Міністерство освіти і науки України
Кам’янець-Подільський національний університет імені Івана Огієнка
Фізико-математичний факультет
Кафедра комп’ютерних наук

Кваліфікаційна робота бакалавра

з теми: «Дослідження алгоритмів пошуку максимального потоку в потоковій мережі»

Виконав: здобувач вищої освіти групи KN1-B21
спеціальності 122 Комп’ютерні науки

Андронік Дмитро Іванович

Керівник: Щирба Віктор Самуїлович,
кандидат фізико-математичних наук, доцент

Рецензент:Отпасюк Сергій Васильович ,
кандидат фізико-математичних наук, доцент

м. Кам’янець-Подільський – 2025 р.

ЗМІСТ

АНОТАЦІЯ

В роботі досліджено ряд класичних алгоритмів пошуку оптимальних маршрутів у потоковій мережі. Актуальність їх розгляду обумовлено доцільністю подати певного класу прикладні задачі мовою графів, а потім шукати оптимальний результат у вихідних термінах. Показано, що арсенал алгоритмів на графах є досить об’ємним, тому в роботі наголошується на необхідності порівняти і показати не лише позитивні сторони, але й недоліки алгоритмів пошуку в мережі.

Матеріал кваліфікаційної роботи може бути використаний при розробці програмного забезпечення для розв’язання оптимізаційних задач, а у навчальному процесі при вивченні освітніх компонент «Алгоритми та структури даних», «Дискретні структури», в ряді вибіркових компонент, а також у практичній діяльності фахівців з комп’ютерних наук.

Ключові слова: теорія графів та її застосування, потокові мережі, оптимізаційні алгоритми на графах.

ВСТУП

При плануванні роботи будь-яких організацій завжди визначальну роль відіграють питання економіки. Сьогодні мотиваційні принципи економіки органічно переплітаються з фактором стрімкого розвитку інформаційних технологій і в подальшому все більш важливого значення набувають прикладні задачі оптимізації потоків у різного роду мережах — від комп'ютерних технологічних систем і логістики транспортних перевезень до енергетичних і навіть соціальних мереж. Усі ці мережі мають спільну рису — їх легко і ефективно можна змоделювати, використовуючи теорію графів. Графові моделі дозволяють за допомогою математичних методів та алгоритмів проводити аналіз, прогнозування та оптимізацію при плануванні роботи організацій, підприємств, фірм чи компаній.

Однією з ключових задач, яка виникає при роботі з графами, є задача пошуку максимального потоку в мережі. Вона має широке застосування в таких галузях, як планування ресурсів, керування транспортними потоками, розподіл інформації, побудова маршрутів передачі даних, а також в економіці та біоінформатиці. Задача максимального потоку є базовою у теорії мереж і виступає фундаментом для розробки більш складних моделей, що враховують додаткові обмеження, цінові або часові характеристики.

Метою даної кваліфікаційної роботи є аналіз, дослідження та реалізація класичних і сучасних алгоритмів пошуку максимального потоку в потокових мережах. Об'єкт дослідження – потокові мережі. Предметом дослідження кваліфікаційної роботи виступають способи та моделі пошуку оптимальних маршрутів у потоковій мережі.

У процесі дослідження розглянуто основи теорії графів як базису для розв'язання прикладних задач, проведено огляд основних алгоритмів, серед яких особливу увагу приділено алгоритмам Форда-Фалкерсона, Едмондса-Карпа та алгоритмам пошуку потоків мінімальної вартості.

У першому розділі роботи на основі аналізу літературних джерел висвітлено теоретичні аспекти графів, класифікацію їхніх видів, базові властивості та загальну класифікацію задач на графах. Другий розділ присвячено безпосередньо дослідженню алгоритмів максимального потоку, їх перевагам, недолікам та умовам застосування.

У третьому розділі представлено програмну реалізацію досліджуваних алгоритмів, приклади їх використання, а також аналіз ефективності в різних умовах задач.

В додатках подано код веб-додатку розроблено з використанням фреймворку Angular.

Актуальність теми і значимість одержаних результатів полягають у високій практичній значущості задач оптимізації в потокових мережах та необхідності використання ефективних алгоритмів для їх вирішення в умовах зростаючої складності інформаційних систем. Результати дослідження можуть бути використані при розробці програмного забезпечення для розв’язання оптимізаційних задач, а у навчальному процесі при вивченні дисциплін «Алгоритми та структури даних», «Дискретні структури», ряді вибіркових компонент, наприклад, «Дослідження операцій», а також у практичній діяльності фахівців з комп’ютерних наук.

За результатами дослідження опубліковано дві роботи:

1. Віктор ЩИРБА, Дмитро АНДРОНІК Задача визначення максимального потоку в транспортній мережі. Вісник Кам’янець-Подільського національного університету імені Івана Огієнка. Фізико-математичні науки. Випуск 17. 2024, с 205-208.
2. Дмитро АНДРОНІК Дослідження алгоритмів пошуку максимального потоку в потоковій мережі. Збірник матеріалів наукової конференції за підсумками науково-дослідної роботи здобувачів вищої освіти фізико-

математичного факультету у 2024-2025 н. р. 9-10 квітня 2025 року. с. 5-7.

Доповідь «Дослідження алгоритмів пошуку максимального потоку в потоковій мережі» представлялася на науковій конференції студентів і магістрантів за підсумками НДР у 2024-2025 навчальному році.

Розроблені в роботі методи та інструментальні засоби можуть впроваджуватися у навчальний процес Кам’янець-Подільського національного університету імені Івана Огієнка на фізико-математичному факультеті.

РОЗДІЛ 1. ТЕОРІЯ ГРАФІВ ТА ЇЇ ЗАСТОСУВАННЯ

1.1 Огляд літературних джерел

Теорія графів надає потужний інструментарій для аналізу та моделювання різного типу взаємодій і взаємозв'язків у багатогранних напрямках дослідження, від проектування комп'ютерних мереж до аналізу соціальних структур. Досить часто вирішення певного прикладного завдання доцільно одержати розв'язок мовою графів, а потім знайдений результат інтерпретувати в необхідних вихідних термінах.

Теорія графів допомагає розкрити як властивості самих графів, наприклад, ізоморфізм, шляхи, цикли, зв'язність, потоки тощо, так і розробляти оптимальні алгоритми, що ефективно використовуються для різноманітних задач обробки моделей, створених на основі графів.

Враховуючи практичну цінність графів, значна кількість науковців працювала і продовжує працювати в напрямку їх дослідження. Тож не дивно, що вивченню графів присвячено чимало наукових, навчальних та навчально методичних праць, зокрема [1 - 8]. Серед них є праці опубліковані лише нещодавно, що свідчить про наукову актуальність теорії графів і повсякчас.

В переліку використаних літературних джерел вказано лише декілька праць навчального характеру, в яких вивчаються графи [1, 7]. Теоретичні відомості про графи можна знайти в усіх підручниках з дискретної математики. Зміст відповідних розділів у них практично не відрізняється.

Також, розглядаючи питання практичного застосування графів, у підручниках з дослідження операцій, наприклад, [2, 5, 6] наводять основні поняття та властивості про них. Також основні поняття про графи і їх властивості описані в [3].

Варто відмітити, що ці праці основну увагу приділяють опису алгоритмів пошуку маршрутів у графах і, на що також можна звернути увагу, що серед усіх можливих методів зберігання графів у пам’яті комп’ютера, вони надають перевагу матриці суміжності, а в оптимізаційних алгоритмах пропонують використовувати вагову матрицю.

Зрозуміло, що практична цінність графів породила чимало алгоритмів розв’язування прикладних задач, які моделюються за допомогою графів. Про їх величезну кількість (об’єм) можна судити, зокрема, по [4]. В підручниках з дослідження операцій [2, 5, 6], в [3] та ряді інших літературних джерелах розглядають найбільш вживані, так би мовити, класичні алгоритми.

Звичайно, що в своїй кваліфікаційній роботі я також розглядатиму лише найпопулярніші алгоритми ті, які стосуються теми моєї роботи.

1.2 Алгоритми на графах

Поняття графа дозволяє швидко структурувати інформацію і алгоритмізувати цілий ряд типових прикладних задач. Саме тому розроблено цілий ряд алгоритмів, які обробляють інформацію, що представлена у вигляді графів. Такі алгоритми ще називають алгоритми на графах.

Практично усі алгоритми на графах пов’язані з пошуком маршрутів. На перший погляд, може здаватися, що в графах з невеликим числом вершин і ребер розв’язок таких задач є очевидним (легко знаходиться візуально).

Наприклад, для пошуку гамільтонового маршруту в задачі комівояжера на повному графі з чотирьох вершин (див. рисунок 1.1) легко знаходимо усі шість варіантів розв’язку:

I - (1, 2, 3, 4, 1); II - (1, 4, 3, 2, 1); III - (1, 3, 4, 2, 1);

IV - (1, 2, 4, 3, 1); V - (1, 3, 2, 4, 1); VI - (1, 4, 2, 3, 1).

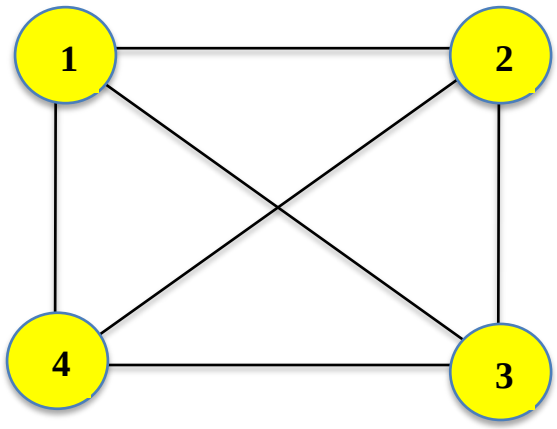

Рисунок 1.1. Повний граф із чотирьох вершин

Кількість варіантів розв’язку задачі значно розшириться, якщо в цій задачі вимагати відвідування, наприклад, вершини 2 два рази.

