КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук і кібернетики
Кафедра дослідження операцій

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
за спеціальністю 113 «Прикладна математика»

на тему:

ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА АЛГОРИТМІВ РОЗВ'ЯЗУВАННЯ ЗАДАЧ ПРО МАКСИМАЛЬНИЙ ПОТІК

студента 4 курсу
Краєвського Олександра Святославовича

Науковий керівник:
доцент, кандидат фізико-математичних наук
Якимів Роман Ярославович

Робота заслухана на засіданні кафедри дослідження операцій та рекомендована до захисту в ЕК, протокол № ..... від .....2021 р.

Завідувач кафедри ДО проф. Іксанов О.М.

Київ-2021

Зміст

Вступ

Вивчення мережевих моделей почалось ще в середині XX століття в зв’язку з появою транспортних задач (задачі, які допомагають перевізникам мінімізувати сумарні витрати при перевезеннях). Але з часом виявилось, що методи, які використовуються при розв’язку такого роду задач, можуть бути використані і для інших мережевих задач (задачі про інформаційні потоки в мережах зв’язку або задачі про дорожні транспортні потоки). Також виявили, що ці методи можуть бути використані для цілого ряду прикладних комбінаторних задач, які ніяк не пов’язані з реально існуючими мережами. Актуальність задач на максимальний потік постійно зростає через розвиток глобальних комунікацій, збільшення користувачів Інтернету, будівництво розмаїтих інфраструктурних мереж і т. д.

Саме через актуальність цієї задачі знання ефективності алгоритму в тій чи іншій ситуації стає дуже нагальним, оскільки воно може забезпечити економію великої кількість ресурсів.

1. Основні поняття та визначення

1.1. Базові терміни

- Граф G = (N, A) складається з ненульової кількості вершин(вузлів) N, та скінченної кількості ребер A, де кожне ребро з'єднує одну вершину з іншою вершиною.
- Нехай e ∈ A — ребро. Кажуть, що ребро e = (v, u) з'єднує вершини v і u, які є його кінцями. Вершини v і u називаються *суміжними*; кожна з них є *інцидентною* ребру *e*.
- Два різних ребра, інцидентних одній і тій самій вершині, називаються *суміжними*.
- *Петля* — це ребро, що з'єднує вершину саму з собою.
- *Кратні ребра*, це такі два або більше ребер, які є інцидентими з двома однаковими вершинами.
- *Простим графом* (*звичайним графом*) називають граф, який не містить в собі кратних ребер та петель.
- Граф, що містить кратні ребра, але не містить петель, називають *мультиграфом*.
- Граф, що обов'язково містить петлі називають *псевдографом.*

Проте будь-який мультиграф або псевдограф може бути перетворений у простий граф шляхом перетворення петель в кратні ребра за допомогою

штучних вершин, а кратні ребра можна переворити в звичайні додавши штучні вершини і розділивши ребра.

Таким чином, надалі ми зможемо розглядати тільки прості (звичайні) графи.

- *Шляхом* з вершини v_1 у вершину v_t (або між вершинами v_1 та v_t) в графі G називають таку послідовність суміжних ребер (v_1, v_2), (v_2, v_3), …, (v_{t-1}, v_t), в якій не повторюється жодне ребро.
- *Простий шлях* — шлях, в якому не повторюються жодне ребро і жодна вершина.
- *Довжина маршруту* — кількість дуг в ньому.
- Граф $G = (N; A)$ називають *зваженим графом*, якщо кожному його ребру призначено число w(e), яке називають *вагою* ребра.
- Якщо ребро має конкретно визначений напрямок, то його називають *дугою*.
- Граф $G = (N; A)$ називають *орієнтованим графом (скорочено орграф)*, якщо всі його ребра є дугами.
- Граф, довільні дві вершини якого можуть бути з'єднані деяким маршрутом (є зв'язаними), називається *зв'язним*. Інакше граф називається *незв'язним*.

Геометрично граф зображується точками (множина вершин *N*) та лініями зі стрілками (множина дуг *A*), що з'єднують деякі пари цих точок. На рисунку 1.1 наведено приклад графічного зображення різних типів графів.

