цією вершиною, при цьому петлю враховують двічі. Степінь вершини v позначається deg(v). Якщо deg(v) = 0, то вершину v називають ізольованою, а якщо deg(v) = 1, то вершину v називають висячою.

У випадку орієнтованого мультиграфа G = (V, E), якщо (u, v) ∈ E, то вершину u називають початковою або ініціальною, а вершину v – кінцевою або термінальною вершиною дуги e = (u, v). Петля має початок і кінець в одній вершині.

У випадку орієнтованого графа змінюються визначення степеня вершини. В орієнтованому мультиграфі, півстепенем входу вершини v називають кількість дуг, які мають вершину v як кінцеву, і це позначається як deg-(v). Півстепенем виходу вершини v називають кількість дуг, для яких вершина v є початковою, і це позначається як deg+(v).

Простий граф H = (W, F) називається підграфом простого графа G = (V, E), якщо W є підмножиною множини V, а F є підмножиною E.

Ка́рка́сний підграф (або фактор) H називають таким підграфом, для якого W = V.

Якщо W ≠ V, а F є множиною всіх ребер з E, таких що ці ребра мають кінці в W, то підграф H називають породженим (або індукованим) множиною W, і позначають як G(W).

Звертаючись до властивостей графів, важливо зазначити, що степінь вершини в неорієнтованому графі може бути використана для вивчення його структури. Високі степені вершин часто вказують на центральні або важливі елементи графа, які мають більше зв'язків з іншими вершинами. З іншого

боку, ізольовані та висячі вершини можуть вказувати на окремі частини графа, що можуть мати особливу роль або значення в контексті досліджуваної проблеми.

У випадку орієнтованих графів, поняття півстепенів входу та виходу дозволяють аналізувати напрямок інформаційного потоку або взаємозв'язки між елементами системи, де граф є моделлю. Це корисне для розуміння спрямованості комунікації, передачі даних або взаємодії між різними частинами системи.

Такі поняття, як підграфи та їх класифікація, дозволяють розглядати графи на різних рівнях деталей та узагальнювати їх структуру для подальшого аналізу. Це допомагає зменшити складність задачі, розглядаючи лише певну підмножину вершин та ребер, що є цікавими або значущими для вивчення конкретних властивостей графа.

1.3. Способи задання графів

Найпростіший та найзрозуміліший спосіб представлення графів для людей полягає у використанні рисунка на площині, де вершини графа відображаються як точки, а ребра - як лінії, що з'єднують ці точки. Однак цей метод не є практичним для комп'ютерного розв'язання задач з графами [1].

Існують інші способи представлення графів, які можуть бути використані для обробки на комп'ютері. Один з таких способів - використання матриці, де кожен елемент матриці може бути 0 або 1, і вона відома як булева матриця.

Матриця інцидентності є способом задання графа, де G = (V, E) є простим графом з множиною вершин V = {v1, v2, ..., vn} і множиною ребер E = {e1, e2, ..., em}. Ця матриця, позначена як M, є булевою матрицею розміром n x m, де кожен елемент mij (i = 1, ..., n, j = 1, ..., m) задається наступним чином: mij = 1, якщо вершина vi і ребро ej є інцидентними, і 0 у протилежному випадку.

Матриця інцидентності може бути використана для представлення мультиграфів, де можуть зустрічатися однакові стовпці (що відповідають кратним ребрам). У випадку псевдографів, елементи матриці, що відповідають петлям, дорівнюють 2 (у такому випадку матриця інцидентності не є булевою).

Також матриця інцидентності може бути використана для орієнтованих графів, але в цьому випадку вона не є булевою. Для орієнтованого графа G = (V, E) з множиною вершин V = {v1, v2, ..., vn} і множиною дуг E = {e1, e2, ..., em}, матриця інцидентності, позначена як M, є nm матрицею, де елементи mij (i = 1, ..., n, j = 1, ..., m) задаються наступним чином: mij = 1, якщо дуга ej виходить з вершини vi, -1, якщо дуга ej входить до вершини vi, 2, якщо дуга ej є петлею в вершині vi, і 0 в інших випадках.

Однак з алгоритмічної точки зору матриця інцидентності є неефективним способом зображення графа. По-перше, цей метод вимагає багато пам'яті, займаючи багато комірок, більшість з яких містять нулі. По-друге, доступ до інформації у матриці інцидентності є незручним [1, 4].

Крім того, матриця інцидентності не зберігає структурну інформацію про граф, так як всі ребра та вершини представлені у вигляді числових індексів. Це може ускладнити аналіз та редагування графа, особливо при великих розмірах. Відновлення початкового зображення графа за його матрицею інцидентності також може бути складним завданням.

У порівнянні з іншими способами представлення графів, такими як списки суміжностей або матриці суміжностей, матриця інцидентності має свої обмеження та недоліки. Проте, в деяких випадках вона все ще застосовується, особливо при вивченні теоретичних властивостей графів та деяких алгоритмах їх обробки.

У практичних задачах розроблено багато інших способів представлення графів, що краще відповідають конкретним вимогам та завданням. Залежно від контексту і застосування, можуть використовуватись інші структури даних, які ефективніше враховують характеристики та

взаємозв'язки в графі.

Матрицю суміжності є одним із способів представлення графів. Для простого графа G з множиною вершин V і множиною ребер E, матриця суміжності A є булевою квадратною матрицею розміром n x n, де n є кількістю вершин. Елемент aij матриці A дорівнює 1, якщо ребро {vi, vj} належить множині E, і 0 у протилежному випадку.

У неорієнтованому графі, матриця суміжності є симетричною, тобто aij = aji. Також, в матриці суміжності простого графа, який не містить петель, елементи на діагоналі aii дорівнюють 0 (i = 1, ..., n).

Матриця суміжності також може бути використана для представлення псевдографа, де елемент aij показує кількість ребер, що з'єднують вершини vi та vj. Петлі у вершині vi позначаються значенням діагонального елемента aii = 1.

У випадку орієнтованого графа, матриця суміжності є булевою nxn матрицею A, де aij = 1, якщо існує ребро {vi, vj} у множині E, і 0 у протилежному випадку. Варто зазначити, що матриця суміжності орієнтованого графа взагалі не є симетричною.

Матриця суміжності також може бути використана для представлення орієнтованого мультиграфа, де елемент aij відповідає кількості дуг, для яких вершина vi є початковою, а вершина vj є кінцевою.

Матриця суміжності дозволяє швидкий доступ до інформації про наявність ребра (дуги) між вершинами vi і vj. Однак, недоліком є те, що обсяг пам'яті, який займає матриця суміжності, залежить від кількості вершин і становить n^2 комірок.

Крім матриці суміжності, існують інші способи представлення графів у пам'яті комп'ютера, такі як списки суміжностей та інші.

Ще одним способом представлення графів у пам'яті комп'ютера є використання списків суміжностей. Кожна вершина графа має свій список, в якому зберігаються вершини, з якими вона має пряме ребро. Цей спосіб ефективний у випадках, коли граф є розрідженим, тобто має невелику

кількість ребер порівняно з кількістю вершин.

У списку суміжностей кожен елемент списку вказує на вершину, з якою він має пряме ребро. Це дозволяє швидко отримувати інформацію про сусідні вершини кожної конкретної вершини.

Існують також інші способи представлення графів, такі як матриця ваг або список ребер. Кожен з цих способів має свої переваги та недоліки і використовується в залежності від конкретних потреб і характеристик графа.

Вибір способу представлення графа у пам'яті комп'ютера залежить від розміру графа, типу операцій, які необхідно виконати над графом, а також від ефективності використання пам'яті та швидкодії операцій. Кожен спосіб має свої особливості, і вибір залежить від конкретних умов і вимог проекту або завдання, що вирішується.

Один з економніших способів задати граф - це використання **списку пар (або списку ребер)**. Цей підхід особливо ефективний щодо використання пам'яті, особливо коли кількість ребер (позначимо як m) значно менша за n^2, де n - кількість вершин.

У цьому способі граф задається списком пар, які відповідають його ребрам або дугам. Кожна пара [u, v] відповідає ребру {u, v} для неорієнтованого графа або дузі (u, v) для орієнтованого графа.

Очевидно, що обсяг пам'яті для задання графа списком пар становить 2n (де n - кількість ребер або дуг). Це є найекономнішим підходом щодо використання пам'яті.

Однак, недоліком цього підходу є велика кількість кроків для знаходження множини вершин, до яких є ребра або дуги, з заданої вершини. Але цю ситуацію можна вдосконалити, якщо відсортувати список пар лексикографічно і використати двійковий пошук для швидкого доступу до потрібних даних [1].

Ще однією перевагою способу задання графа списком пар є його гнучкість. Додавання або видалення ребра може бути здійснено швидко, просто за допомогою операцій додавання або видалення пари зі списку. Це

зручно, особливо якщо граф підлягає частим змінам.

Крім того, список пар надає інтуїтивний спосіб представлення зв'язків у графі. Кожен елемент списку містить дві вершини, які з'єднані ребром або дугою. Це дозволяє легко переглядати та аналізувати структуру графа.

Узагальнюючи, задання графа списком пар є економним способом, особливо коли кількість ребер або дуг невелика порівняно з квадратом кількості вершин. Він має гнучку структуру та легкість у зміні, а також надає зрозуміле представлення зв'язків у графі.

Зображення графа списками суміжності базується на описаному способі задання графа. Цей метод використовує масив списків, який називається Adj, розміром $|V| = n$, де кожен список відповідає одній вершині.

У зображенні графа списками суміжності, для кожної вершини $u \in V$, список Adj[u] містить вказівники (або посилання) на всі вершини, з якими u зв'язана у множині Г(u). Вказівники у списку можуть бути в довільному порядку.

Цей метод дозволяє ефективно представити зв'язки між вершинами графа із використанням мінімального обсягу пам'яті. Кожен список Adj[u] містить лише необхідну кількість вказівників, що вказують на сусідні вершини.

Зображення графа списками суміжності дозволяє швидко отримувати інформацію про зв'язки вершин. За допомогою одного кроку можна переглянути всі сусідні вершини для будь-якої заданої вершини.

Узагальнюючи, зображення графа списками суміжності є ефективним способом представлення зв'язків у графі. Використання масиву списків дозволяє зберігати і отримувати інформацію про сусідні вершини з мінімальним обсягом пам'яті та операцій.

Зображення графа списками суміжності має декілька переваг і особливостей. Одна з переваг полягає в тому, що обсяг пам'яті, необхідний для збереження графа, залежить від кількості ребер, а не від загальної кількості можливих ребер. Це особливо корисно в разі розріджених графів, де

кількість ребер значно менша, ніж кількість можливих ребер.

Крім того, зображення графа списками суміжності сприяє ефективним операціям знаходження сусідніх вершин. Для заданої вершини u, список Adj[u] містить усі вершини, з якими u має зв'язок. Це дозволяє швидко переглянути всіх сусідів вершини без необхідності перебирати всі ребра графа.

Недоліком зображення графа списками суміжності є велика кількість кроків для знаходження сусідів заданої вершини, якщо відсутній впорядок у списку Adj[u]. Однак, це можна покращити, впорядковуючи списки лексикографічно і використовуючи алгоритм двійкового пошуку для ефективного пошуку вершин.

Зображення графа списками суміжності є одним із багатьох способів представлення графа в пам'яті комп'ютера. Вибір конкретного методу залежить від вимог до швидкодії, обсягу пам'яті та типу операцій, які потрібно виконувати з графом. Кожен метод має свої переваги і обмеження, і вибір підходячого залежить від конкретних вимог і умов використання.

1.4. Шляхи та цикли. Зв'язність

Шляхом довжини r від вершини u до вершини v у неорієнтованому графі є послідовність ребер виду e1 = {x0, x1}, e2 = {x1, x2}, ..., er = {xr−1, xr}, де x0 = u, xr = v, а r є натуральним числом. У такому шляху ребра враховують стільки разів, скільки вони зустрічаються на шляху. Вершини u і v є крайніми, а решта вершин на шляху є внутрішніми [4].

Циклом у неорієнтованому графі називається шлях, в якому початкова і кінцева вершини співпадають, тобто u = v.

У випадку простого графа шлях можна задати послідовністю вершин, через які він проходить: x0, x1, ..., xr-1, xr. Шлях або цикл називають простим, якщо не містить повторюючихся ребер.

Розрізом графа є пара множин вершин, які не перетинаються, а їх

об'єднання утворює вихідний граф. В транспортній мережі джерело та стік належать різним множинам. Величина розрізу визначається сумою ваг ребер, які мають рівно один кінець у даному розрізі.

Шлях з крайніми вершинами u та v називається з'єднуючим шляхом між вершинами u та v.

Неорієнтований граф є зв'язним, якщо будь-які дві вершини з'єднані шляхом. Якщо граф не є зв'язним, то його називають незв'язним. Незв'язний граф складається з двох або більше зв'язних підграфів, кожна пара яких не має спільних вершин. Ці зв'язні підграфи називають компонентами зв'язності або просто компонентами графа [1]. Відстанню d(u, v) між різними вершинами u та v називається довжина найкоротшого шляху (u, v), а сам найкоротший шлях має назву геодезичного шляху. Очевидно, що найкоротший шлях не містить повторюючихся вершин (і ребер).

З цього випливає теорема: Між кожною парою різних вершин зв'язного графа існує простий шлях.

Для орієнтованого графа вводять поняття орієнтованого шляху (або простого шляху) від вершини u до вершини v, який є скінченною послідовністю дуг виду e1 = {x0, x1}, e2 = {x1, x2}, ..., er = {xr−1, xr}, де x0 = u, xr = v, а r є натуральним числом. Вершини u та v називаються крайніми, а решта вершин на шляху – внутрішніми. Кількість дуг, які складають шлях, називається його довжиною. Орієнтований шлях або цикл називають простими, якщо жодна дуга не входить у них понад один раз.

Граф називається дводольним, якщо множину його вершин V можна розбити на дві неперетинаючіся підмножини ($V = V1 \cup V2$, $V1 \cap V2 = \emptyset$), таким чином, що кожне ребро з'єднує вершину з V1 та вершину з V2.