При старті з вершини 2 одержимо шість варіантів, як і в попередньому випадку.

При старті з іншої вершини, наприклад 1, розв’язками будуть маршрути:

I - (1, 2, 3, 4, 2, 1); II - (1, 2, 4, 3, 2, 1); III - (1, 3, 2, 4, 2, 1);

IV - (1, 2, 4, 2, 3, 1); V - (1, 4, 2, 3, 2, 1); VI - (1, 2, 3, 2, 4, 1).

Також не так швидко візуально знаходяться розв’язки наступної задачі: на товарній станції, схема якої подана на рисунку 1.2, потрібно вагони переставити у відповідність їх номерів і номерів колій. Яку послідовність дій потрібно виконати маневровому локомотиву після виходу з депо, якщо на кожній із віток колій можна одночасно розмістити не більше семи вагонів?

Якщо позначити ділянку колій від депо до розгалужень через вершину з номером 0, а інші ділянки позначити через номера відповідних віток, з’єднавши їх належним чином ребрами, то одержимо модель у вигляді граф-дерева, зображеного на рисунку 1.3.

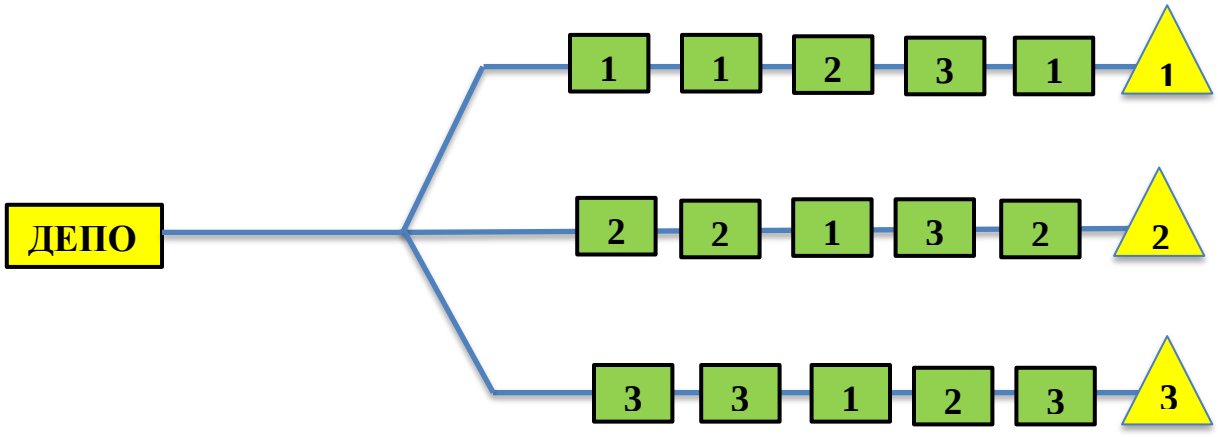

Рисунок 1.2. Схема розміщення вагонів на товарній станції

Тоді перехід з однієї колії на іншу буде асоціюватися із відвідуванням вершин (аналогічно, як у задачі комівояжера).

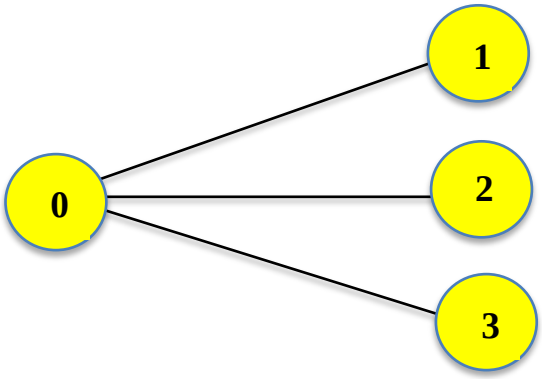

Рисунок 1.3. Граф-дерево із чотирьох вершин

На відмінну від попередніх задач, де ми намагалися відшукати всеможливі маршрути, тут нас цікавлять, очевидно, лише ті маршрути, які є найкоротшими, де найменша кількість переходу з однієї колії на іншу. Таким чином, ми маємо справу з оптимізаційною задачею. Це вже інший тип задач на графах. В окремих прикладних задачах вимагається пошук найдовшого (критичного) маршруту. Прикладом, зокрема, може слугувати задача

планування будівельно-ремонтних робіт, коли потрібно визначити термін завершення цих робіт.

Оптимізаційні задачі можна, в свою чергу, поділити на два різних типи: коли ребра є рівнозначними, як в попередній задачі, і коли мають різну вагу, як у задачі пошуку найкоротшого маршруту між містами.

Остання задача може бути, в свою чергу, ще складнішою, багатокритеріальною. Наприклад, між містами на маршруті можуть знаходитися мости, через які є обмеження на перевезення вантажів, (проїзд автомобілів обмежені за вагою) або можуть бути ділянки з обмеженою швидкістю руху. Потрібно додатково встановити максимально можливий потік за об’ємом перевезення, чи мінімальний за часом доставки.

Кожен з зазначених вище типів задач потребує іншого підходу до розв’язання задачі, іншого алгоритму. Тому існує велика різноманітність алгоритмів та постійно з’являються нові.

У Вікіпедії [4] зазначено більше п’ятдесяти (див. таблицю 1.1) різних алгоритмів і це ще, зрозуміло, не всі. Там, зокрема, не відображені досить ефективні та потужні, в якійсь мірі ще зовсім нові і малодосліджені, сучасні алгоритми, наприклад, евристичні алгоритми, мурашині алгоритми, генетичні алгоритми, алгоритми обробки великих даних тощо.

Можливо, при невеликій кількості вершин і ребер можна легко візуально відшукати необхідний розв’язок задачі, але, коли їх кількість вимірюється десятками чи сотнями і більше, зробити це без комп’ютерної обробки графу просто не реально.

Як показує практика, при розв’язуванні таких задач особливо важливо, по-перше, чітко та вірно побудувати модель і раціонально розмістити її в пам’яті комп’ютера. По-друге, із великого переліку алгоритмів на графах вибрати типові ключові алгоритми. Тоді чіткість опису моделі забезпечують вибору серед них найбільш ефективних алгоритмів.

Таблиця 1.1. Лексикографічне впорядкування посилань на алгоритми обробки графів

А	**Л**
АВЛ-дерево	Лексикографічний пошук у ширину
Алгоритм Беллмана — Форда	**М**
Алгоритм Борувки	Метод найближчого сусіда
Алгоритм Гопкрофта — Карпа	Мінімакс
Алгоритм Дейкстри – Шолтена	Модель Барабаші — Альберт
Алгоритм Джонсона	Модель вкладених множин
Алгоритм Дініца	**Н**
Алгоритм Катхілл — Маккі	Найменший спільний предок
Алгоритм Косараджу	**О**
Алгоритм Крускала	Обхід дерева
Алгоритм Левіта	Офлайновий алгоритм Тарджана для пошуку найменшого спільного предка
Алгоритм пошуку A*	**П**
Алгоритм Прима	Потокова мережа
Алгоритм створення лабіринту	Пошук в глибину з ітеративним заглибленням
Алгоритм Тар'яна	Пошук з обмеженням глибини
Алгоритм Флойда — Воршелла	Пошук за критерієм вартості
Алгоритм Форда — Фалкерсона	Пошук по графу
В	Пошук у глибину
Веретено Мозера	Пошук у ширину
Виродженість (теорія графів)	Пошук циклу у графі
Вкладення графа	
Г	

Граф Бігса — Сміта	Пошук шляху
Граф Фрухта	Алгоритм пошуку D*
Граф Хівуда	**С**
Д	Силові алгоритми візуалізації графів
Алгоритм Дейкстри	Спрощений алгоритм з обмеженням
З	пам'яті
Задача комівояжера	**Т**
Задача листоноші	Теорема Курселя
Задача пошуку ізоморфного	**Х**
підграфа	Хвильовий алгоритм
Задача про максимальний потік	**Ц**
Задача про найдовший шлях	Центральність
Задача про найкоротший шлях	**Ш**
Задача про найширший шлях	Шарнір (теорія графів)
Задача про незалежну множину	**В**
Задача про хід коня	Користувач:BilanN/пісочниця
І	**Р**
Ізоморфізм графів	PQ-дерево
К	Категорії:
Компонента сильної зв'язності	Алгоритми
графа	Теорія графів

Висновки до розділу 1

Переважна більшість прикладних задач зводиться до пошуку маршруту на графах. Чіткість та коректність формування схеми та опису даних при побудові моделі за допомогою у графу забезпечують ефективність вибору та використання оптимального алгоритму. У теоретико-графових термінах

формулюється і моделюється багато прикладних задач, що пов’язані з дискретними даними. Такі задачі виникають, зокрема, при проектуванні схем управління підприємствами, досліджені автоматичних пристроїв, в логічних дослідженнях поведінки соціальних груп, блок-схем програмних продуктів тощо.