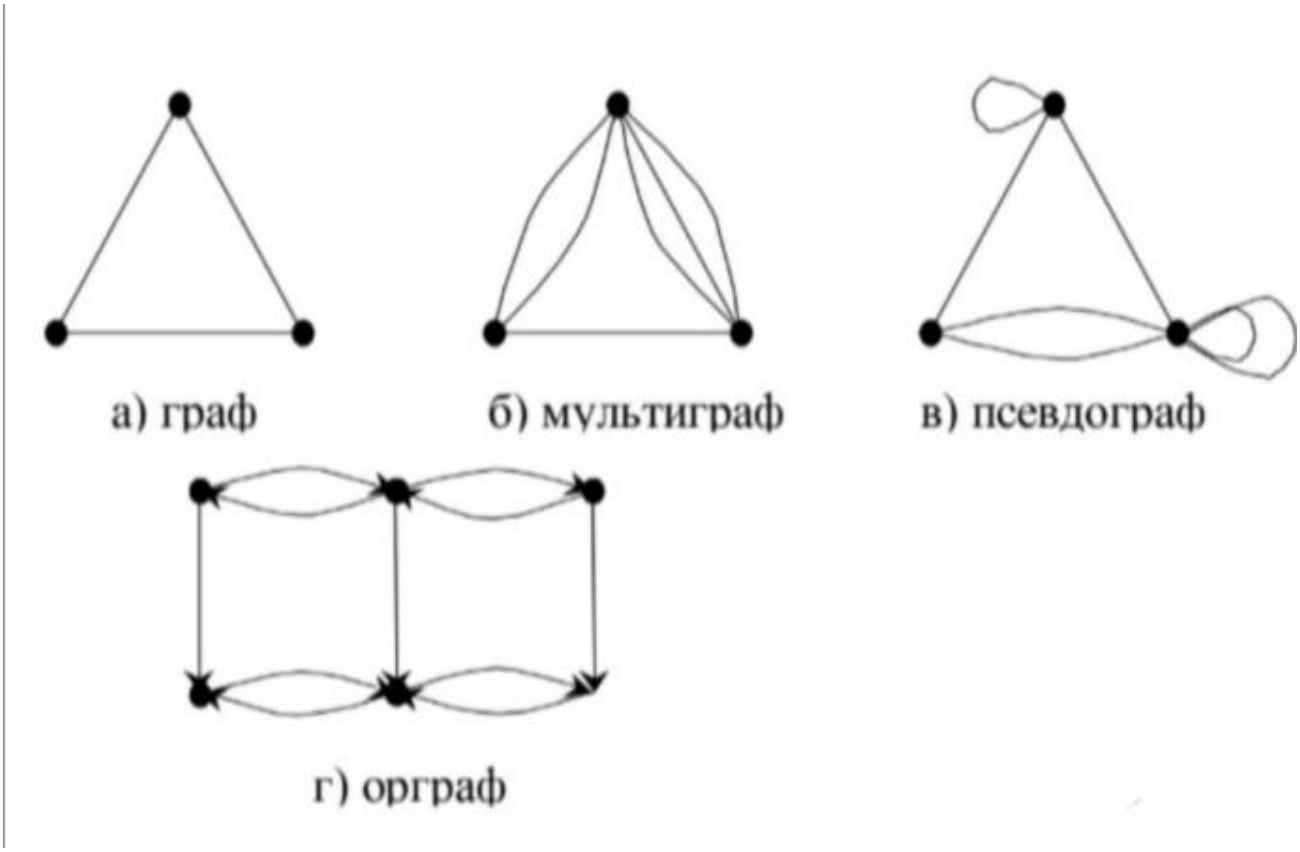


Рис. 1.1

1.2. Мережі та потоки

- *Мережею* називається граф, елементам якого поставлені у відповідність деякі параметри.
- *Елементами графа* вважаються його вершини, дуги, або більш складні конструкції, утворені з вказаних елементарних.
- Мережу можна побудувати таким чином:

1. Кожній вершині $i \in N$ поставим у відповідність число p_i, що називається її інтенсивністю. Вершина i називається джерелом, якщо $p_i > 0$, стоком, якщо $p_i < 0$, і нейтральною, якщо $p_i = 0$.
2. Кожній дузі $(i, j) \in A$ поставимо у відповідність числа u_{ij} та c_{ij}, що називають, відповідно, функцією пропускної здатності (спроможності) та функцією вартості.