Дводольний граф називається повним, якщо кожна вершина з V1 з'єднана з усіма вершинами, що належать до V2.

Зв'язний граф називається таким, що між будь-якими двома його вершинами існує шлях. Іншими словами, в зв'язному графі можна знайти шлях від однієї вершини до будь-якої іншої вершини. Якщо граф не є

зв'язним, то його називають незв'язним. Незв'язний граф складається з двох або більше зв'язних підграфів, причому кожна пара підграфів не має спільних вершин. Ці зв'язні підграфи називаються компонентами зв'язності або просто компонентами графа.

Розріз графа - це пара множин вершин, які є неперетинаючими, а їх об'єднання утворює початковий граф. У випадку транспортної мережі, джерело та стік належать різним множинам в розрізі. Величиною розрізу називається сума ваг ребер, у яких рівно один кінець належить даному розрізу.

Таким чином, шляхи, цикли, зв'язність, дводольність та розрізи є важливими поняттями теорії графів, які допомагають аналізувати та вивчати структуру графів та їх взаємозв'язки.

РОЗДІЛ 2. ОСНОВНІ АЛГОРИТМИ

2.1 Мережі та алгоритми пошуку максимального потоку у них

Задача знаходження максимального потоку в мережі є класичною задачею оптимізації, яка залишається актуальною й сьогодні. Методи розв'язання цієї задачі застосовуються у транспортних, комунікаційних мережах, при моделюванні хімічних, фізичних процесів та в інших ситуаціях, де потрібно оптимізувати потік через мережу вузлів. Часто в таких задачах на графах потрібно враховувати додаткову інформацію, наприклад, фактичну відстань між окремими пунктами, вартість проїзду або час проїзду. Для цього використовують поняття "зваженого графу" та "мережі".

Зваженим називається простий граф, у якого кожному ребру e приписано дійсне число w(e), яке називається вагою ребра. Аналогічно, визначається зважений орієнтований граф, де кожній дузі e приписується вага w(e) - дійсне число.

Для зображення зваженого графа G = (V, E) в пам'яті комп'ютера існують різні підходи. Нехай |V| = n, |E| = m.

Існують п’ять способи представлення зваженого графа:

Перший спосіб - задання графа матрицею ваг W, яка подібна до матриці суміжності. У цій матриці елемент wij = w(vi, vj), якщо ребро {vi, vj} належить E (у разі орієнтованого графа, дуга {vi, vj} належить E). Якщо ребро {vi, vj} (або дуга {vi, vj}) відсутнє, то wij = 0 або wij = ∞, залежно від задачі, яку потрібно вирішити.

Другий спосіб - задання графа списком ребер. Для зваженого графа, кожному елементу списку E відповідають три комірки - дві для ребра та одна для його ваги. Таким чином, загальна кількість комірок дорівнює 3 * |E|.

Третій спосіб - задання графа списками суміжності. У випадку зваженого графа, кожен список Adj[u] містить вказівники на всі вершини v, які належать множині Г(u), а також числа w(u, v), що представляють ваги ребер.

Четвертий спосіб - задання графа матрицею суміжності, де кожен

елемент aij відповідає наявності ребра між вершинами vi та vj. Якщо ребро існує, то значення aij відповідає вазі цього ребра. Якщо ребра немає, то можна встановити значення aij = 0 або велике число, щоб вказати відсутність зв'язку між вершинами.

П'ятий спосіб - задання графа виглядом графічного представлення, де вершини позначені точками, а ребра - стрілками або лініями, з вагами ребер позначеними поруч. Цей спосіб часто використовується для візуалізації зваженого графа та демонстрації його структури.

Всі ці способи представлення зваженого графа мають свої переваги та використовуються в залежності від конкретного контексту та задачі, яку необхідно вирішити. Важливо вибрати оптимальний спосіб представлення для ефективної обробки та аналізу графа.

Довжина шляху в зваженому графі визначається як сума ваг ребер або дуг, що складають цей шлях. Якщо граф не має ваг, то довжину шляху розглядають як кількість ребер або дуг у ньому.

Мережею називається орієнтований граф G = (V, E), де кожне ребро (u, v) має додатню пропускну здатність c(u, v) > 0. Якщо (u, v) не належить до E, то вважається, що c(u, v) = 0. У транспортній мережі виділяються дві вершини - джерело (позначається s) та стік (позначається t).

Функція потоку f:E→R в мережі задовольняє певні умови: (1) антисиметричність: f(u, v) = -f(v, u); (2) обмеженість: f(u, v) ≤ c(u, v), тобто потік уздовж дуги не може перевищувати пропускну здатність ребра; (3) відсутність потоку на відсутніх дугах: якщо (u, v) не існує, то f(u, v) = 0; (4) закон збереження потоку: для кожного вузла, який не є джерелом або стоком, сума потоків вхідних дуг дорівнює сумі потоків вихідних дуг.

Максимальний потік в мережі відноситься до потоку, який може бути направлений з джерела s до стоку t. Значення f(u, v) можна розглядати як об'єм рідини, яка протікає вздовж дуги (u, v).

Максимальний потік в мережі відображає максимальну кількість

ресурсів, таких як рідини, енергія або інформація, яку можна перенести з джерела до стоку. Це показник ефективності транспортної системи або потокової мережі.

У задачі знаходження максимального потоку використовуються різні алгоритми, такі як алгоритм Форда-Фалкерсона, алгоритм Едмондса-Карпа та алгоритм Диніця. Ці алгоритми шукають оптимальний шлях для перенесення максимального потоку в мережі.

Задача знаходження максимального потоку має широкі застосування у різних галузях, включаючи транспортну логістику, маршрутизацію даних у комп'ютерних мережах, електричні мережі та економіку. Вирішення цієї задачі дозволяє оптимізувати розподіл ресурсів і планування в умовах обмежень.

Розріз в контексті даної задачі означає розділення множини вершин V на дві підмножини A і B таким чином, що вершина s належить до A, вершина t належить до B, і множини A і B не мають спільних елементів.

Залишкова мережа Gf, яка створюється для заданої мережі G = (V, E) та потоку f, складається з ребер $Ef = \{(u, v) \in V \times V \mid c(u, v) - f(u, v) > 0\}$. Тут c(u, v) вказує пропускну здатність ребра, а f(u, v) вказує величину потоку, що протікає через це ребро. Від'ємні значення потоку для дуг не вказуються, оскільки їх можна легко вивести з властивості антисиметрії потоку (f(u, v) = -f(v, u)). На графі позначені повністю насичені ребра та ребра з нульовим значенням потоку для зручності. Для кожної вершини, яка не є джерелом або стоком, сума потоків вхідних ребер дорівнює сумі потоків вихідних ребер, і величина потоку через кожне ребро ніколи не перевищує його пропускної здатності.

Мережний розріз (cut) визначається як поділ множини вершин V на дві підмножини S і T таким чином, що джерело s належить S, стік t належить T, і всі ребра, які йдуть від вершини S до вершини T, перетинають розріз. Розріз відокремлює джерело від стоку в мережі.

Мінімальний розріз (min-cut) в мережі визначається як розріз, для якого

сумарна пропускна здатність ребер, що перетинають розріз, є мінімальною серед усіх можливих розрізів. Мінімальний розріз представляє обмеження найбільшої кількості потоку, який може пройти від джерела до стоку в мережі.

2.2. Алгоритм Едмондса-Карпа

Алгоритм Едмондса-Карпа є одним з популярних алгоритмів для знаходження максимального потоку у мережі. Цей алгоритм є модифікацією алгоритму Форда-Фалкерсона та використовує пошук у ширину для знаходження найкоротших шляхів у решітчній мережі.

Процес роботи алгоритму Едмондса-Карпа полягає в наступних кроках:

1) Ініціалізація: Початково всі ребра у мережі мають нульовий потік. Також створюється резидуальна мережа, яка використовується для визначення можливих збільшень потоку.

2) Пошук шляху: Використовуючи пошук у ширину (BFS) у резидуальній мережі, знаходяться усі можливі шляхи від джерела до стоку.

3) Оновлення потоку: Знаходиться мінімальна пропускна здатність ребра на знайденому шляху. Ця величина використовується для збільшення потоку на цьому шляху.

4) Оновлення резидуальної мережі: Після збільшення потоку на шляху оновлюються ваги ребер у резидуальній мережі. Для кожного ребра встановлюється нова вага, яка відображає розмір доступного ще потоку через це ребро.

5) Повторення: Кроки 2-4 повторюються до тих пір, поки більше немає шляхів з джерела до стоку у резидуальній мережі.

6) Завершення: Коли більше немає шляхів у резидуальній мережі, алгоритм Едмондса-Карпа завершується, і максимальний потік визначений.

Один з ключових аспектів алгоритму Едмондса-Карпа - використання пошуку у ширину. Це дозволяє знаходити найкоротші шляхи у решітчній

мережі, тобто шляхи, які потребують мінімальної кількості ребер. Такий підхід сприяє більш ефективному збільшенню потоку, оскільки знаходяться шляхи, які можуть мати максимальні збільшення потоку.

Алгоритм Едмондса-Карпа має часову складність O(V * E^2), де V - кількість вершин у мережі, а E - кількість ребер. Хоча цей алгоритм не є найшвидшим для великих мереж з великою кількістю ребер, він є простим у реалізації та забезпечує коректні результати для багатьох випадків задач оптимізації потоку.

2.3. Алгоритм Форда-Фалкерсона

Теорема Форда-Фалкерсона стверджує, що величина потоку в графі дорівнює величині пропускної здатності його мінімального розрізу, який розділяє джерело і стік.

Доказ: Будь-який потік між вершинами s та t не може перевищувати величину будь-якого розрізу. Розглянемо потік і розріз. Величина потоку складається з суми "вантажу", що протікає по всіх можливих шляхах від вершини s до t. Кожен з таких шляхів має спільне ребро з розрізом. Оскільки потік через кожне ребро не може перевищувати його пропускної здатності, загальна величина потоку менша або рівна загальній пропускній здатності розрізу.

Алгоритм Форда-Фалкерсона починає з нульового потоку і на кожній ітерації збільшує його значення в мережі. На кожному кроці визначається величина, на яку може збільшитися потік уздовж дуги на поточному ланцюзі. Потік по даному ланцюзі збільшується до максимально можливого значення, поки ланцюг не стає насиченим.

Початкове значення загальної величини потоку в мережі приймається рівним нулю.

На кожній ітерації алгоритму використовується техніка "міток". Починаючи з джерела, ми позначаємо його як "відвідане" і знаходимо всі

сусідні вершини, позначаючи їх як "видимі". Якщо ребра, які з'єднані з видимою і відвіданою вершиною, ще не насичені (тобто величина потоку менша за пропускну здатність), ми переходимо до однієї з видимих вершин. Ми позначаємо цю вершину як "відвідану" і визначаємо максимальну величину потоку, яка може пройти через ребро. Ця величина обчислюється як мінімальна серед пропускних здатностей ребра, яким ми досягли поточної вершини, та величини потоку, що досягла вершини, з якої ми прийшли. Ми продовжуємо цей процес від джерела до стоку. Загальну величину потоку для цього доповнюючого шляху збільшуємо на кількість "рідини", яка досягла стоку. Якщо на певному кроці сток не досягнутий і видимих вершин більше немає, ми можемо перерозподілити потік, використовуючи фіктивне ребро, величина потоку якого відповідає від'ємному значенню частини потоку, яку ми плануємо повернути.

Якщо у послідовності побачених вершин немає стоку і більше немає видимих вершин, максимальна величина потоку в мережі визначається розміром потоку, який досяг стоку, і алгоритм закінчує свою роботу.

Якщо мережа не має єдиного джерела або стоку, ми додаємо одну вершину-джерело і одну вершину-сток, які з'єднується з усіма джерелами і стоками за допомогою ребер з необмеженою пропускною здатністю.

Оскільки ми можемо виключати по одному ребру за кожен прохід мережею, і це повторюється для кожної вершини на кожній ітерації, у найгіршому випадку ми переглядаємо всі ребра, що призводить до загальної складності методу O(nm^2), де n - кількість вершин, а m - кількість ребер у мережі [4].

Після кожного збільшення потоку на доповнюючому шляху, оновлюється залишкова мережа Gf. За кожною доповнюючою шляху ребра в Gf можуть змінювати свою пропускну здатність відповідно до залишкової пропускної здатності. Якщо у ребрі (u, v) є пропускна здатність c(u, v) і вже протікає потік f(u, v), то залишкова пропускна здатність на цьому ребрі дорівнює c(u, v) - f(u, v). Якщо потік в зворотному напрямку f(v, u) > 0, то в

залишковій мережі буде присутнє ребро (v, u) з пропускною здатністю f(v, u), а залишкова пропускна здатність на цьому ребрі буде рівна f(v, u).

Процес знаходження доповнюючих шляхів та оновлення залишкової мережі продовжується до тих пір, поки не буде можливості знайти новий доповнюючий шлях у залишковій мережі Gf. Коли більше немає доповнюючих шляхів, отримана мережа Gf є максимально насиченою і відповідає максимальному потоку вихідної мережі G.

Значення максимального потоку можна визначити, обчисливши суму потоків, які виходять з джерела. Використовуючи мережу G та отриманий максимальний поток f, можна також відновити розподіл потоку по ребрах мережі та визначити значення потоку вздовж кожного ребра.

Алгоритм Форда-Фалкерсона забезпечує знаходження максимального потоку в мережі G за допомогою пошуку доповнюючих шляхів та оновлення залишкової мережі. Він є ефективним і коректним методом розв'язання задачі максимального потоку.

2.4. Метод проштовхування передпотоку

Алгоритм, який був опублікований у 1986 році Голдбергом і Тар'яном, вимагає введення певних означень перед його описом.