Разом з тим, розгляд лінійних маршрутів та циклів у графах є пріоритетним, ключовим напрямком використанням структури графів для вирішення багатьох прикладних задач, що пов'язані з оптимізацією маршрутів і виявленням зв'язків у системах.

РОЗДІЛ 2 ОПТИМАЛЬНІ ШЛЯХИ В ПОТОКОВІЙ МЕРЕЖІ

2.1 Теоретичні аспекти алгоритмів максимального потоку

В багатьох прикладних задачах, пов’язаних з використанням мережі, зазвичай, доводиться враховувати її обмежені можливості. Тоді виникає потреба визначати максимально допустимий потік у мережі. Зокрема, такі задачі характерні для оптимізації транспортних потоків, визначені маршрутів у комп'ютерних мережах та при розподілі енергії в енергетичних мережах.

Алгоритми пошуку максимального потоку допомагають у логістиці визначати оптимальний маршрут для перевезення вантажів з мінімальними затратами часу чи ресурсів. У комп'ютерних мережах такі алгоритми дозволяють забезпечити балансування навантаження та ефективне використання каналів зв'язку. Такі алгоритми також використовуються в задачах планування раціонального використання ресурсів, наприклад, для оптимізації використання виробничих потужностей.

Таким чином, задача відшукання максимального потоку в мережі є однією з основних задач в теорії графів і має численні практичні застосунки. Дослідження алгоритмів максимального потоку надає можливість краще зрозуміти основи теорії графів та їх практичного застосування.

Виникає потреба проаналізувати основні алгоритми пошуку максимального потоку, їх теоретичні аспекти та сфери можливого застосування. Серед найбільш загальних алгоритмів можна назвати алгоритми пошуку в глибину чи ширину, Форда-Фалкерсона, Едмондса-Карпа, Дейкстри, проштовхування предпотоку, Беллмана-Форда та знаходження потоку мінімальної вартості.

Кожен з них має свої унікальні особливості і застосування в різних сценаріях, наприклад:

- алгоритм пошуку в глибину (DFS): використовується для обходу графа, він допомагає виявити деякий маршрут, рухаючись по якому можна обійти послідовно всі вершини графа, які доступні з початкової вершини; можна застосовувати, зокрема, при навігації в місті або плануванні маршруту;
- алгоритм пошуку в ширину (BFS): також використовується для обходу графа, проте спочатку відвідує всі сусідні вершини поточної вершини перед тим, як переходити до наступної, тобто знаходиться шлях, що містить найменшу кількість ребер; можна застосовувати, зокрема, для маршрутизації в комп’ютерних мережах;
- алгоритм Крускала і алгоритм Прима: обидва використовуються для знаходження мінімального остовного дерева у зваженому графі; можна застосовувати, зокрема, для оптимізації маршрутів, для проектування ефективних дорожних мереж;
- алгоритм Дейкстри і алгоритм Беллмана-Форда: використовуються для знаходження найкоротшого шляху між двома вершинами у зваженому графі; можна застосовувати, зокрема, при плануванні автомобільних і авіа-маршрутів, в протоколах маршрутизації;
- алгоритм Форда-Фалкерсона – алгоритм, який широко використовується для пошуку максимального потоку в мережах з метою ефективного знаходження шляхів доповнення із малою кількістю ітерацій. Він спирається на три ключові концепції: залишкові мережі, доповнюючі шляхи та розрізи. Він використовує метод пошуку в глибину або ширину для визначення шляху з невичерпаною пропускною здатністю. Алгоритм працює до тих пір, поки існує доповнюючий шлях, що дозволяє збільшити потік. Основна

ідея алгоритму полягає в поступовому збільшенні потоку шляхом знаходження ланцюгів з залишковою пропускною здатністю. Однією з ключових особливостей алгоритму є його залежність від вибору доповнюючих шляхів. Хоча теоретично алгоритм може працювати нескінченно довго на графах з дійсними значеннями пропускних здатностей, на практиці він часто застосовується разом з методом BFS, що значно прискорює роботу.

- Алгоритм Едмондса-Карпа є покращеною версією алгоритму Форда-Фалкерсона, яка використовує метод пошуку в ширину (BFS) для знаходження найкоротших доповнюючих шляхів. Це дозволяє зменшити часову складність до O(V * E^2). Ідея алгоритму полягає в тому, щоб кожного разу знаходити найкоротший доповнюючий шлях і збільшувати потік вздовж цього шляху. Алгоритм застосовується для розв'язання задач з великими графами, де оптимізація часу є критичною. Однією з переваг є детермінованість та гарантоване знаходження оптимального рішення за кінцеву кількість ітерацій.
- Метод проштовхування предпотоку ґрунтується на використанні висоти вершин та надлишкового потоку. Основна ідея полягає у тому, щоб проштовхувати потік з переповнених вершин до сусідніх вершин з меншою висотою. Операції підйому та проштовхування забезпечують ефективне усунення надлишків. Метод особливо ефективний на щільних графах, де інші алгоритми можуть бути менш продуктивними. Він демонструє добру продуктивність завдяки одночасному обробленню кількох переповнених вершин.
- Потік мінімальної вартості спрямований на мінімізацію вартості потоку при дотриманні обмежень на пропускну здатність. Він використовується в задачах оптимізації логістичних мереж та

управління ресурсами. Основний підхід базується на комбінуванні методів максимального потоку з мінімальною вартістю шляхів.

Кожен з зазначених алгоритмів має свої переваги та обмеження, що спонукає вибирати оптимальний алгоритм в залежності від конкретної прикладної задачі.

2.2 Оптимальні маршрути в стаціонарних задачах

2.2.1 Алгоритм Форда-Фалкерсона для пошуку максимального потоку

Алгоритм Форда-Фалкерсона є досить популярним алгоритмом для знаходження максимального потоку в потоковій мережі. Форд і Фалкерсон запропонували його вперше у 1956 році. Пізніше з’явилися покращені версії алгоритму, зокрема методи, які працюють швидше на великих графах.

Передісторія появи алгоритму була наповнена результатами кропітких операційних досліджень, що мотивовані цілим рядом актуальних оптимізаційних прикладних задач. Виділялися, зокрема, задачі пошуку маршруту в різних областях військового та економічного характеру, що гостро стояли як під час війни, так і в повоєнний відбудовчий період.

Задача максимального потоку передбачає визначення максимального об’єму потоку (перевезень), який можна здійснити від стартової вершини-джерела до фінішної вершини-приймача в зваженому графі з врахуванням обмежень пропускної здатності на дугах (ребрах).

Алгоритм використовує жадібний підхід, поступово збільшуючи потік доти, доки не буде знайдено більше шляхів для його збільшення. Жáдібний алгори́тм — це алгоритм, який спочатку визначає допустимий розв’язок, а потім шукає кращий розв’язок, виходячи з наявних на кожному етапі даних.

Цей алгоритм став основою для багатьох подальших досліджень у теорії потоків у мережах і досі використовується в різних сферах, зокрема, в:

- моделюванні транспортних систем;
- розподілі ресурсів;
- комп’ютерних мережах;
- теорії ігор;
- машинному навчанні.

Нехай маємо зважений граф, в якому вага ребра позначає пропускну здатність між вершинами. Потрібно знайти максимальний потік, який можна пропустити з деякої стартової вершини *s* (витоку) у деяку фінішну вершину *f* (стік).

Крок 0. Введемо параметр *i*, який визначає кількість циклічних ітерацій алгоритму, поклавши спочатку i=1. Введемо також масив змінних m_i і покладемо m_1=0.

Крок 1. Знаходимо який-небуть можливий шлях від вершини *s* до вершини *f*. Можна скористатися, наприклад, алгоритмом в глибину.

Якщо такого шляху не існує (граф не зв’язний), то завершуємо цикл і переходимо на Крок 6.

Крок 2. В одержаному маршруті визначимо ребро з найменшою вагою. Його вагу необхідно присвоїти змінній m_i.

Крок 3. В одержаному маршруті вагу ребер зменшуємо на величину m_i.

Крок 4. Вилучаємо всі ребра з нульовою вагою (їх може виявитися декілька).

Крок 5. Збільшуємо параметр *i* на одиницю та переходимо на Крок 1.

Крок 6. Максимальний потік з пункту *s* у пункт *f* визначається як сума потоків, що виходять із вершини *s* і входять у вершину *f* , та чисельно дорівнює сумі параметрів m_i.

Маршрути, які забезпечують цей потік, визначаються через сукупність маршрутів, побудованих на кроці 1.

Проілюструємо роботу алгоритму на наступній задачі. Припустимо, що керівництво Південно-Західної залізниці вирішило визначити можливість відправки додаткових потягів з Києва до Хмельницького. Інтернет джерела надають різні схеми Південно-Західної залізниці, одна із яких подана на рисунку 2.1.

Побудуємо спочатку граф-модель сполучення між тими містами, відкинувши тупикові та безперспективні маршрути (наприклад, через Жмеринку на південь).

Пронумеруємо вершини графа у такій послідовності:

s – Київ.