Почнемо з означення **передпотоку**, яке є функцією $fp: V \times V \rightarrow R$ і має такі властивості:

1) Антисиметричність: $fp(u, v) = -fp(v, u)$.
2) Обмеження пропускною здатністю: $fp(u, v) \leq c(u, v)$, де $c(u, v)$ - пропускна здатність ребра (u, v).
3) Ослаблена умова збереження потоку: Для кожної вершини u, що не є джерелом або стоком, сума всіх вхідних потоків $fp(v, u)$ (для всіх v, що належать до V і не дорівнюють ні s, ні t) не менша за нуль.

Ці означення визначають передпотік, який є проміжним кроком у виконанні алгоритму.

Надлишковим потоком, який входить у вершину u, ми називаємо суму значень передпотоку fp(v, u) для всіх вершин v, що належать до V. Таким чином, вершина u (яка не є ні джерелом, ні стоком) називається переповненою, якщо значення надлишкового потоку e(u) більше нуля.

Функція h: V → Z+ називається висотою вершини, якщо вона відповідає таким умовам:

1) h(s) = |V|, де s - джерело.
2) h(t) = 0, де t - сток.
3) Для кожного ребра (u, v) ∈ E: h(u) ≤ h(v) + 1.

Уявімо мережу з визначеною для кожного ребра пропускною здатністю c та виділеними вершинами: джерелом s та стоком t. Можна уявити цю мережу як систему резервуарів, розташованих на різних висотах і з'єднаних трубами з заданими діаметрами. Сам алгоритм подібний до процесу перекачування рідини (операції протікання) з одного резервуара в інші, які знаходяться на нижчій висоті, до тих пір, поки не будуть заповнені резервуари, що відповідають переповненим вершинам. Іноді може статися так, що резервуари, що сусідять з поточною переповненою вершиною u, знаходяться на такій же висоті, як u або навіть вище. В такому випадку застосовується операція підйому, щоб підняти висоту вершини на одну умовну одиницю вище, ніж найнижчий сусідній резервуар. Тоді буде існувати хоча б один шлях, по якому можна перенести рідину з поточної вершини.

Проштовхування: ця операція застосовується, коли у вершини u є надлишковий потік (e(u) > 0), існує пропускна здатність на ребрі (cf(u, v) > 0), і висота вершини u дорівнює h(v) + 1. В цьому випадку максимально можливий потік проходить через ребро (u, v), рівний мінімуму між надлишковим потоком в вершині u та залишковою пропускною здатністю ребра (u, v). Це призводить до зменшення надлишкового потоку в вершині u і збільшення потоку через ребро (u, v).

Підйом: ця операція застосовується до вершини u, якщо у неї є

надлишковий потік (e(u) > 0), і для всіх суміжних ребер (u, v) виконується умова h(u) ≤ h(v). Висота вершини u підвищується на одиницю більше, ніж найнижча висота серед усіх суміжних вершин (u, v) таких, що fp(u, v) - c(u, v) < 0. Це дозволяє знайти шлях, по якому можна перенести рідину з переповненої вершини u.

Ініціалізація: для кожного ребра, яке є інцидентним до джерела, встановлюємо максимальний потік, тим самим збільшуючи надлишковий потік для суміжних вершин. Для всіх інших вершин надлишковий потік дорівнює нулю. Крім того, для всіх вершин, крім джерела, встановлюємо висоту рівною нулю.

$f_p(u, v) = \{c(u, v), \quad u = s \; -c(v, u), \qquad v = s \; 0, \quad u \neq s$ та $v \neq s$

$$h(u) = \{|V|, \; u = s \; 0, \; u \neq s$$

Поки існує вершина з надлишком, обираємо одну з них.

Якщо існує суміжна вершина з меншою висотою, застосовуємо операцію проштовхування. Це означає, що потік переноситься через ребро до вершини з меншою висотою, зменшуючи надлишковий потік в поточній вершині та збільшуючи потік через ребро.

У протилежному випадку, якщо не існує суміжної вершини з меншою висотою, застосовуємо операцію підйому. Висота поточної вершини збільшується на одиницю більше, ніж найнижча висота серед всіх суміжних вершин, для яких є залишок у пропускній здатності. Це дозволяє знайти шлях для перенесення надлишкового потоку з поточної вершини.

Повторюємо кроки 2-4 до тих пір, поки існують вершини з надлишком.

Алгоритм завершується, коли жодна вершина не має надлишкового потоку.

Це є загальна структура алгоритму Форда-Фалкерсона.

Доведення коректності алгоритму можна показати за допомогою наступних допоміжних лем:

Лема 1: Під час виконання алгоритму жодна властивість висоти не порушується.

Лема 2: Нехай f є передпотоком в мережі. Тоді для будь-якої переповненої вершини можна виконати операцію проштовхування або підйому.

Доведення: Розглянемо вершину u, для якої e(u) > 0. Для будь-якого ребра (u, v) з множини ребер, виконується h(u) ≤ h(v) + 1 з властивості функції висоти. Якщо до вершини u застосовна операція проштовхування, то лема доведена.

В іншому випадку, для всіх існуючих ребер (u, v) справедлива нерівність h(u) < h(v) + 1. Звідси випливає, що h(u) ≤ h(v). Таким чином, дана вершина задовольняє вимоги для операції підйому і до неї можна застосувати цю операцію.

Теорема про простий шлях стверджує, що якщо існує шлях між двома вершинами графа, то існує вершинно-простий шлях між ними, в якому жодна з вершин не повторюється.

Лема 3: стверджує, що у залишковій мережі Gf, яка виникає в результаті застосування алгоритму, не існує шляху від джерела s до стоку t.

Доведення: Для доведення протилежного твердження, припустимо, що в Gf існує шлях від s до t. З теореми про існування простого шляху випливає, що в Gf існує простий шлях p = ⟨v0, v1, ..., vk⟩, де v0 = s і vk = t. Оскільки p є простим шляхом, то k < |V|. З властивості функції висоти h(vi) ≤ h(vi+1) + 1 для всіх i ∈ {0, 1, ..., k-1}. З цього випливає, що h(s) ≤ h(t) + k. Оскільки h(t) = 0, то h(s) ≤ k < |V|, що суперечить умові h(s) = |V|.

Теорема стверджує, що якщо алгоритм завершується, то знайдений передпотік є максимальним потоком.

Доведення: Покажемо, що перед кожною перевіркою умови в циклі f є потоком. Перед циклом, після ініціалізації, f є передпотоком. У циклі виконуються лише операції проштовхування та підйому, які не змінюють

величину потоку, а лише змінюють висоту вершини. Операція проштовхування застосовується лише тоді, коли існує надлишковий потік у вершині u, збільшуючи потік через ребро (u, v) на величину, яка не перевищує його пропускну здатність. Таким чином, якщо перед виконанням операції проштовхування f був передпотоком, то й після неї f залишається передпотоком. Після закінчення роботи алгоритму, для кожної вершини u, яка не є ні джерелом, ні стоком, виконується рівність e(u) = 0, що випливає з лем (1), (2) і з того, що перед виконанням операцій проштовхування та підйому f є передпотоком. Таким чином, f задовольняє умову збереження потоку, отже f є потоком.

Оскільки з леми (1) функція h є функцією висоти і після роботи алгоритму в залишковій мережі Gf немає шляху від s до t (з леми 3), то з теореми Форда-Фалкерсона f є максимальним потоком.

Метод проштовхування передпотоку є ефективним алгоритмом для знаходження максимального потоку в мережі. Його основна ідея полягає у використанні вершинної ізоляції, висоти вершин і переповнених вершин для пошуку шляхів з джерела до стоку та збільшення потоку через ребра.

Алгоритм починається з ініціалізації, де кожному ребру присвоюється передпотік нульового значення, надлишковий потік для вершин, які суміжні з джерелом, встановлюється на максимальне значення, а для усіх інших вершин встановлюється нульова висота.

Потім алгоритм переходить до основного циклу, де обирається вершина з надлишковим потоком. Якщо існує суміжна вершина з меншою висотою, застосовується операція проштовхування, яка збільшує потік через ребро, що з'єднує ці вершини, на максимум з надлишковим потоком у початковій вершині та вільною пропускною здатністю ребра. В іншому випадку виконується операція підйому, яка змінює висоту поточної вершини на мінімальне значення серед суміжних вершин, підвищуючи її на одиницю.

Цей процес повторюється до тих пір, поки не буде існувати жодної вершини з надлишковим потоком. Після завершення алгоритму знайдений

передпотік є максимальним потоком у мережі.

Алгоритм проштовхування передпотоку має часову складність O(V^3), де V - кількість вершин у мережі. Ітерації циклу проштовхування передпотоку залежать від висоти вершин, і у худшому випадку можуть зайняти до O(V^2) кроків. Однак, застосування різних оптимізаційних методів може покращити швидкість алгоритму у практичних застосуваннях.

2.5. Алгоритм Дініца.

Алгоритм, запропонований ізраїльським вченим Юхимом Дініцем у 1970 році, має часову складність O(V^2E). Основна ідея цього алгоритму полягає в збільшенні потоку одночасно вздовж усіх найкоротших ланцюгів певної довжини. Для цього на кожному кроці алгоритму будується допоміжна мережа, відома як "мережа шарів" або "мережа рівнів". Ця мережа містить всі збільшуючі ланцюги, довжина яких не перевищує найкоротшого шляху від джерела до стоку.

Побудова мережі шарів здійснюється за допомогою алгоритму обходу в ширину, з урахуванням того, що насичені дуги та дуги, що мають кінцеву вершину на рівні, не більшому за рівень початкової вершини, ігноруються. Вершини мережі шарів нумеруються залежно від номеру вхідного ребра у знайденому ланцюгу. Отримана допоміжна мережа міститиме лише ребра, які починаються у вершині з рівня L і закінчуються у вершині з рівня L + 1, а також ненасичені ребра.

Якщо мережа шарів досягає стоку, алгоритм завершується. У протилежному випадку, починаючи з джерела мережі шарів, виконується пошук в глибину до досягнення стоку. Після досягнення стоку знаходиться дуга з найменшою залишковою пропускною здатністю, і потік в мережі збільшується на це значення. Пошук в глибину повторюється, поки стік досяжний. Затім граф рівнів оновлюється з новими значеннями потоків, і процес починається знову - будується нова мережа шарів.

Отже, схема алгоритму виглядає наступним чином:

1) Ініціалізуємо потік f(u, v) = 0 для кожного ребра (u, v).
2) Будуємо допоміжну мережу. Якщо досягнення стоку неможливе, алгоритм завершується і повертає потік f.
3) Знаходимо блокуючий потік f'.
4) Доповнюємо потік f приростиом f' і переходимо до кроку 2.
5) Повторюємо кроки 2-4 до тих пір, поки в допоміжній мережі шарів не вдасться знайти блокуючий потік.
6) Коли в допоміжній мережі шарів не можна знайти блокуючий потік, алгоритм завершується і повертає потік f.
7) Отриманий потік f є максимальним потоком у вихідній мережі.

Доведемо, що якщо алгоритм завершується, то отриманий потік є максимальним. Припустимо, що в допоміжній мережі шарів не вдається знайти блокуючий потік. Отже, в цій мережі неможливо досягнути стоку з джерела. Але, оскільки мережа містить всі найкоротші шляхи від джерела до стоку, то в залишковій мережі Gf немає шляху з джерела до стоку. Згідно з теоремою Форда-Фалкерсона, отримуємо, що поточне значення потоку є максимальним.

В кожній ітерації алгоритму Дініца, блокуючий потік f' знаходиться шляхом обходу в ширину в допоміжній мережі шарів. Цей потік додається до потоку f і доповнює його. Алгоритм продовжується до тих пір, поки не буде можливо знайти блокуючий потік. Коли в допоміжній мережі шарів неможливо знайти блокуючий потік, це означає, що не можна досягнути стоку з джерела, і потік f є максимальним потоком.

Загальна часова складність алгоритму Дініца складає O(V^2E), де V - кількість вершин у графі, а E - кількість ребер. Це робить його ефективним для багатьох практичних задач зі стрімкими мережами.

2.6. Потік мінімальної вартості

Задача мінімального потоку вартості (min-cost flow) в транспортній мережі є задачею оптимізації, яка полягає в пошуку найбільш економічного шляху для переміщення певного об'єму K потоку. Ця задача є розширенням задачі максимального потоку, в якій додатково враховуються витрати на перенесення одиниці потоку через кожне ребро мережі та встановлюється чітке обмеження на величину потоку, яку джерело та стік можуть випустити та прийняти відповідно.

В розглянутому випадку орієнтованого графу, де між будь-якою парою вершин існує не більше одного ребра, визначається пропускна здатність Uij ребра (i, j), ціна Cij за одиницю потоку вздовж цього ребра та величина потоку Fij на ребрі (i, j), які спочатку всі приймають значення нуль. Для побудови залишкової мережі додається зворотне ребро (j, i) з нульовою пропускною здатністю Uji = 0 та вартістю Cji = -Cij. У процесі виконання алгоритму враховується умова Fij = -Fji. Залишкова мережа включає лише ненасичені ребра, а пропускну здатність кожного ребра вираховується як різниця пропускної здатності відповідного ребра в початковій мережі та величини потоку Uij - Fij.

Алгоритм знаходження потоку мінімальної вартості працює за таким принципом: на кожній ітерації алгоритму від джерела S до стоку T шукаємо найкоротший шлях в залишковій мережі, враховуючи вартість Cij ребер. Якщо шлях не знайдений, то алгоритм завершується, а знайдений потік F є рішенням. В іншому випадку збільшуємо потік вздовж знайденого шляху на максимально можливе значення, враховуючи дугу з найменшою залишковою пропускною здатністю. Алгоритм продовжується досягненням обмеження потоку K, якщо воно задано. Зазначений алгоритм також може вирішити задачу пошуку найбільш економічного потоку максимального значення, якщо встановити K=∞.

Додатково, в алгоритмі потоку мінімальної вартості використовується поняття дуальності. Дуальна до задачі потоку мінімальної вартості є задача

про пошук найкоротшого шляху в мережі з вагами на ребрах, де вага визначається вартістю одиниці потоку. Це означає, що вартість шляху в дуальній задачі відповідає величині потоку в вихідній задачі.

У процесі вирішення задачі потоку мінімальної вартості можуть виникати ситуації, коли виникають від'ємні цикли в залишковій мережі. Це може призвести до нескінченного зростання потоку та некоректних результатів. Для запобігання цьому використовуються різні підходи, наприклад, алгоритм Беллмана-Форда, який виявляє та усуває від'ємні цикли.

Задача потоку мінімальної вартості має широке застосування в різних сферах, таких як транспортні мережі, логістичне планування, розподіл ресурсів та багато інших. Шляхом знаходження оптимального потоку мінімальної вартості можна ефективно вирішувати проблеми розподілу та використання ресурсів з мінімальними витратами.

Основною метою задачі потоку мінімальної вартості є досягнення оптимального балансу між витратами та результатом, забезпечуючи найефективніше використання ресурсів та мінімізацію загальних витрат.

2.7. Мережі Петрі

Мережа Петрі є графічним і математичним інструментом для моделювання процесів та систем. Формально, вона визначається як сукупність множини N = (P, T, G, Ω), де P представляє сукупність позицій (вузлів), T - сукупність переходів, G - сукупність дуг мережі, а Ω - сукупність ваг дуг.

У мережі Петрі позиції представляють вузли, а переходи - події. Граф є орієнтованим і двохдольним, і містить мітки на вершинах. Позиції можуть бути відміченими або вільними, тоді як переходи не мають міток. Між позицією і переходом може існувати лише одна дуга, а два переходи або дві позиції не можуть бути з'єднані дугами.

Мережі Петрі були розроблені у 1962 році з метою представлення

координації подій. Вони знайшли широке застосування у моделюванні та аналізі різних процесів, таких як паралельні обчислення, диспетчеризація ресурсів, протоколи мереж і багато інших. Мережі Петрі дозволяють візуалізувати та аналізувати взаємодію компонентів системи, досліджувати їхню поведінку та виявляти можливі проблеми або недоліки.

Роботу мережі Петрі можна уявити так: вузли відповідають певним умовам, а переходи - подіям. Таким чином, стан мережі в будь-який момент часу визначається системою умов. Для зручності задання умов використовуються маркери, які позначаються крапками всередині вузлів. Виникнення певної комбінації маркерів у вузлах призводить до виникнення певної події, яка, в свою чергу, змінює стан умов мережі. Таким чином, роботу мережі Петрі можна описати наступними правилами:

Активізація: перехід активізований, якщо всі вхідні вузли цього переходу мають маркери.

Увімкнення: активізований перехід може виконатися. Усі мітки з усіх вхідних вузлів переходу зникають, і на всіх вихідних вузлах поточного переходу з'являються мітки.

Якщо в мережі одночасно активовано декілька переходів, не завжди очевидно, який з них буде виконаний першим - це властивість невизначеності.

Мережі Петрі є зручним та універсальним засобом для моделювання дискретних систем. Це проявляється, зокрема, у можливості формування моделей з потрібним рівнем деталізації. Наприклад, для задачі попереднього планування виробничого процесу може використовуватися менший рівень деталізації, ніж для безпосереднього керування обладнанням в реальному часі.

Мережі Петрі є не лише зручним і універсальним засобом для моделювання дискретних систем, але й мають додаткові переваги.

Одна з переваг Мереж Петрі полягає у їх здатності до аналізу і верифікації систем. Завдяки формальній природі Мереж Петрі, можна

проводити аналіз властивостей системи, таких як безпека, живучість, обмеженість ресурсів і інші. Використовуючи різноманітні аналітичні методи та інструменти, можна виявити можливі проблеми або вдосконалити функціональність системи.

Ще одна перевага Мереж Петрі полягає у їх здатності до моделювання паралельних та розподілених процесів. Мережі Петрі дозволяють відображати одночасність подій та взаємодію між компонентами системи. Це особливо корисно для моделювання систем, де декілька подій можуть відбуватися одночасно або де різні компоненти системи працюють паралельно.

Крім того, Мережі Петрі є інтуїтивно зрозумілими для представлення та комунікації моделей. Графічне зображення Мереж Петрі дозволяє легко візуалізувати структуру системи, її компоненти та зв'язки між ними. Це сприяє зрозумілості та співпраці між розробниками, аналітиками та зацікавленими сторонами при роботі з моделями систем.

Загалом, Мережі Петрі є потужним і гнучким інструментом для моделювання та аналізу систем. Вони дозволяють представити складні процеси та взаємодії між компонентами системи, а також провести аналіз та верифікацію їх властивостей. Це робить їх корисними для широкого спектору застосувань, від проектування програмного забезпечення до оптимізації виробничих процесів.

3. ОПИС ПРОГРАМИ

У веб-додатку реалізовано використання пошуку заданого потоку з мінімальною ціною, проштовхування передпотоку. Після виконання цих алгоритмів, результат відображається на панелі нижче. Щоб зрозуміти результати краще, повністю насичені ребра мережі позначаються червоною лінією, а порожні ребра - штриховою лінією.

Додаток розроблено з використанням фреймворку Angular, який є ідеальною платформою для створення односторінкових додатків, та мови TypeScript, яка є розширенням JavaScript.

Для відображення ребер і вершин, а також виконання операцій з ними, були створені відповідні класи AppEdge та AppNode. Клас AlgorithmService містить методи, які повертають результати виконання алгоритмів у вигляді об'єктів класу ResultFlowReturn.

Демонстрація веб-додатка:

Для запуску проекту переходимо за адресою 'http://localhost:4200/'. На сторінці знаходиться кнопка "Редагувати", розташована зліва внизу.

Рис.3.1.

Після виконання цієї дії ви побачите наявність двох кнопок - "Додати вершину" і "Додати ребро". Рекомендується обрати кнопку "Додати вершину" та натиснути на неї.

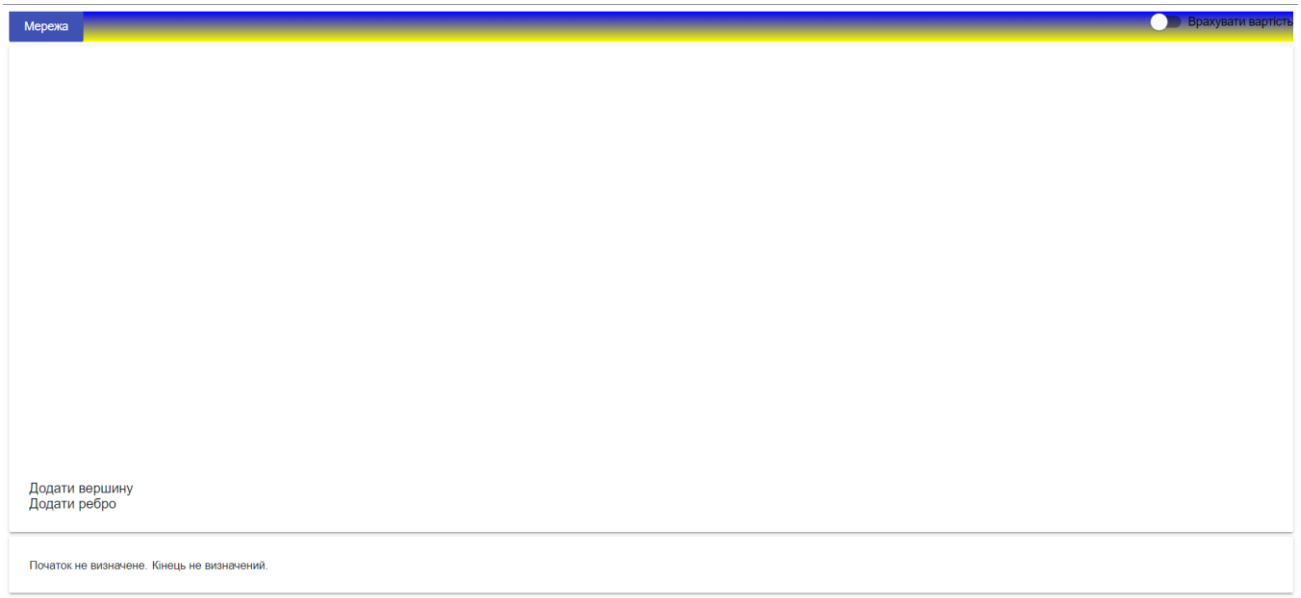

Рис.3.2.

Після цього ви можете почати розставляти вершини, використовуючи ліву кнопку миші.

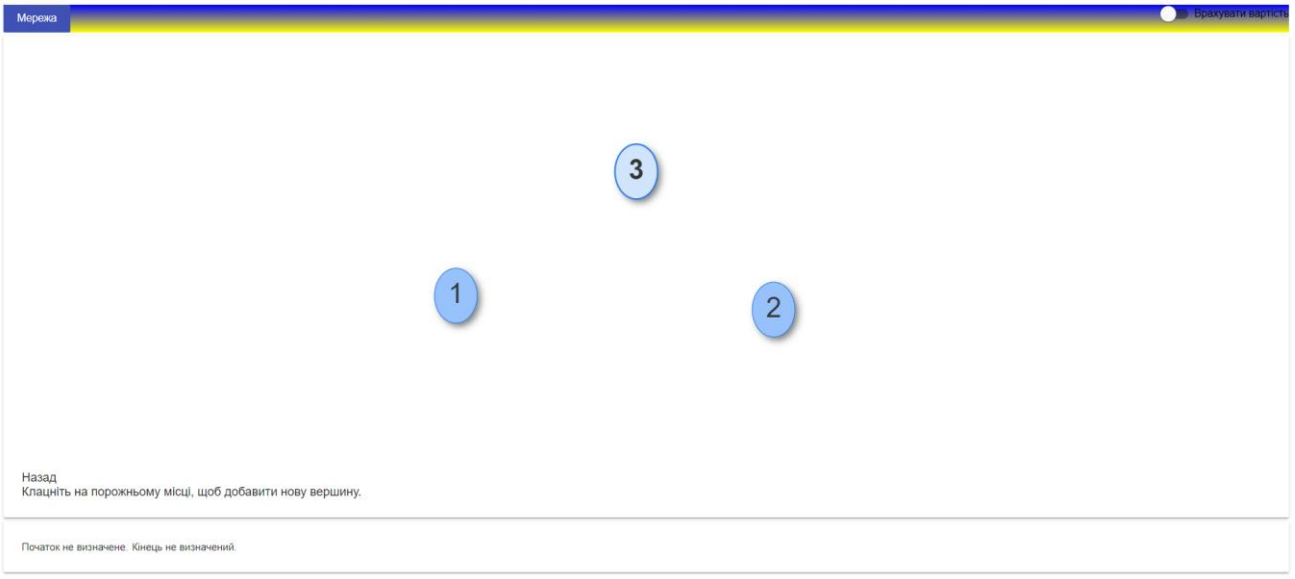

Рис.3.3

Для повернення назад, натисніть кнопку "Назад". Щоб додати ребро, виберіть відповідну кнопку. Потім ви зможете створювати ребра, натискаючи на вершину та перетягуючи ребро до іншої вершини, щоб їх з'єднати.

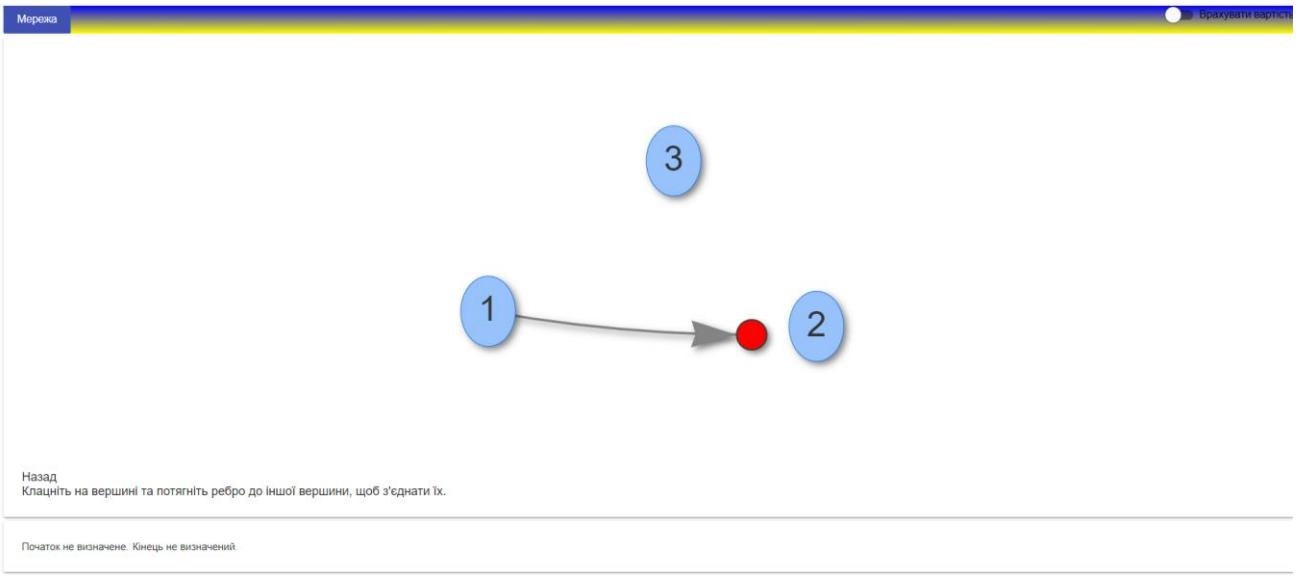

Рис.3.4.

Після успішного з'єднання двох вершин, відкриється вікно, де ви повинні ввести пропускну здатність ребра (наприклад, 3).