1 – Коростень;

2 – Фастів;

3 – Новоград-Волинський;

4 – Житомир;

5 – Козятин;

6 – Шепетівка;

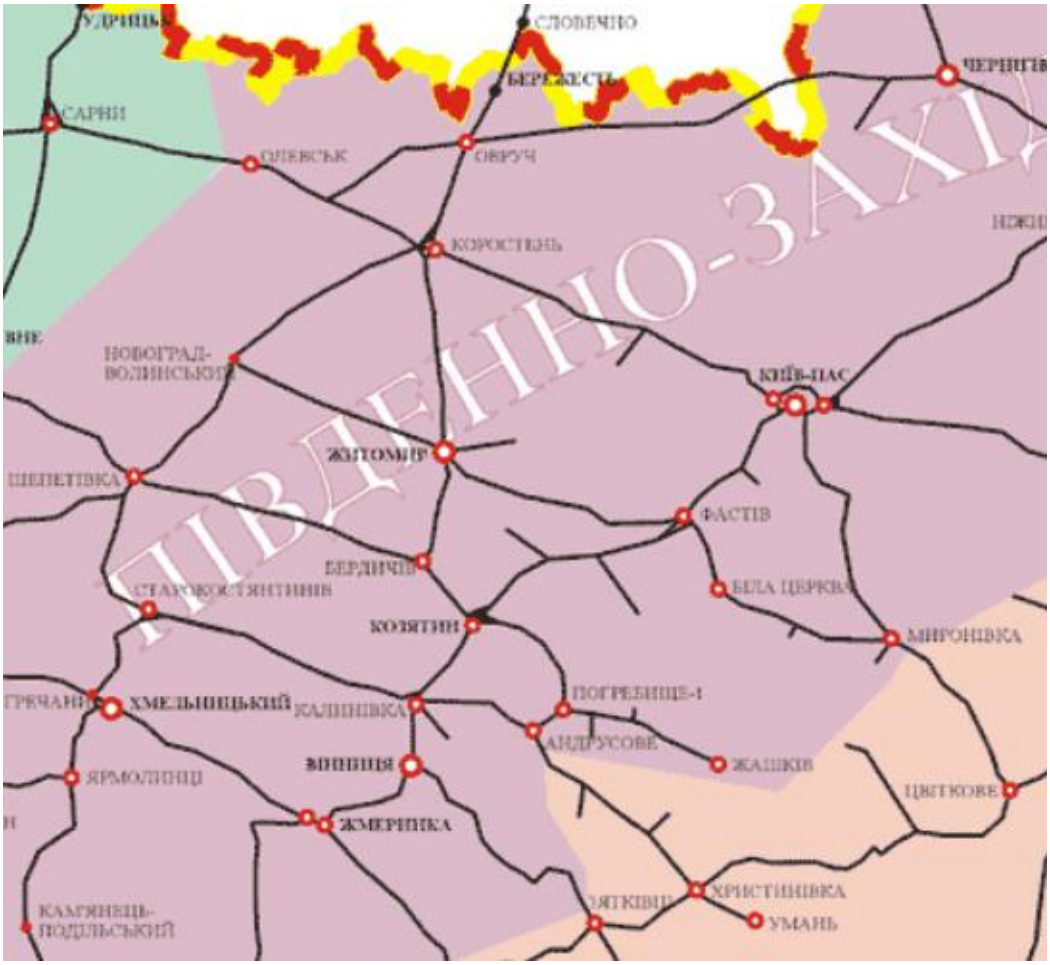

Рисунок 2.1. Схема сполучень Південно-Західної залізниці

7 – Старокостянтинів;

8 – Бердичів;

9 – Калинівка;

f – Хмельницький.

Деякі вузлові станції, а саме: Вінниця, Жмеринка, не включаємо у схему, оскільки вони не впливають на перерозподіл потоків. Вагу ребер (кількість можливостей проведення додаткових потягів) заповнюємо довільними значеннями.

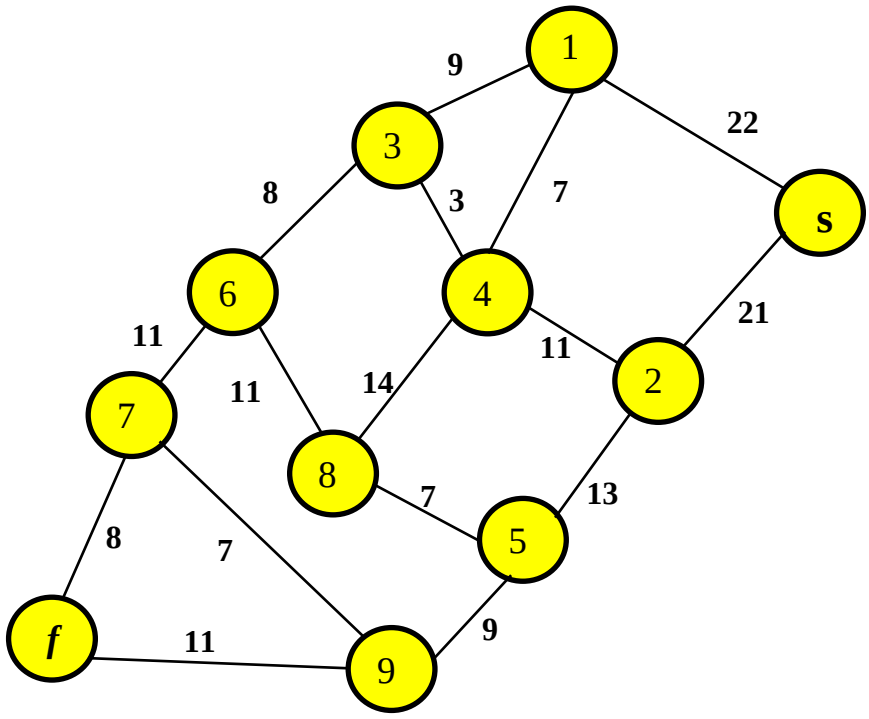

Рисунок 2.2. Граф-модель залізничного сполучення між містами Київ і Хмельницький

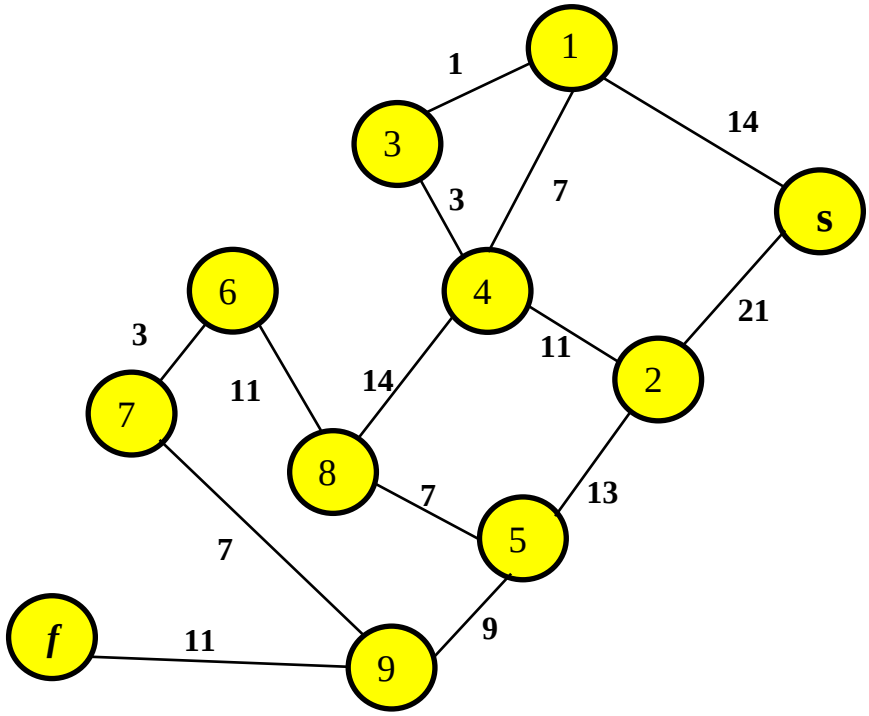

Рисунок 2.3. Граф-модель після першої ітерації

На першому циклі ітераційної програми побудуємо, наприклад, маршрут: *s*, 1, 3, 6, 7, *f* (крок 1). В цьому маршруті мінімальна вага ребер становить 8. Покладаємо $m_1 = 8$ (крок 2). В побудованому маршруті вагу ребер зменшити на 8 (крок 3). Вилучити всі ребра з нульовою вагою. Їх виявитися два (крок 4). В результаті одержимо наступний граф (див. рисунок 2.3).

Збільшуємо параметр *i* на одиницю та переходимо на Крок 1.

На другому циклі ітераційної програми побудуємо, наприклад, маршрут: *s*, 1, 3, 4, 8, 6, 7, 9, *f* (крок 1). В цьому маршруті мінімальна вага ребер становить 1.

Зауваження 2.1. Ми спеціально вибрали «нехороший» маршрут (дуже довгий та ще й з мізерним потоком). Цим самим ми підкреслюємо недоліки алгоритму Форда-Фалкерсона.