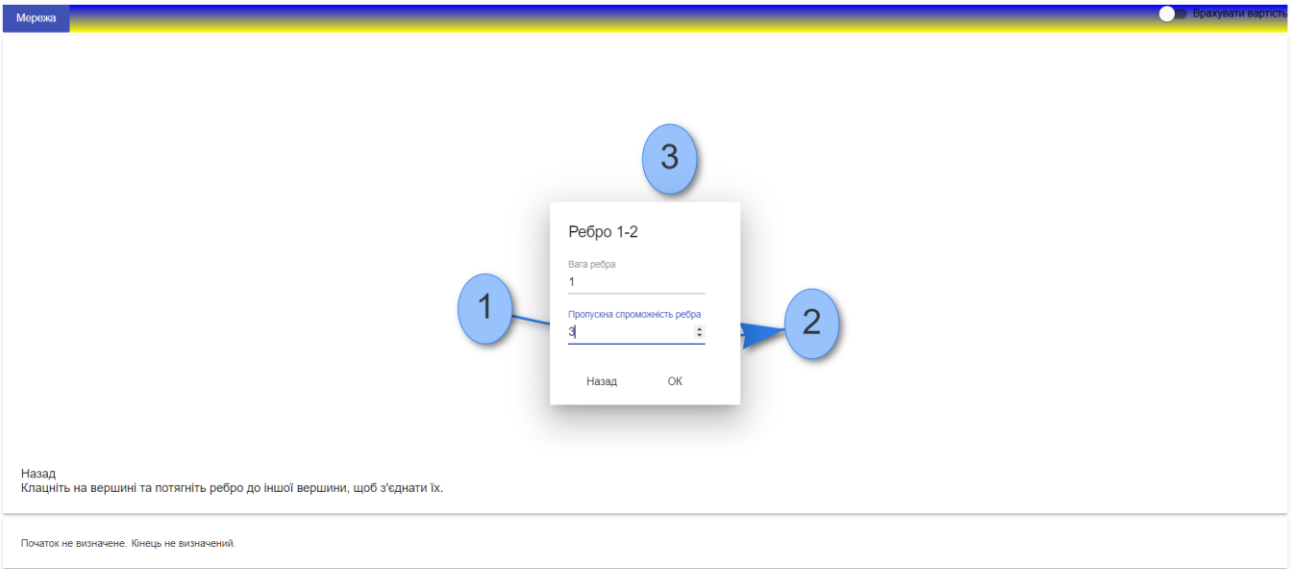

Рис.3.5.

Натисніть кнопку "ОК". Після цього ребро з'явиться з відповідними позначками. Продовжуйте повторювати попередні кроки, щоб додавати більше вершин та ребер. Щоб змінити існуюче ребро, виділіть його та натисніть "Змінити ребро". Ви зможете змінити суміжні вершини цього ребра та змінити числові значення цього ребра.

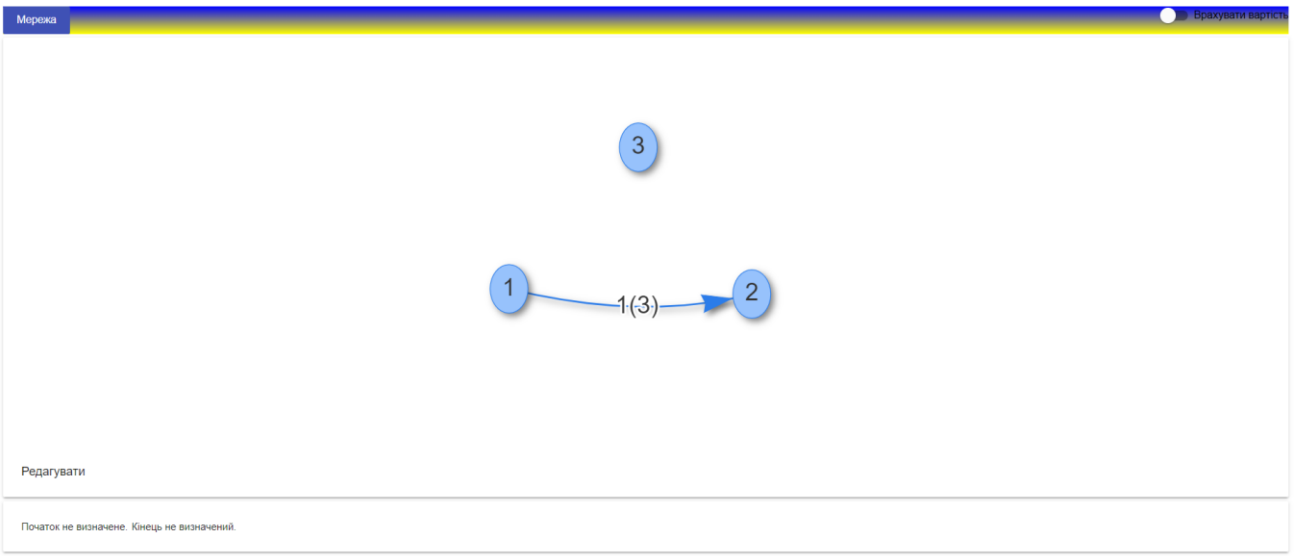

Рис.3.6.

Застосування алгоритмів

Для позначення початку та кінця візьмемо готову мережу. Клацніть на вершині "1", щоб виділити її. Далі клацніть на кнопку "Мережа" і оберіть опцію "Позначити джерелом мережі".

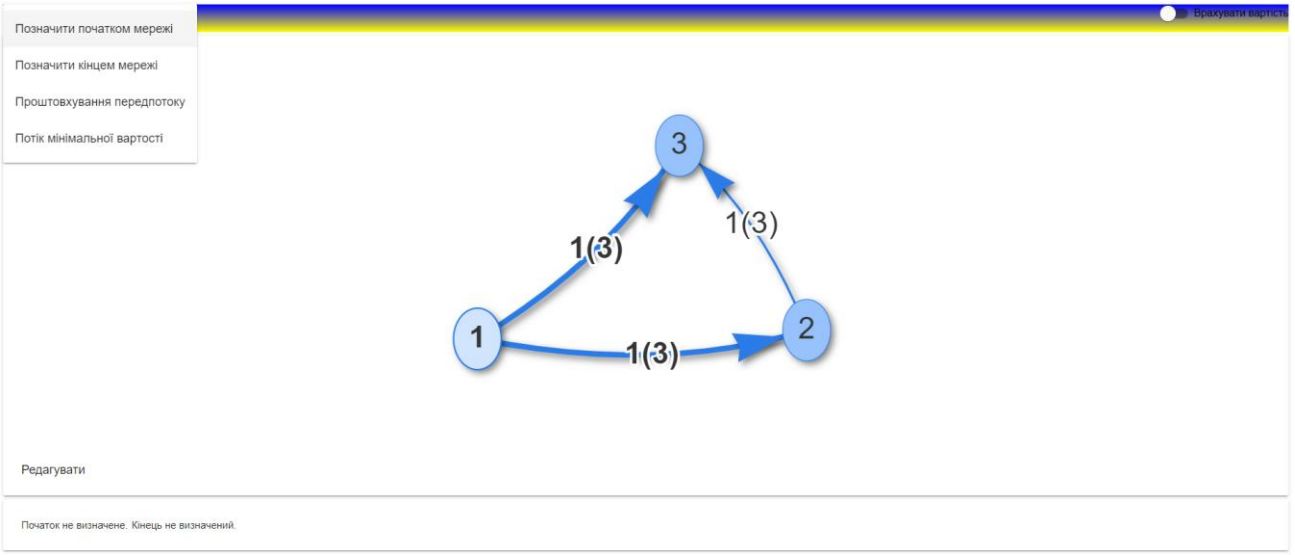

Рис.3.7.

Аналогічно оберіть вершину "3", але виберіть опцію "Позначити кінецем мережі".

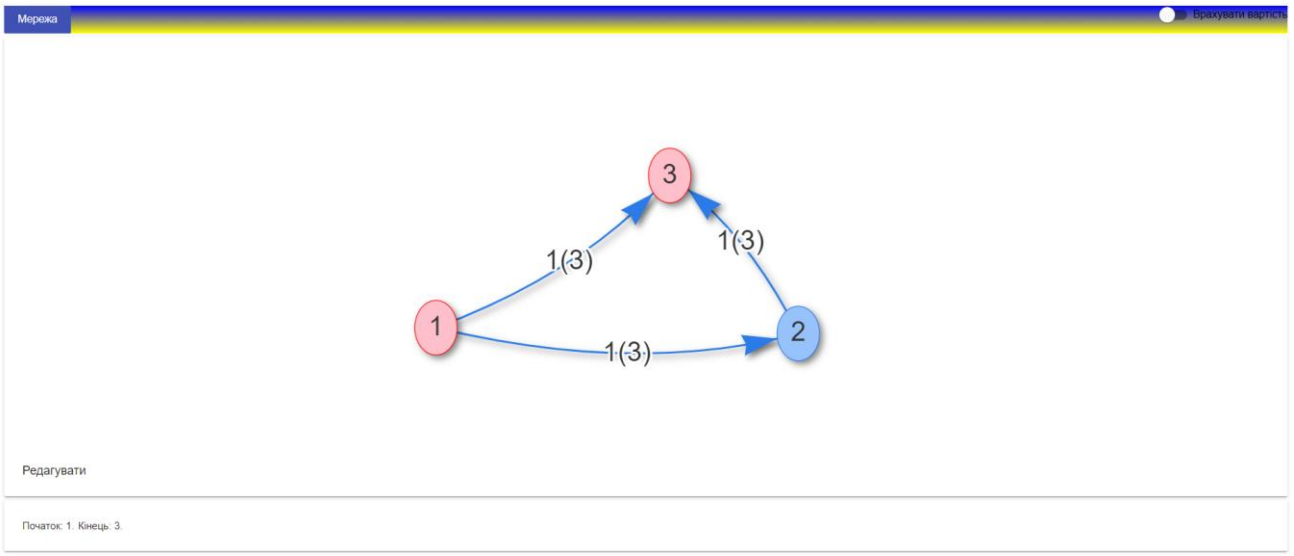

Рис.3.8.

Тепер, коли у нас є мережа з позначеним початком та кінцем, ми можемо запустити алгоритм. Натисніть на кнопку "Мережа" і оберіть алгоритм "Проштовхування передпотоку". Після цього кожне ребро мережі отримає нове значення потоку, а на панелі нижче буде відображена інформація про максимальний потік у мережі.

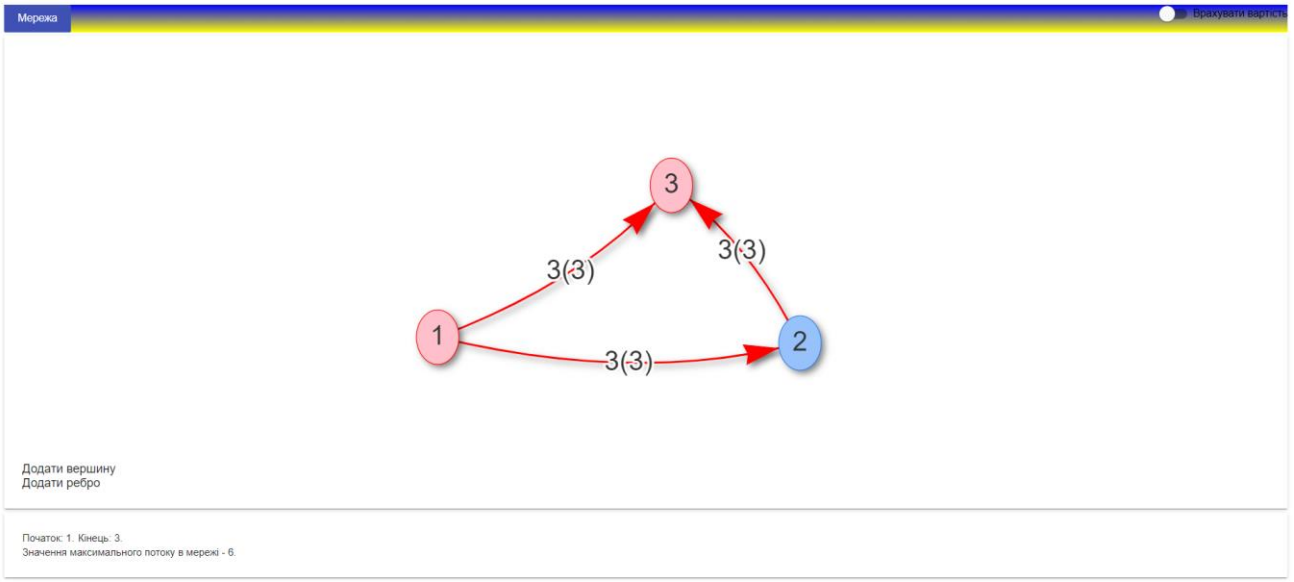

Рис.3.9.

Наступний приклад :

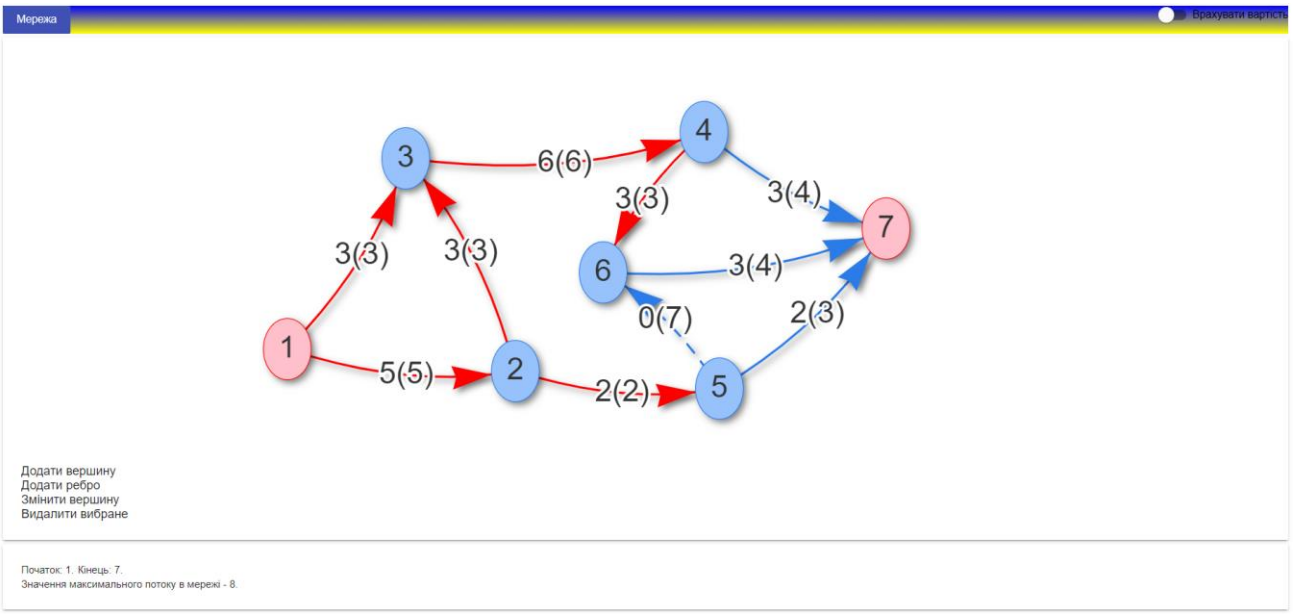

Рис.3.10.

Також є можливість застосувати алгоритм пошуку найдешевшого потоку. Для цього вмикаємо перемикач у правому верхньому куті під назвою "Враховувати вартість" та вводимо бажане значення потоку.

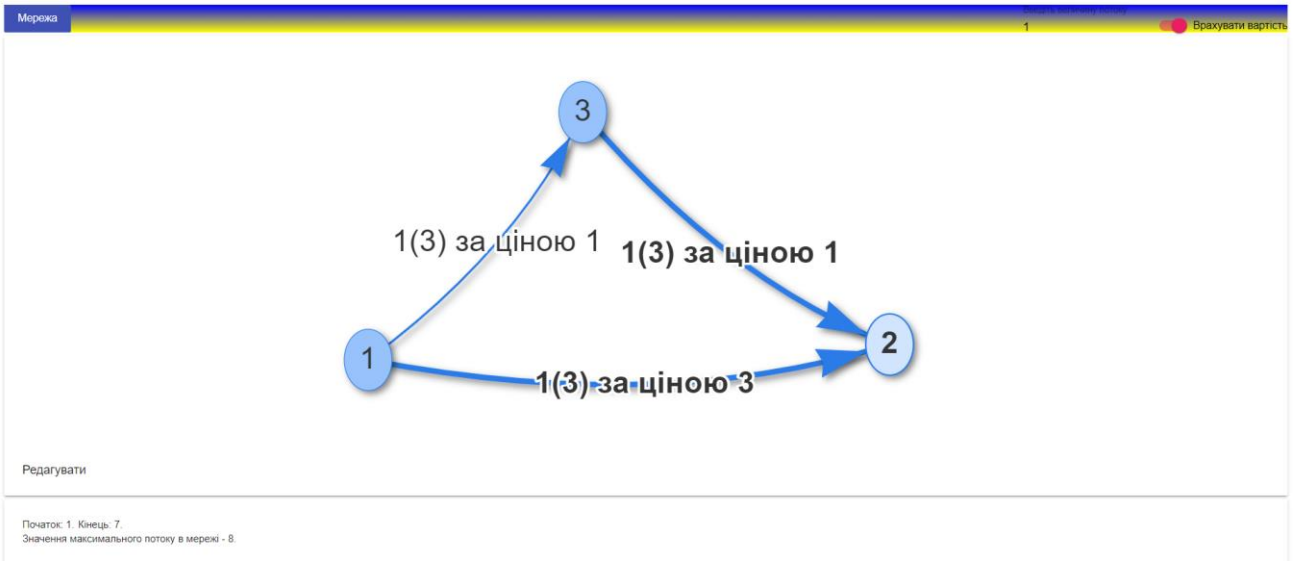

Рис.3.11.

Після цього оберіть вершину як початоком та іншу вершину як кінцем мережі, а потім натисніть на кнопку "Мережа" і оберіть опцію "Потік мінімальної ціни".

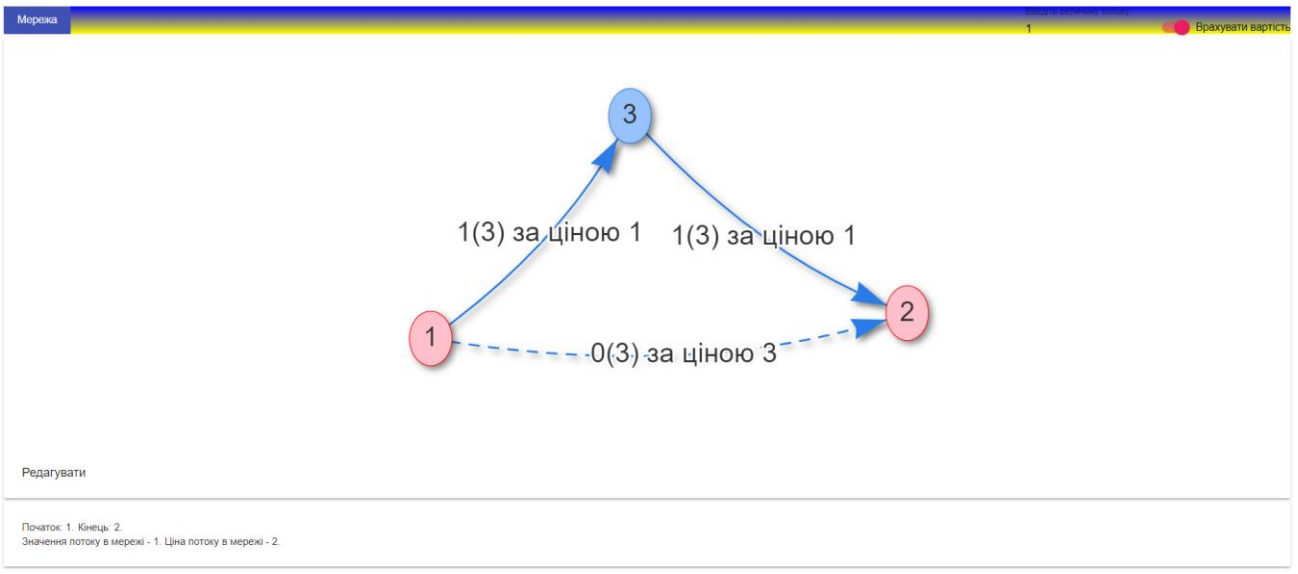

Рис.3.12.

Наступний приклад :

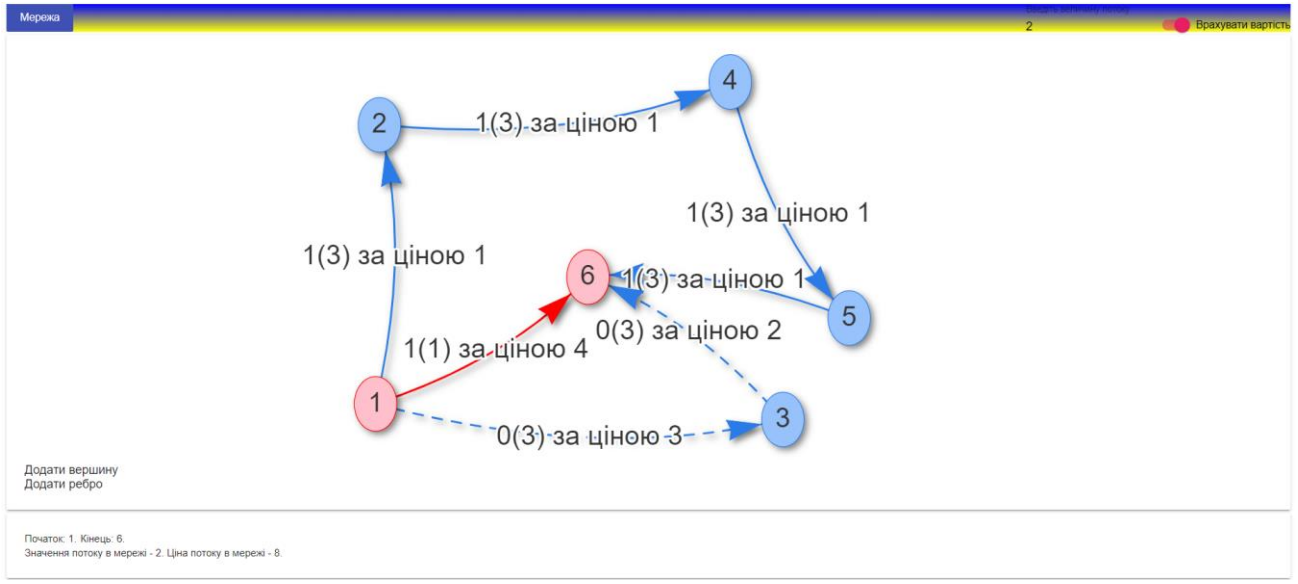

Рис.3.13.

Також є можливість застосувати алгоритм Форда-Фалкерсона.

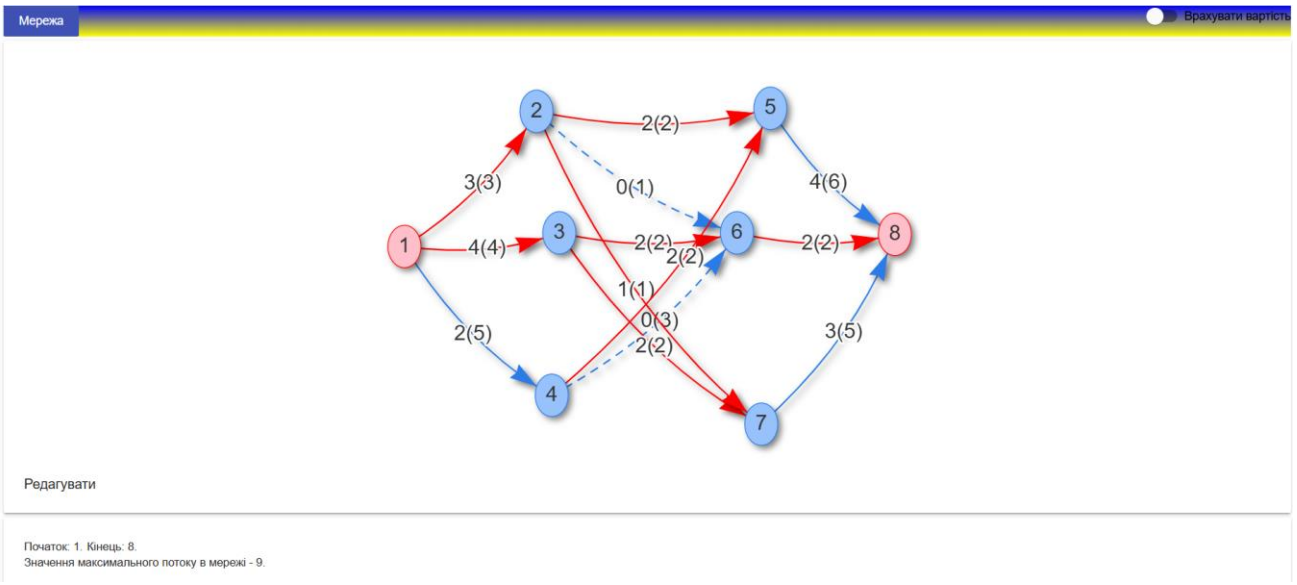
Мережа
Врахувати вартість
2
5
2(2)
3(3)
0(1)
4(6)
1
3
6
8
4(4)
2(2)
2(2)
2(2)
1(1)
0(3)
2(5)
2(2)
3(5)
4
7
Редагувати
Початок: 1. Кінець: 8.
Значення максимального потоку в мережі - 9.

Рис.3.14.

ВИСНОВОК

В роботі розглянуті та реалізовані основні алгоритми для пошуку величини потоку у мережі з можливими додатковими обмеженнями у вигляді ціни за одиницю потоку для кожного ребра. Крім того, коротко розглянуті мережі Петрі.

Оцінка роботи алгоритму Форда-Фалкерсона складає O(VE), а методу проштовхування передпотоку та алгоритму Дініца - O(EV^2). Останні два методи є більш ефективними, особливо в тих випадках, коли кількість ребер перевищує кількість вершин. Проте алгоритм Дініца може бути повільним для графів, де мало доповнюючих шляхів у кожній мережі рівнів. Це означає, що застосування обходу в ширину для знаходження мережі рівнів та обходу в глибину на цій мережі може бути повільнішим, ніж виконання кількох обходів в ширину для знаходження доповнюючих шляхів.

Розглянуті алгоритми продовжують бути предметом досліджень, існують модифіковані версії методу проштовхування передпотоку, які працюють швидше. У 2010 році Джонатан Келнер та Александр Мондри запропонували покращений алгоритм для пошуку максимального потоку у мережі.