Покладаємо $m_2 = 1$ (крок 2). В побудованому маршруті вагу ребер зменшити на 1 (крок 3). Вилучити єдине ребро з нульовою вагою (крок 4). В результаті одержимо наступний граф

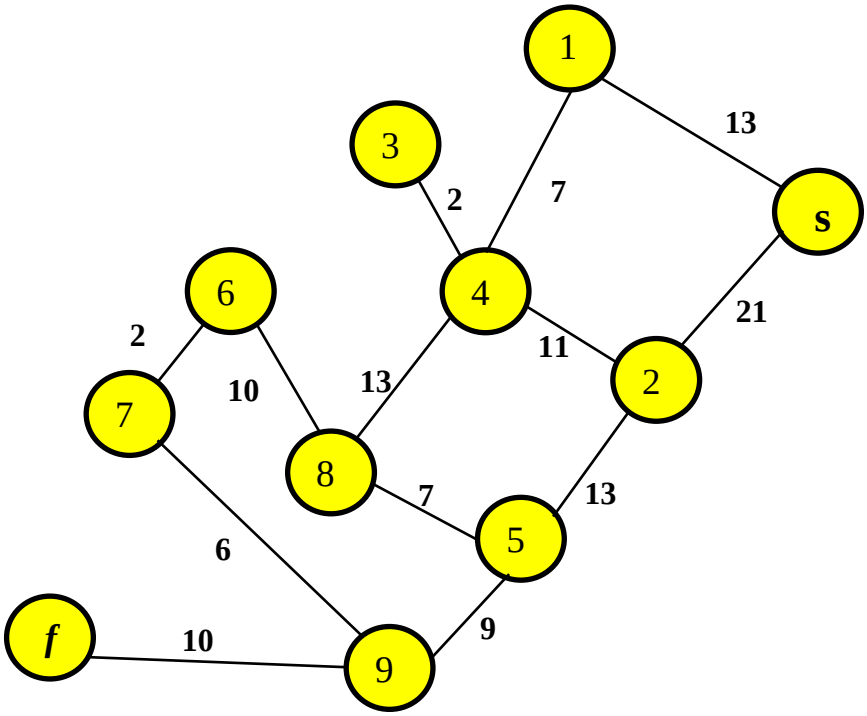

Рисунок 2.4. Граф-модель після другої циклічної ітерації

Збільшуємо параметр *i* на одиницю та переходимо на Крок 1.

На третьому циклі ітераційної програми побудуємо, наприклад, маршрут: *s*, 1, 4, 8, 6, 7, 9, *f* (крок 1). В цьому маршруті мінімальна вага ребер становить 2. Покладаємо $m_3 = 2$ (крок 2). В побудованому маршруті вагу ребер зменшити на 2 (крок 3). Вилучити єдине ребро з нульовою вагою (крок 4). В результаті одержимо новий граф. Збільшуємо параметр *i* на одиницю та переходимо на Крок 1.

Як зауважено вище, на другому циклі вибраний маршрут є допустимим, але не раціональним. Те саме можна сказати і про третій цикл. Очевидно, що не раціонально вибирати на четвертому циклі ітераційної програми маршрут: *s*, 1, 4, 8, 5, 9, *f* (крок 1). Краще обрати маршрут *s*, 2, 5, 9, *f*. В цьому маршруті мінімальна вага ребер становить 8. Покладаємо $m_4 = 8$.

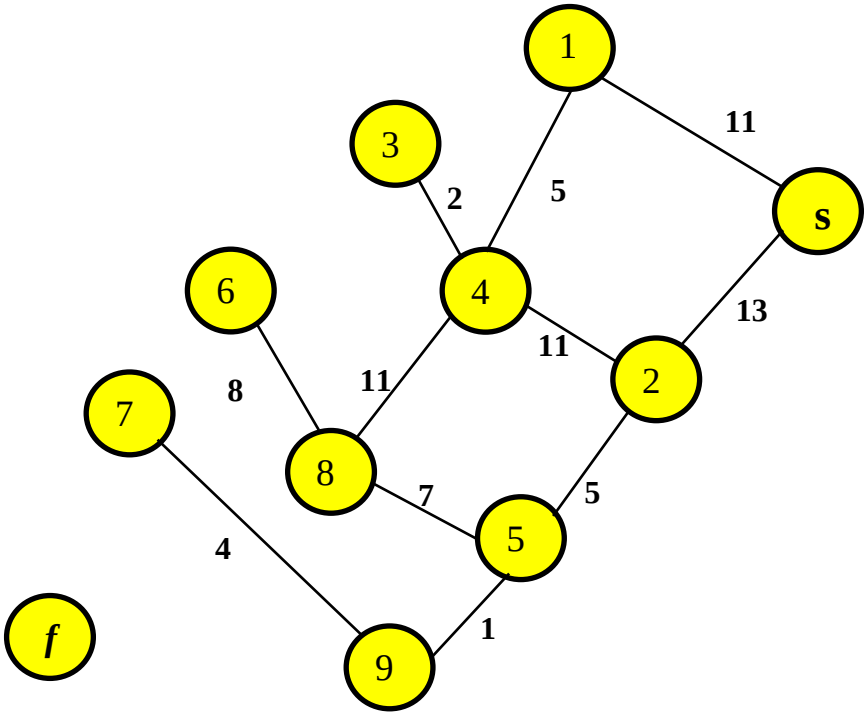

Рисунок 2.5. Граф-модель після четвертої циклічної ітерації

В побудованому маршруті вагу ребер зменшити на 8 (крок 3). Вилучити єдине ребро з нульовою вагою (крок 4). В результаті одержимо новий граф (див рисунок 2.5), в якого фінішна вершина виявилася ізольованою. Більше не можливо побудувати маршрутів від стартової вершини до фінішної.

Переходимо на Крок 6.

В результаті одержимо, максимальний потік від вершини *s* до вершини *f* становить $m_1 + m_2 + m_3 + m_4 = 8 + 1 + 2 + 8 = 11$ одиниць.

Зауваження 2.2. Взагалі кажучи, алгоритм Форда-Фалкерсона не лише визначає максимальний об'єм потоку, але й встановлює допустимий варіант його одержання, інша справа, що цей варіант не завжди є раціональним. Тому розробляються інші алгоритми пошуку максимального потоку з різними показниками оптимальності (найкоротші маршрути, рівномірна завантаженість тощо).

Зауваження 2.3. Розглядаючи попередній приклад, було підкреслено, що вибрані на першому кроці допустимі маршрути з стартової вершини у фінішну можуть бути «нехороший», але алгоритм Форда-Фалкерсона не звертає на це увагу. Разом з тим така незалежність у виборі може призвести до невірного розв'язку. Якщо в графі (див. рисунок 2.6) вибрати маршрут 1, 2, 3, 4, то після вилучення ребер (крок 4) прийдемо до не зв'язного графу і виходить, що максимальний потік становить 7 одиниць. Якщо ж вибрати спочатку маршрут 1, 2, 4, то на наступних етапах ще будуть маршрути 1, 3, 3, 4 та 1, 3, 4 і максимальний потік буде 21. Отже, алгоритм Форда-Фалкерсона може видати хибний результат.

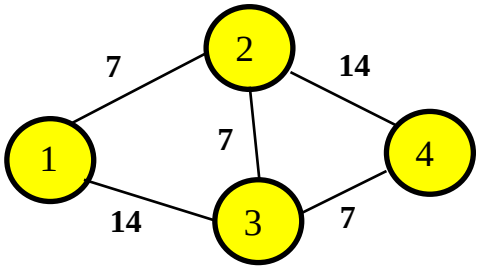

Рисунок 2.6. Приклад хибного розв'язку.

Зауваження 2.4. На прикладі залізничної мережі можна розглянути специфіку алгоритму Форда-Фалкерсона у варіанті можливого зустрічного, реверсного потоку. Якщо станції пов'язані так, як показано на рисунку 2.7, то

вибір маршрутів 1, 2, 3, 4 і 1, 3, 2, 4 спонукає до реверсного використання ребра (2, 3). Використання реверсного потоку вимагає додаткових витрат часу на зміну напрямку руху і потрібно враховувати, що в протилежному напрямку може пройти уже не 7, а менше потягів.

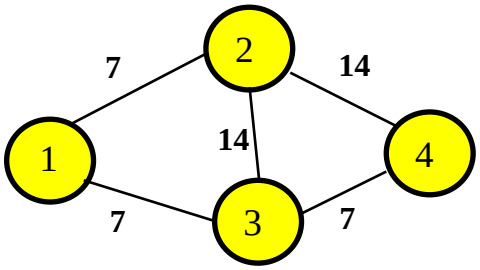

Рисунок 2.7. Приклад використання реверсних потоків.

2.2.2 Алгоритм Едмондса-Карпа для пошуку максимального потоку

Алгоритм Едмондса-Карпа є покращеною версією класичного алгоритму Форда-Фалкерсона. Його було запропоновано в 1972 році Джеком Едмондсом і Річардом Карпом. Головна ідея цього методу полягає у виборі на кожному кроці найкоротшого (за кількістю ребер) допустимого шляху з вершини-джерела до вершини-приймача. Такий підхід дозволяє уникнути нераціональних маршрутів, що можуть призвести до зниження ефективності, як це інколи трапляється в класичному варіанті Форда-Фалкерсона.

Основна ідея алгоритму:

Замість того, щоб шукати довільний допустимий шлях у залишковій мережі (як у класичному алгоритмі Форда-Фалкерсона), Едмондс і Карп запропонували використовувати **пошук у ширину (BFS)** для знаходження найкоротшого шляху з s до f. Це зменшує кількість ітерацій і дозволяє уникнути «поганих» маршрутів із малим потоком та великою довжиною.

Формалізований опис роботи алгоритму:

Крок 0. Встановлюємо початковий потік у всіх ребрах графа на нуль. Формуємо залишкову мережу.

Крок 1. Використовуючи пошук у ширину (BFS), знаходимо найкоротший допустимий шлях із вершини s до вершини f у залишковій мережі.

Якщо такого шляху не існує — завершуємо алгоритм (перехід на Крок 6).

Крок 2. Знаходимо мінімальну пропускну здатність уздовж знайденого шляху — позначимо її як *mi*.

Крок 3. Зменшуємо пропускні здатності всіх ребер на шляху на *mi* і збільшуємо зворотні ребра (реверсні потоки) на *mi*. Це дає змогу алгоритму «відкочувати» неправильні вибори на попередніх етапах.