ПЕРЕЛІК ДЖЕРЕЛ

1. AHUJA R. K., ORLIN J.B. A fast and simple algorithm for the maximum flow problem // Operation Research. – 1989. – 37, № 5. – P. 748–759. [Електронний ресурс] - JSTOR
2. Фріс Якоб Марк, Олсен Стін Бейєр. Експериментальне порівняння алгоритмів максимального потоку. – 2014. – 142 с.
3. Нікольський Ю.В., Пасічник В.В., Щербина Ю.М. Дискретна математика. – К.: Видавнича група BHV, 2007. – 368 с.
4. Глибовець М.М. Основи комп'ютерних алгоритмів. – К.: Видавничий дім "КМ Академія", 2003. – 452 с.
5. Лісовенко І.Д., Яковлєва І.Д. Паралельні та розподілені обчислення. – Чернівці: Чернівецький національний університет, 2013. – 64 с.
6. Караванова Т.П. Інформатика: методи побудови алгоритмів та їх аналіз: обчислювальні алгоритми. – К.: Генеза, 2009. – 336 с.
7. Jakob Mark Friis, Steen Beier Olesen. An experimental comparison of max flow algorithms. – 2014. – 142 p. [Електронний ресурс] - cs.au.dk
8. "TypeScript eBook" [Електронний ресурс] - RIP Tutorial
9. "Learning Angular – Third Edition" [Електронний ресурс] - Learning-Angular--Third-Edition

ДОДАТОК 1

```
import {EventEmitter, Injectable, Output} from '@angular/core';
import {AppEdge, AppNode, ResultFlowReturn, VertexLabel} from '../Models/graphModels';
import {Node} from 'vis';
import {AppConstants} from '../Models/appConstants';

@Injectable({
  providedIn: 'root'
})

export class AlgorythmService {
  @Output() flowReturned = new EventEmitter<ResultFlowReturn>();
  constructor() { }

  private async emitResultFlow(flow: number, netflow: [number[]], flowcost: number = 0) {
    // this.flowReturned.emit(new ResultFlowReturn(flow, netflow, flowcost));
    // await this.sleep(2000);

  }

  private async sleep(msec) {
    return new Promise(resolve => setTimeout(resolve, msec));
  }

  async FF(nodes: Node[], edges: AppEdge[], sourceNodeId: number, sinkNodeId: number):
Promise<ResultFlowReturn> {
    // tslint:disable-next-line:triple-equals
    const s = nodes.findIndex(x => x.id == sourceNodeId);
    // tslint:disable-next-line:triple-equals
    const f = nodes.findIndex(x => x.id == sinkNodeId);

    const netFlow: [number[]] = [[]];
    const C: [number[]] = [[]];
    for (let i = 0; i < nodes.length; i++) {
      netFlow.push([]);
      C.push([]);
      for (let j = 0; j < nodes.length; j++) {
        netFlow[i].push(0);
        C[i].push(this.getGraphEdgeCapacity(nodes, edges, i, j));
      }
    }
    let flow = 0;

    while (true) {
      const q: number[] = [];

      const lbl: VertexLabel[] = [];

      for (let i = 0; i < nodes.length; i++) {
        lbl.push(new VertexLabel());
```

```
  q.push(-1);
 }
 q[0] = s;
 lbl[0].previous = -1;
 lbl[0].flow = AppConstants.INFINITE;
 let head = 0;
 let tail = 1;
 const st: number[] = [];
 st.push(s);
 let flag = false;

 // marking
 while (head < tail && !flag) {
  let v = 0;
  while (v < nodes.length && !flag) {
   const u = q[head];
   if (C[u][v] - netFlow[u][v] > 0 && !st.includes(v)) {
    q[tail] = v;
    st.push(v);
    lbl[tail].previous = head;
    lbl[tail].flow = Math.min(lbl[head].flow, C[u][v] - netFlow[u][v]);
    tail++;
    // tslint:disable-next-line:triple-equals
    if (v == f) {
     flag = true;
    }
   }
   v++;
  }

  if (!flag) {
   head++;
  }

 }
 const newFlow = lbl[tail - 1].flow;

 if (flag) {
  flow += newFlow;

  // flows
  let k = tail - 1;
  // tslint:disable-next-line:triple-equals
  while (lbl[k].previous != -1) {
   const u = q[lbl[k].previous];
   const v = q[k];

   netFlow[u][v] = netFlow[u][v] + lbl[tail - 1].flow;
   netFlow[v][u] = -netFlow[u][v];
   k = lbl[k].previous;

  }
```

```
      await this.emitResultFlow(flow, netFlow);
    } else {
      break;
    }
  }
  const result = new ResultFlowReturn(flow, netFlow);
  return result;
}

// @ts-ignore
private initializePreflow(nodes: AppNode[],
  edges: AppEdge[],
  preflow: [number[]],
  vertexLabel: number[],
  excessFlow: number[],
  sourceNodeId: number) {
  for (let i = 0; i < nodes.length; i++) {
    vertexLabel.push(0);
    excessFlow.push(0);
    vertexLabel[sourceNodeId] = nodes.length;
  }
  for (let i = 0; i < nodes.length; i++) {
    for (let j = 0; j < nodes.length; j++) {
      const capacity = this.getGraphEdgeCapacity(nodes, edges, i, j);
      // tslint:disable-next-line:triple-equals
      if (capacity == AppConstants.INFINITE) {
        preflow[i][j] = capacity;
      }
    }
  }
  for (let i = 0; i < nodes.length; i++) {
    const capacity = this.getGraphEdgeCapacity(nodes, edges, sourceNodeId, i);
    // tslint:disable-next-line:triple-equals
    if (capacity != AppConstants.INFINITE && capacity != 0) {
      preflow[sourceNodeId][i] = capacity;
      preflow[i][sourceNodeId] = -capacity;
      excessFlow[i] = capacity;
      excessFlow[sourceNodeId] -= capacity;
    }
  }
}

// @ts-ignore
private push(nodes: AppNode[],
  edges: AppEdge[],
  nodeU: number,
  nodeV: number,
  preflow: [number[]],
  vertexLabel: number[],
  excessFlow: number[]) {
  const capacity = this.getGraphEdgeCapacity(nodes, edges, nodeU, nodeV);
  const d = Math.min(excessFlow[nodeU], capacity - preflow[nodeU][nodeV]);
```

```
    preflow[nodeU][nodeV] += d;
    preflow[nodeV][nodeU] = -preflow[nodeU][nodeV];
    excessFlow[nodeU] -= d;
    excessFlow[nodeV] += d;
  }

  // @ts-ignore
  private relabel(nodes: AppNode[],
    edges: AppEdge[],
    nodeU: number,
    preflow: [number[]],
    vertexLabel: number[]
  ) {
    const labels = [];
    for (let i = 0; i < nodes.length; i++) {
      const capacity = this.getGraphEdgeCapacity(nodes, edges, nodeU, i);
      if (preflow[nodeU][i] - capacity < 0) {
        labels.push(vertexLabel[i]);
      }
    }
    if (labels.length > 0) {
      vertexLabel[nodeU] = Math.min.apply(Math, labels) + 1;
    }
  }

  async PushRelabel(nodes: AppNode[], edges: AppEdge[], sourceNodeId: number, sinkNodeId:
number): Promise<ResultFlowReturn> {
    const s = nodes.findIndex(x => x.id == sourceNodeId); // Джерело
    const f = nodes.findIndex(x => x.id == sinkNodeId);   // Сток
    const vertexLabel: number[] = Array(nodes.length).fill(0); // Висоти вершин
    const excessFlow: number[] = Array(nodes.length).fill(0);  // Надлишкові потоки
    const preflow: number[][] = Array.from({ length: nodes.length }, () =>
Array(nodes.length).fill(0)); // Префлоу

    // Черга активних вузлів
    const activeNodes: number[] = [];

    // Ініціалізація префлоу
    this.initializePreflow(nodes, edges, preflow, vertexLabel, excessFlow, s);

    // Додати всі активні вузли (з надлишком потоку) у чергу
    for (let i = 0; i < nodes.length; i++) {
      if (i !== s && i !== f && excessFlow[i] > 0) {
        activeNodes.push(i);
      }
    }

    while (activeNodes.length > 0) {
      const u = activeNodes.shift()!; // Витягнути активний вузол
      let pushed = false;

      // Спроба виконати операцію Push для всіх сусідів
```

```
  for (let v = 0; v < nodes.length; v++) {
    if (excessFlow[u] > 0 && preflow[u][v] < this.getGraphEdgeCapacity(nodes, edges, u, v)) {
      if (vertexLabel[u] > vertexLabel[v]) {
        this.push(nodes, edges, u, v, preflow, vertexLabel, excessFlow);
        pushed = true;

        // Якщо сусід стає активним, додаємо його у чергу
        if (!activeNodes.includes(v) && excessFlow[v] > 0 && v !== s && v !== f) {
          activeNodes.push(v);
        }
      }
    }
  }

  // Якщо Push неможливий, виконуємо Relabel
  if (!pushed) {
    this.relabel(nodes, edges, u, preflow, vertexLabel);

    // Додаємо вузол назад у чергу
    if (!activeNodes.includes(u)) {
      activeNodes.push(u);
    }
  }

  // Додаткова візуалізація для проміжних результатів
  let flowValue = 0;
  for (let i = 0; i < nodes.length; i++) {
    if (preflow[i][f] > 0 && preflow[i][f] !== AppConstants.INFINITE) {
      flowValue += preflow[i][f];
    }
  }
  // @ts-ignore
  await this.emitResultFlow(flowValue, preflow);
}

// Обчислення максимального потоку
let maxFlowValue = 0;
for (let i = 0; i < nodes.length; i++) {
  if (preflow[i][f] > 0 && preflow[i][f] !== AppConstants.INFINITE) {
    maxFlowValue += preflow[i][f];
  }
}

// Упорядкування результатів
for (let i = 0; i < nodes.length; i++) {
  for (let j = 0; j < nodes.length; j++) {
    if (preflow[i][j] < 0) {
      preflow[i][j] = 0;
    }
  }
}
```

```
    // @ts-ignore
    return new ResultFlowReturn(maxFlowValue, preflow);
  }

// Операція Push
  // @ts-ignore
  private push(nodes: AppNode[], edges: AppEdge[], u: number, v: number, preflow: number[][],
vertexLabel: number[], excessFlow: number[]): void {
    const capacity = this.getGraphEdgeCapacity(nodes, edges, u, v);
    const flowToPush = Math.min(excessFlow[u], capacity - preflow[u][v]);

    preflow[u][v] += flowToPush;
    preflow[v][u] -= flowToPush;

    excessFlow[u] -= flowToPush;
    excessFlow[v] += flowToPush;
  }

// Операція Relabel
  // @ts-ignore
  private relabel(nodes: AppNode[], edges: AppEdge[], u: number, preflow: number[][],
vertexLabel: number[]): void {
    let minHeight = Infinity;

    for (let v = 0; v < nodes.length; v++) {
      const capacity = this.getGraphEdgeCapacity(nodes, edges, u, v);
      if (preflow[u][v] < capacity) {
        minHeight = Math.min(minHeight, vertexLabel[v]);
      }
    }

    if (minHeight < Infinity) {
      vertexLabel[u] = minHeight + 1;
    }
  }

// Ініціалізація префлоу
  // @ts-ignore
  private initializePreflow(nodes: AppNode[], edges: AppEdge[], preflow: number[][],
vertexLabel: number[], excessFlow: number[], source: number): void {
    vertexLabel[source] = nodes.length; // Висота джерела
    for (let v = 0; v < nodes.length; v++) {
      const capacity = this.getGraphEdgeCapacity(nodes, edges, source, v);
      if (capacity > 0) {
        preflow[source][v] = capacity;
        preflow[v][source] = -capacity;
        excessFlow[v] = capacity;
        excessFlow[source] -= capacity;
      }
    }
  }
```

```
// Отримання пропускної здатності ребра
  // @ts-ignore
  private getGraphEdgeCapacity(nodes: AppNode[], edges: AppEdge[], u: number, v: number):
number {
    const edge = edges.find(e => e.from === nodes[u].id && e.to === nodes[v].id);
    // @ts-ignore
    return edge ? edge.capacity : 0;
  }


  async MCMF(nodes: AppNode[], edges: AppEdge[], sourceNodeId: number, sinkNodeId:
number,
    targetFlow: number = AppConstants.INFINITE): Promise<ResultFlowReturn> {
    const cap: [number[]] = [[]];
    const startCap: [number[]] = [[]];

    const n = nodes.length;
    for (let i = 0; i < n; i++) {
      cap.push([]);
      startCap.push([]);
      for (let j = 0; j < n; j++) {
        cap[i].push(this.getGraphEdgeCapacity(nodes, edges, i, j));
        startCap[i].push(cap[i][j]);
      }
    }

    // tslint:disable-next-line:triple-equals
    const s = nodes.findIndex(x => x.id == sourceNodeId);
    // tslint:disable-next-line:triple-equals
    const f = nodes.findIndex(x => x.id == sinkNodeId);

    const d: number[] = [];
    const p: number[] = [];
    for (let flow = 0, flowcost = 0; ; ++flow) {
      for (let i = 0; i < n; i++) {
        d[i] = (AppConstants.INFINITE);
      }
      d[s] = 0;
      for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
          for (let k = 0; k < n; k++) {
            const cost = this.getGraphEdgeCost(nodes, edges, j, k);
            if (cap[j][k] > 0
              && d[j] < AppConstants.INFINITE
              && d[k] > d[j] + cost) {
              d[k] = d[j] + cost;
              p[k] = j;
            }
          }
        }
      }
```

```
      const flowEdges = this.subtractMatricesIgnoreNegative(startCap, cap);
      // tslint:disable-next-line:triple-equals
      if (flow == targetFlow || d[f] == AppConstants.INFINITE) {

        // let flowEdges = this.subtractMatricesIgnoreNegative(startCap, cap);

        // tslint:disable-next-line:triple-equals
        if (flow != targetFlow) {
          flow = 0;
          flowEdges.forEach(x => { if (x[f]) { flow += x[f]; } });
          return new ResultFlowReturn(flow, flowEdges, flowcost);
        }
        const result = new ResultFlowReturn(flow, flowEdges, flowcost);
        return result;
      } else {
        await this.emitResultFlow(flow, flowEdges, flowcost);
      }

      flowcost += d[f];
      // tslint:disable-next-line:triple-equals
      for (let v = f; v != s; v = p[v]) {
        --cap[p[v]][v];
        ++cap[v][p[v]];
      }
    }
  }

  private subtractMatricesIgnoreNegative(a: [number[]], b: [number[]]) {
    const c: [number[]] = [[]];
    for (let i = 0; i < a.length; i++) {
      c.push([]);
      for (let j = 0; j < a[i].length; j++) {
        c[i].push(a[i][j] - b[i][j]);
        if (c[i][j] < 0) {
          c[i][j] = 0;
        }
      }
    }
    return c;
  }

  // @ts-ignore
  private getGraphEdgeCapacity(nodes: Node[], edges: AppEdge[], i: number, j: number):
number {
    // tslint:disable-next-line:triple-equals
    if (i == j) {
      return 0;
    }

    const startNodeId = nodes[i].id;
    const endNodeId = nodes[j].id;
```

```
    // tslint:disable-next-line:triple-equals
    const edge = edges.find(x => x.from == startNodeId && x.to == endNodeId);
    // tslint:disable-next-line:triple-equals
    return edge != undefined ? edge.getCapacity() : 0;
  }

  private getGraphEdgeCost(nodes: Node[], edges: AppEdge[], i: number, j: number): number {
    // tslint:disable-next-line:triple-equals
    if (i == j) {
      return 0;
    }

    const startNodeId = nodes[i].id;
    const endNodeId = nodes[j].id;

    // tslint:disable-next-line:triple-equals
    const edge = edges.find(x => x.from == startNodeId && x.to == endNodeId);
    // tslint:disable-next-line:triple-equals
    return edge != undefined ? edge.getCost() : AppConstants.INFINITE;
  }

}
```