Крок 4. Повторюємо Кроки 1–3 до тих пір, поки існує шлях у залишковій мережі від s до f.

Крок 5. Сумуємо усі *mi* — вони і дають максимальний потік.

Крок 6. Завершення. Повертаємо сумарний максимальний потік. Також можна зберегти маршрути, через які цей потік було реалізовано.

Приклад використання

Уявімо, що необхідно оптимізувати потік товарів між центральними складами у Києві та Хмельницькому, як і в попередньому прикладі (див. рисунок 2.2). Тепер використаємо алгоритм Едмондса-Карпа. Основна відмінність — на кожному етапі будемо шукати найкоротший шлях за кількістю станцій (вузлів), а не просто довільний маршрут.

Наприклад, на першій ітерації алгоритм вибере найкоротший шлях:

s → 2 → 5 → 9 → f,

якщо пропускні здатності дозволяють, і обчислить мінімальний потік уздовж цього шляху.

На другій ітерації буде обрано наступний найкоротший доступний шлях (наприклад, через s → 1 → 3 → 6 → 7 → f) — уже з урахуванням змінених залишкових можливостей.

Таким чином, алгоритм ефективно розподіляє потік, мінімізуючи кількість кроків і уникаючи «пасток» у вигляді довгих і неефективних маршрутів, що можуть зменшити загальний потік або призвести до передчасної ізоляції фінішної вершини.

Зауваження 2.5. Алгоритм Едмондса-Карпа гарантує оптимальний результат, незалежно від черговості маршрутів, на відміну від класичного Форда-Фалкерсона. Використання BFS забезпечує завжди найкоротший шлях на кожній ітерації, що запобігає потенційним помилкам у побудові потоку.

Зауваження 2.6. Використання реверсних ребер у залишковій мережі дозволяє переглядати попередні рішення і коригувати потік — це важлива перевага алгоритму.

Зауваження 2.7. Алгоритм зберігає допустимі маршрути для кожної одиниці потоку, що може бути корисно для аналізу або подальшого моделювання (наприклад, в логістиці або проєктуванні мережевих структур).

2.2.3 Потік мінімальної вартості

У багатьох реальних задачах недостатньо просто знайти максимальний потік — важливо також мінімізувати сумарну вартість доставки потоку з джерела у стік. Наприклад, у транспортних, енергетичних чи логістичних задачах часто виникає необхідність перевезти певну кількість ресурсу з найменшими витратами.

Постановка задачі

Нехай маємо орієнтований граф, де:

- кожне ребро (u, v) має пропускну здатність c(u, v),
- вартість одиниці потоку cost(u,v),
- джерело потоку — вершина s,
- приймач — вершина f,

- потрібно передати з вершини s у вершину f потік заданого об’єму d (або максимального можливого) з мінімальними загальними витратами.

Алгоритм розв’язання

Найпоширеніші методи знаходження потоку мінімальної вартості:

1. Алгоритм послідовних найкоротших шляхів:
   - Повторюється знаходження найкоротшого (найдешевшого) шляху у залишковому графі.
   - По ньому відправляється можливий потік (не більше пропускної здатності).
   - Повторюється доти, доки потрібно передати весь потік.
2. Алгоритм потенціалів:
   - Змінює ваги, щоб уникнути негативних вартостей і прискорити пошук найкоротших шляхів.
   - Відомий як алгоритм з потенціалами або алгоритм Джонсона.
3. Алгоритм видалення циклів від’ємної вартості:
   - Після досягнення допустимого потоку перевіряє, чи існують цикли з негативною вартістю у залишковому графі.
   - Якщо такі є, то покращує розв’язок, пропускаючи потік по циклу у зворотному напрямі.

Приклад

Розглянемо простий граф з трьома вершинами:

- s→as \rightarrow as→a: пропускна здатність = 5, вартість = 2
- a→fa \rightarrow fa→f: пропускна здатність = 5, вартість = 1
- s→fs \rightarrow fs→f: пропускна здатність = 3, вартість = 5

Потрібно передати 7 одиниць потоку з s до f з мінімальною вартістю.

Розв’язання:

1. Через s→a→fs \rightarrow a \rightarrow fs→a→f можна передати 5 одиниць потоку з вартістю 5·(2+1)=155 \cdot (2+1) = 155·(2+1)=15.

2. Через s→fs \rightarrow fs→f передаємо ще 2 одиниці з вартістю 2·5=102 \cdot 5 = 102·5=10.

Загальна вартість:

15+10=2515 + 10 = 2515+10=25

Застосування на практиці

Задачі потоку мінімальної вартості застосовуються в:

- логістиці — мінімізація вартості доставки товарів між складами та магазинами;
- енергетичних системах — передача електроенергії з мінімальними втратами;
- телекомунікаціях — передача даних через канали з різною вартістю;
- оптимальному призначенні — задачі розподілу ресурсів між агентами з урахуванням витрат.

Зауваження 2.8

Існування потоку заданої величини з мінімальною вартістю не завжди гарантоване — наприклад, якщо у графі немає шляхів потрібної пропускної здатності. Також важливо враховувати від’ємні вартості ребер, які можуть спричинити від’ємні цикли, що потребують додаткової обробки (наприклад, методом потенціалів).

Зауваження 2.9

Якщо вартість кожного ребра дорівнює одиниці, то задача потоку мінімальної вартості зводиться до пошуку найкоротших маршрутів у графі, які передають задану кількість потоку. У такому випадку застосовуються модифікації BFS або Dijkstra.

Висновки до розділу 2

Задача встановлення максимально потоку в мережі виникає при наявності обмежень на пропускну здатність каналів зв’язку. Прикладом таких моделей

можуть служити транспортні мережі, комп'ютерні системи передачі даних, використання ресурсів виробничих потужностей тощо.

Серед найбільш загальних алгоритмів можна назвати алгоритми пошуку в глибину чи ширину, Форда-Фалкерсона, Едмондса-Карпа, Дейкстри, проштовхування предпотоку, Беллмана-Форда та знаходження потоку мінімальної вартості.

Дослідження алгоритмів максимального потоку надає можливість краще зрозуміти нюанси їх практичного застосування, зокрема, не лише принципи роботи алгоритмів, але й умови їх застосування, переваги та обмеження у використанні.

РОЗДІЛ 3 ПРАКТИЧНА РОБОТА З ДОСЛІДЖЕННЯ ОПТИМІЗАЦІЙНИХ АЛГОРИТМІВ

3.1 Інструкція використання 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.

3.1.1 Демонстрація веб-додатка

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

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

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

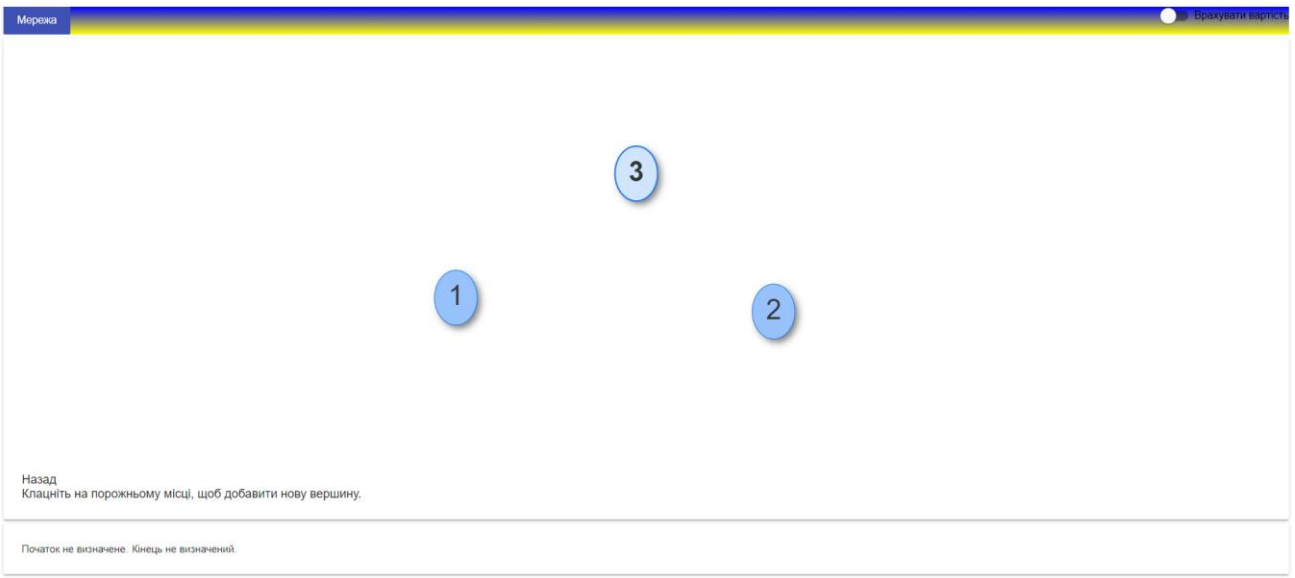

Рисунок 3.1 Вигляд робочого вікна додатку на етапі розміщення вершин

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.