ДОДАТОК 2

```
import { Component, OnInit, ApplicationRef, OnDestroy } from '@angular/core';
import { AppNode, AppEdge, AppColor, VertexLabel, EdgeColor, ResultFlowReturn } from
'../Models/graphModels';
import { MatDialog } from '@angular/material';
import * as Vis from 'vis';
import { Network, Edge, Node, Options } from 'vis';
import { DialogComponent } from '../dialog/dialog.component';
import { DataTransferService } from '../data-transfer.service';
import { AppConstants } from '../Models/appConstants';
import { AlgorythmService } from '../Services/algorythm.service';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-view',
  templateUrl: './view.component.html',
  styleUrls: ['./view.component.css']
})

export class ViewComponent implements OnInit, OnDestroy {

  graphNodes: AppNode[] = [];
  graphEdges: AppEdge[] = [];
  network: Network = null;
  networkElement: any;
  sinkNode: AppNode = null;
  sourceNode: AppNode = null;
  // exportData: string = '';
  options: Options;
  resultsText = '';
  areCostsEnabled = false;
  targetFlow = 1;
  subscription: Subscription;

  constructor(
    public transferService: DataTransferService,
    private algoService: AlgorythmService,
    public dialog: MatDialog,
    private appRef: ApplicationRef) {
    this.subscription = this.algoService.flowReturned.subscribe(x => {

      this.writeFlows(x);
      this.network.redraw();
      this.appRef.tick();
    });
  }

  ngOnInit() {
    const self = this;
    this.options = {
      // height: '400px',
```

```
// enabled: true,
// initiallyActive: true,
// addNode: true,
// addEdge: true,
// editNode: undefined,
// editEdge: true,
// deleteNode: true,
// deleteEdge: true,
// controlNodeStyle: {
//   // all node options are valid.
// },

// autoResize: false,
locale: 'ukr',
locales: AppConstants.UALocale,
manipulation: {
  addNode: function (nodeData, callback) {
    self.onAddNode(nodeData, callback);
  },
  addEdge: function (edgeData, callback) {
    self.onAddEdge(edgeData, callback);
  },
  deleteNode: function (nodeEdgesData, callback) {
    self.onDeleteNode(nodeEdgesData, callback);
  },
  deleteEdge: function (edgeData, callback) {
    self.onDeleteEdge(edgeData, callback);
  },
  editEdge: function (edgeData, callback) {
    self.onEditEdge(edgeData, callback);
  },
  editNode: function (nodeData, callback) {
    self.onEditNode(nodeData, callback);
  }
},

nodes: {
  shadow: true
},
edges: {
  arrows: {
    to: {
      enabled: true
    },
  },
  smooth: {
    enabled: true,
    forceDirection: 'none',
    type: 'curvedCCW',
    roundness: 0.14
  },
  shadow: true,
```

```
      color: {
        inherit: false
      }
    },
    physics: {
      enabled: false,
      timestep: 0.6,
      repulsion: {
        springConstant: 0.03
      }
    },
    interaction: {
      navigationButtons: true
    }
    // ,height: '400px'
  };

  this.buildGraph(this.options);
 }

 ngOnDestroy() {
  this.subscription.unsubscribe();
 }

 onAddNode(nodeData: Node, callback: any) {
  console.log('add', nodeData);

  const newNode = new AppNode(nodeData.id, (this.graphNodes.length + 1).toString(),
nodeData.x, nodeData.y);

  this.graphNodes.push(newNode);
  callback(newNode);
  this.network.addNodeMode();
 }

 onAddEdge(edgeData: Edge, callback: any) {

  const newEdge = new AppEdge(edgeData.from as number, edgeData.to as number, 1, 1, 0,
this.areCostsEnabled, edgeData.id);
  this.graphEdges.push(newEdge);
  callback(newEdge);
  this.openDialog(newEdge);
  this.network.addEdgeMode();
 }

 onEditNode(nodeData: Node, callback: any) {
  callback(nodeData);
 }

 onEditEdge(edgeData: Edge, callback: any) {

  const newEdge = new AppEdge(edgeData.from as number, edgeData.to as number, 0, 0, 0,
```

```
this.areCostsEnabled, edgeData.id);
    // tslint:disable-next-line:triple-equals
    const index = this.graphEdges.findIndex(ge => ge.id == newEdge.id);
    if (index > -1) {

      this.graphEdges[index].from = newEdge.from;
      this.graphEdges[index].to = newEdge.to;
      // this.graphEdges[index]
      this.openDialog(this.graphEdges[index]);
      callback(this.graphEdges[index]);
    }

    // this.network.addEdgeMode();
  }

  onDeleteNode(nodeEdgesData: any, callback: any) {
    (nodeEdgesData.edges as string[]).forEach(x => {
      // tslint:disable-next-line:triple-equals
      const index = this.graphEdges.findIndex(ge => ge.id == x);
      if (index > -1) {
        this.graphEdges.splice(index, 1);
      }
    });

    (nodeEdgesData.nodes as string[]).forEach(x => {
      // tslint:disable-next-line:triple-equals
      const index = this.graphNodes.findIndex(gn => gn.id == x);
      if (index > -1) {
        this.graphNodes.splice(index, 1);
      }
    });

    callback(nodeEdgesData);
  }

  onDeleteEdge(nodeEdgesData: any, callback: any) {
    this.onDeleteNode(nodeEdgesData, callback);
  }

  openDialog(edge: AppEdge): void {
    const copied = new AppEdge(edge.from, edge.to, edge.getWeight(), edge.getCapacity(),
edge.getCost(), this.areCostsEnabled, edge.id);
    // tslint:disable-next-line:triple-equals
    const fromLabel = this.graphNodes.find(x => x.id == copied.from).label;
    // tslint:disable-next-line:triple-equals
    const toLabel = this.graphNodes.find(x => x.id == copied.to).label;
    const dialogRef = this.dialog.open(DialogComponent, {
      width: '250px',
      data: { edge: copied, fromLabel: fromLabel, toLabel: toLabel }
    });

    dialogRef.afterClosed().subscribe(result => {
```

```
    // console.log('The dialog was closed');
    if (result) {
      edge = result.edge;
      // tslint:disable-next-line:triple-equals
      const index = this.graphEdges.findIndex(x => x.id == edge.id);
      if (index >= 0) {
        this.graphEdges[index] = edge;
        this.updateNetwork();

        // this.network.redraw();
      }
    }
  });
}

setCanvasHeight(height) {
  const canvas = document.getElementsByTagName('canvas')[0];
  // canvas.width = 800;
  canvas.height = height;
}

buildGraph(options: any) {
  this.networkElement = document.getElementById('network');

  const nodesDataset = new Vis.DataSet(this.graphNodes);
  const edgesDataSet = new Vis.DataSet(this.graphEdges);

  const data = {
    nodes: nodesDataset,
    edges: edgesDataSet
  };

  this.network = new Vis.Network(this.networkElement, data, options);

  this.dragNodesSubscribe();
  this.setCanvasHeight(600);
}

updateNetwork() {
  const nodesDataset = new Vis.DataSet(this.graphNodes);
  const edgesDataSet = new Vis.DataSet(this.graphEdges);
  const data = {
    nodes: nodesDataset,
    edges: edgesDataSet
  };

  this.network.setData(data);
  this.setCanvasHeight(600);
}

exportNetwork() {
  this.transferService.exportNetwork(this.graphEdges, this.graphNodes);
```

```
  }

  importNetwork() {
    this.graphEdges = [];
    this.graphNodes = [];
    this.clearNetwork();
    const networkElement = document.getElementById('network');
    const data = this.transferService.importNetwork(this.graphEdges, this.graphNodes);

    this.network = new Vis.Network(networkElement, data, this.options);
    this.setCanvasHeight(400);

    this.dragNodesSubscribe();
  }

  clearNetwork() {
    this.sourceNode = null;
    this.sinkNode = null;
    this.resultsText = '';

    this.dragNodesUnsubscribe();
    this.network.destroy(); // setData(data);
  }


  isSetSourceSinkDisabled(): boolean {
    if (!this.network) {
      return false;
    }
    // tslint:disable-next-line:triple-equals
    return this.network.getSelectedNodes().length != 1;
  }

  onSetSource() {
    const sourceNodeId = this.network.getSelectedNodes()[0];
    // tslint:disable-next-line:triple-equals
    this.sourceNode = this.graphNodes.find(x => x.id == sourceNodeId);
    this.sourceNode.color = new AppColor('pink', 'red');

    this.updateNetwork();
  }

  onSetSink() {
    const sinkNodeId = this.network.getSelectedNodes()[0];
    // tslint:disable-next-line:triple-equals
    if (!this.sinkNode || sinkNodeId != this.sinkNode.id) {
      if (this.sinkNode) {
        this.sinkNode.color = new AppColor(AppConstants.DefaultBgColor,
AppConstants.DefaultBorderColor);
      }
      // tslint:disable-next-line:triple-equals
      this.sinkNode = this.graphNodes.find(x => x.id == sinkNodeId);
```

```
    this.sinkNode.color = new AppColor('pink', 'red');
    this.updateNetwork();
  }
}

// methods
BFS() {
  let node = -1;
  const visited = [];
  const queue = [];
  const vertexes = [[]];

  for (let i = 0; i < this.graphNodes.length; i++) {
    vertexes.push([]);
    for (let j = 0; j < this.graphNodes.length; j++) {
      const edgeValue = this.getGraphEdge(i, j);
      // tslint:disable-next-line:triple-equals
      if (edgeValue != AppConstants.INFINITE && edgeValue != 0) {
        vertexes[i].push(j);
      }
    }
    visited.push(false);
  }

  // tslint:disable-next-line:triple-equals
  const s = this.graphNodes.findIndex(x => x.id == this.sourceNode.id);
  // tslint:disable-next-line:triple-equals
  const f = this.graphNodes.findIndex(x => x.id == this.sinkNode.id);
  queue.push(s);
  visited[s] = true;

  while (queue.length > 0) {
    node = queue.pop();
    // tslint:disable-next-line:triple-equals
    if (node == f) {
      console.log(true);
      return true;
    }
    for (let i = 0; i < vertexes[node].length; i++) {
      if (!visited[vertexes[node][i]]) {
        queue.push(vertexes[node][i]);
        visited[vertexes[node][i]] = true;
      }
    }
  }
  console.log(false);
  return false;
}

getGraphEdge(i: number, j: number): number {
  const startNodeId = this.graphNodes[i].id;
  const endNodeId = this.graphNodes[j].id;
```

```
    // tslint:disable-next-line:triple-equals
    const edge = this.graphEdges.find(x => x.from == startNodeId && x.to == endNodeId);
    // tslint:disable-next-line:triple-equals
    return edge != undefined ? edge.getWeight() : AppConstants.INFINITE;
  }

  writeFlows(resultFlow: ResultFlowReturn) {
    // redraw
    this.graphEdges.forEach(e => {
      // tslint:disable-next-line:triple-equals
      const fromIndex = this.graphNodes.findIndex(n => n.id == e.from);
      // tslint:disable-next-line:triple-equals
      const toIndex = this.graphNodes.findIndex(n => n.id == e.to);
      e.setWeight(resultFlow.flowEdges[fromIndex][toIndex]);
      this.updateNetwork();
    });

    if (this.areCostsEnabled) {
      this.resultsText = 'Значення потоку в мережі - ' + resultFlow.flowValue.toString() + '.';
      this.resultsText += ' Ціна потоку в мережі - ' + resultFlow.totalCost.toString() + '.';
    } else {
      this.resultsText = 'Значення максимального потоку в мережі - ' +
resultFlow.flowValue.toString() + '.';
    }
  }

  // ff
  FF() {
    const result = this.algoService.FF(this.graphNodes, this.graphEdges, this.sourceNode.id,
this.sinkNode.id);
    result.then(x =>
      this.writeFlows(x)
    );
  }

  // PR
  PushRelabel() {
    const result = this.algoService.PushRelabel(this.graphNodes, this.graphEdges,
this.sourceNode.id, this.sinkNode.id);
    result.then(x => {
      this.writeFlows(x);
    });
  }

  // mincost
  MinCostMaxFlow() {
    this.areCostsEnabled = true;
    this.costChanged({ checked: true });
    const result = this.algoService.MCMF(this.graphNodes, this.graphEdges, this.sourceNode.id,
this.sinkNode.id, this.targetFlow);
    result.then(x => {
      this.writeFlows(x);
```

```
    });
  }

  costChanged($event) {
    this.graphEdges.forEach(x => x.setUseCost($event.checked));
    this.updateNetwork();
  }

  dragNodesSubscribe() {
    // add event listener
    const self = this;
    // this.network.on('dragEnd', self.onSelect);
    this.network.on('dragEnd', function (properties: any) { self.onSelect(properties); });
  }

  dragNodesUnsubscribe() {
    const self = this;
    this.network.off('dragEnd', function (properties: any) { self.onSelect(properties); });
  }

  onSelect(properties) {
    // alert('selected nodes: ' + properties.nodes);
    if (properties.nodes.length > 0) {
      // tslint:disable-next-line:triple-equals
      const node = this.graphNodes.find(x => x.id == properties.nodes[0]);
      node.x = properties.pointer.canvas.x;
      node.y = properties.pointer.canvas.y;
    }
  }
}
```