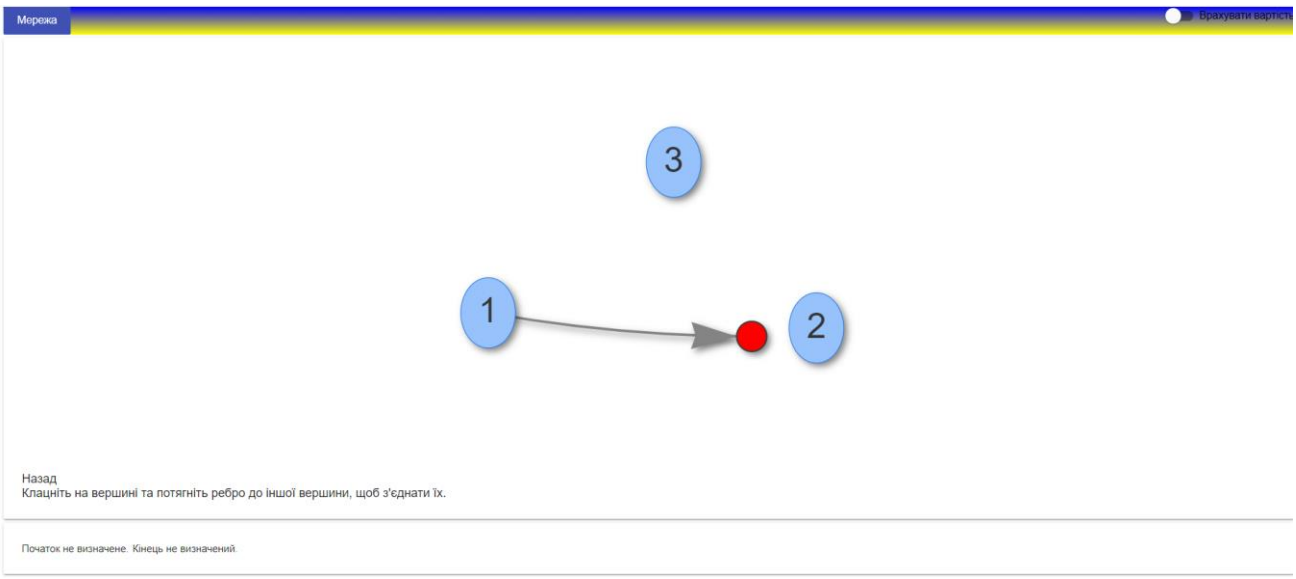

Рисунок 3.2 Вигляд робочого вікна додатку на етапі розміщення ребер

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

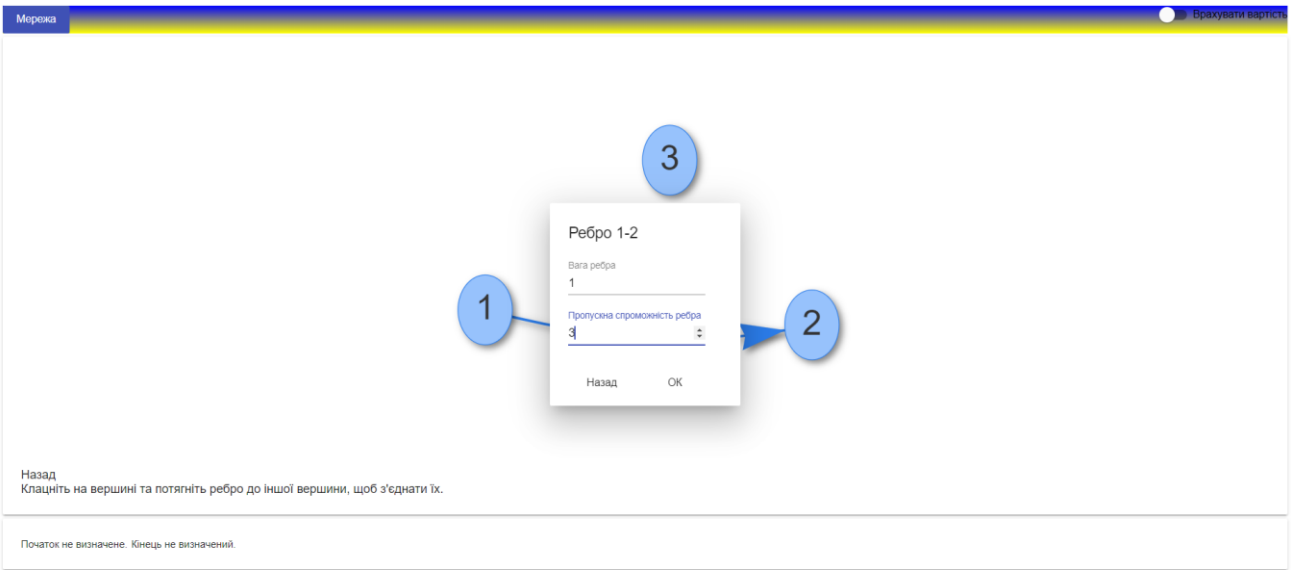

Рисунок 3.3 Вигляд робочого вікна додатку на етапі встановлення ваги ребер

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.

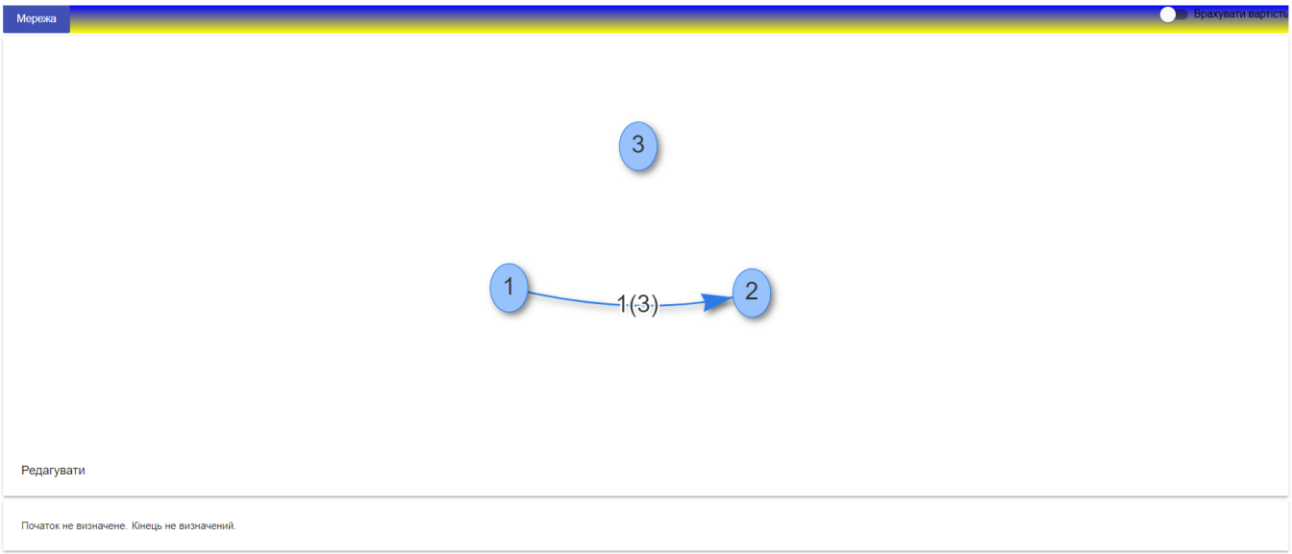

Рисунок 3.4 Вигляд робочого вікна додатку по завершенню редагування графа

3.1.2 Застосування алгоритмів

Розглянемо роботу програмного продукту для дослідження алгоритму "Проштовхування передпотоку".

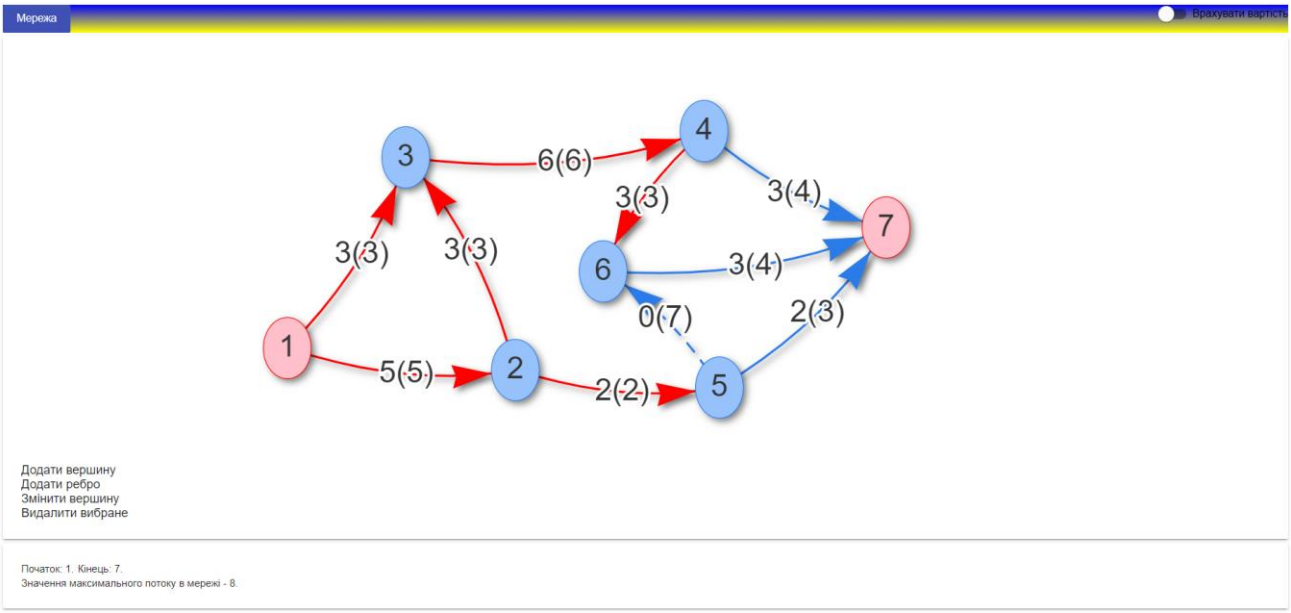

Рисунок 3.5. Вигляд робочого вікна при дослідженні алгоритму "Проштовхування передпотоку"

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".

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

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.

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. Після цього оберіть

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

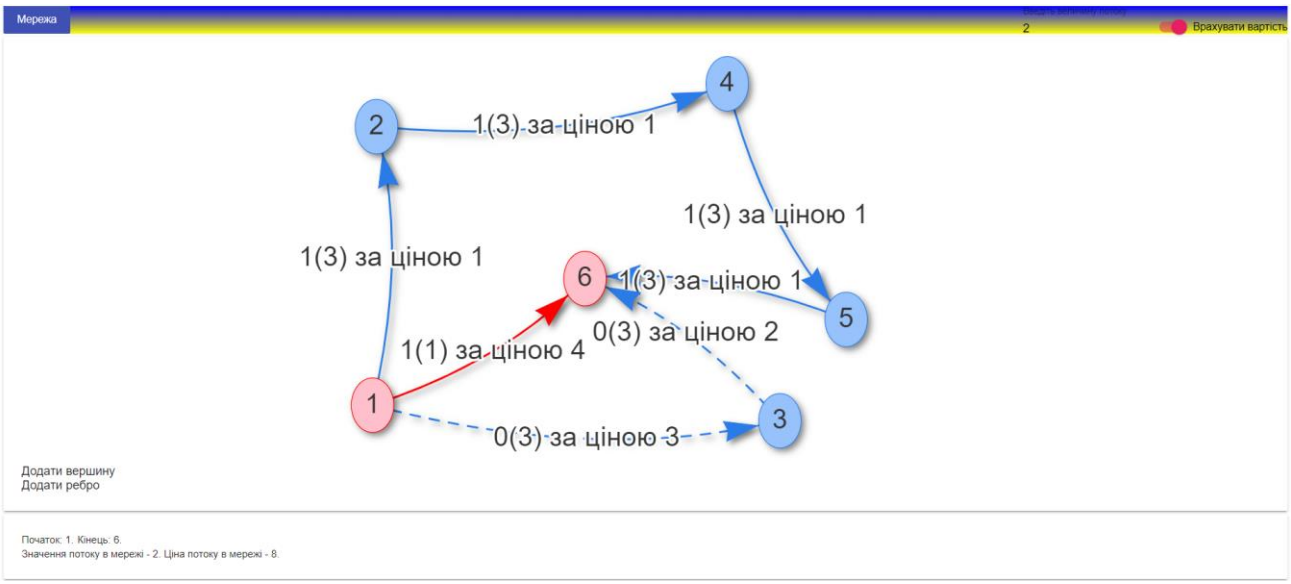

Рисунок 3.6. Вигляд робочого вікна при відшуканні найдешевшого потоку

3.2 Приклад розв’язання задачі за допомогою представленого продукту

Для перевірки коректності роботи розробленого програмного забезпечення було протестовано приклад задачі пошуку максимального потоку, поданий у другому розділі.

З початкової задачі відомо, що мережею є граф, де вершина 1 є джерелом, а вершина 11 — стоком. У другому розділі, під час ручного розв’язання, було отримано значення максимального потоку, яке становить 11. Проте, після побудови мережі в реалізованому додатку та запуску алгоритму Форда-Фалкерсона, було отримано правильне значення максимального потоку — 17, що видно з результату внизу вікна програми.

Це свідчить про наявність помилки в обчисленнях або логіці під час ручного розв’язання задачі, яку вдалося виявити завдяки точності реалізованого алгоритму.

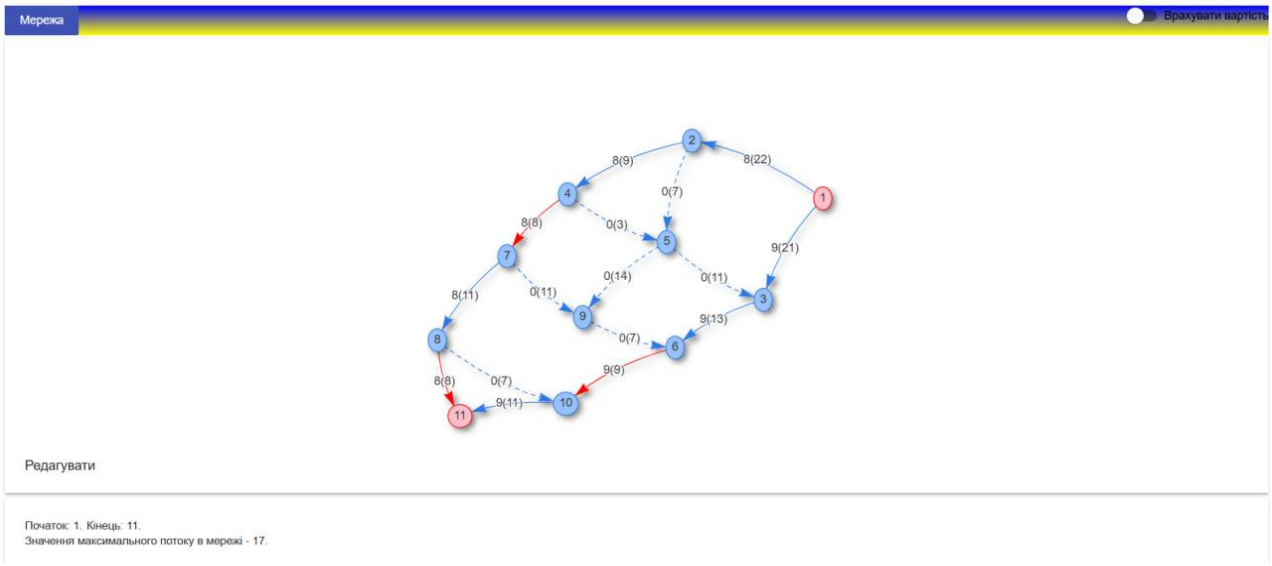

Рисунок 3.7. Візуалізація мережі та результат роботи алгоритму Форда-Фалкерсона

Висновки до розділу 3

Запропонований додаток дозволяє виконувати пошук у заданому потоці алгоритмами дослідження потокової мережі. Він є досить зручним, зрозумілим та ефективним. Звичайно, він, як і будь-який програмний продукт, не є ідеальним і його можна наповнювати рядом додаткових опцій.

ВИСНОВКИ

В даній кваліфікаційній роботі була спроектована та розроблена система дослідження ряду класичних алгоритмів маршрутів в потокових мережах: Форда-Фалкерсона, Едмондса-Карпа, Дейкстри, проштовхування предпотоку, Беллмана-Форда та знаходження потоку мінімальної вартості.

Актуальність таких досліджень обумовлена потребою розв’язання ряду прикладних задач, які моделюються в графових термінах, зокрема, при проектуванні схем управління підприємствами, досліджені автоматичних пристроїв, в логічних дослідженнях поведінки соціальних груп, блок-схем програмних продуктів тощо.

Запропоновано додаток, який дозволяє візуалізувати процес задання графової моделі та отримувати оптимальні рішення досліджуваними алгоритмами. Він надає можливість порівнювати алгоритми роботи з потоковими задачами, демонструвати їх як позитивні сторони, так і недоліки. Це дозволяє використовувати запропонований продукт для навчальних цілей.

Перспективами подальших досліджень може слугувати наповнення продукту рядом додаткових опцій, зокрема, автоматичне формування зображення на основі вагової матриці.

Матеріал кваліфікаційної роботи може бути використаний при розробці програмного забезпечення для розв’язання оптимізаційних задач, а у навчальному процесі при вивченні освітніх компонент «Алгоритми та структури даних», «Дискретні структури», в ряді вибіркових компонент, а також у практичній діяльності фахівців з комп’ютерних наук.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Балога С.І. Дискретна математика. Навчальний посібник. Ужгород : ПП «АУТДОРШАРК», 2021. 124 с.

2. Бартіш М. Я. Дослідження операцій. Ч. 2. Алгоритми оптимізації на графах, 2007. 168 с.

3. Давидюк Н. В. Знаходження найкоротших маршрутів у графах. Робота на здобуття освітнього ступеня «Магістр». Луцьк. 2024. 42 с. https://evnuir.vnu.edu.ua/bitstream/123456789/26491/1/davydiuk_2024.pdf

4. Категорія: Алгоритми на графах. https://uk.wikipedia.org/wiki/Категорія: Алгоритми на графах

5. Катренко А. В. Дослідження операцій: Підручник. Львів: “Магнолія Плюс”, 2004. 549 с.

6. Козаченко Д.М., Вернигора Р.В., Малашкін В.В. Основи дослідження операцій у транспортних системах: приклади та задачі: навчальний посібник для ВНЗ. Дніпропетровськ, 2015. 277 с.

7. Нікольський Ю. В., Пасічник В. В., Щербина Ю. М. Дискретна математика: Підручник. Київ: Видавнича група BHV, 2007. 368 с.

8. Турчина В.А., Гулько К.П. Застосування алгоритму Форда-Фалкерсона для виявлення надлишкової інформації Питання прикладної математики і математичного моделювання. Випуск 19. 2019. С. 175 -181.

ДОДАТОК

Веб-додаток дослідження максимального потоку

```
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;
  }

}
```