- *Потоком* f в G називають дійсну функцію $f: N \times N \rightarrow R$, яка задовольняє умови:
  1. $f(i, j) = -f(j, i)$ (антисиметричність);
  2. $|f(i, j)| \leq u(i, j)$ (обмеження пропускної здатності), якщо ребра немає то $f(i, j) = 0$;
  3. $\sum_v f(i, j) = 0$ для всіх вершин i, окрім випадків коли ця вершина є джерело або стік (закон збереження (неперервності) потоку)

1.3. Графічне представлення мереж та їх перерізів

Наведемо приклад [1] найпростішої мережі з вершинами, дугами і пропускними здатностями дуг (Рис.1.3.1). Дана мережа включає в себе 4 вершини та 4 дуги. У вказаному графі, джерелом є вершина A, а стоком вершина D. Дуга між A та B має пропускну здатність 4, а дуга між A та C має пропускну спроможність 5.

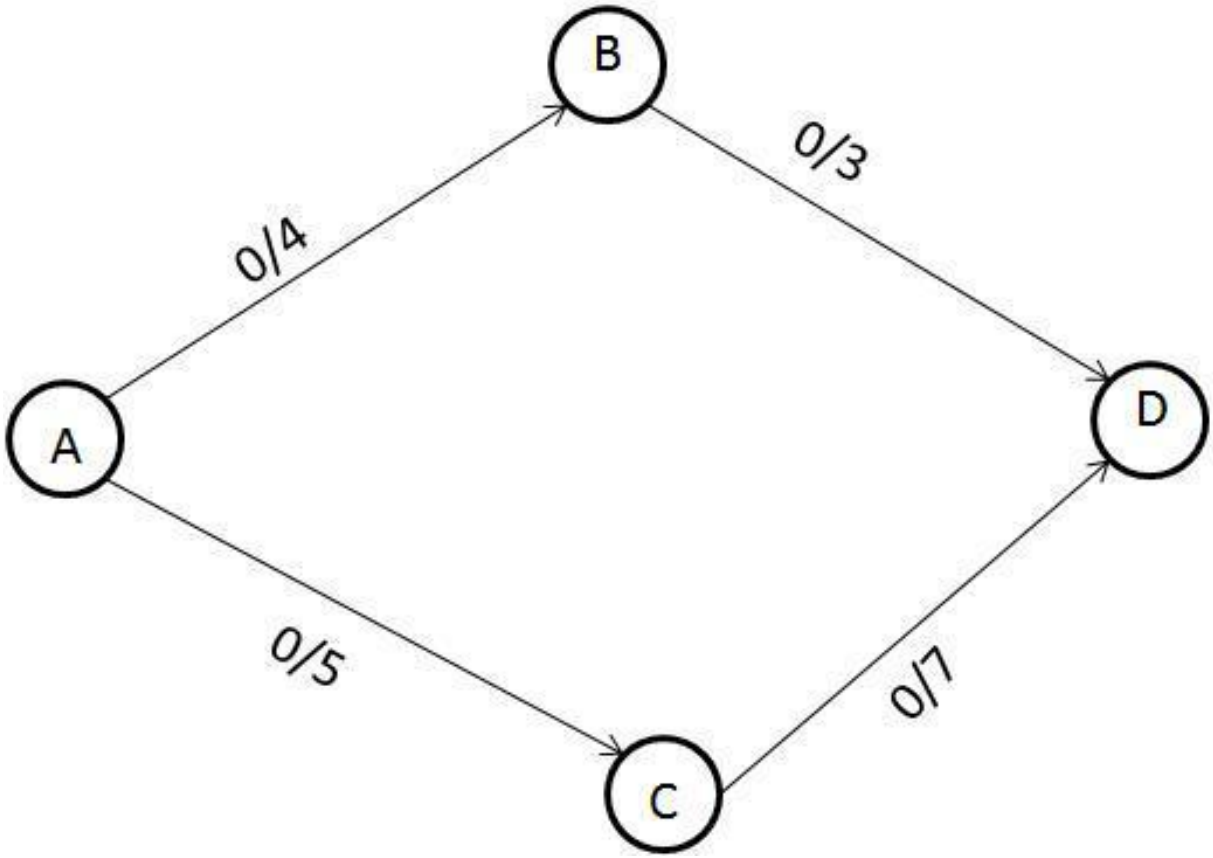


Рис. 1.3.1

Подібним чином, пропускні здатності дуг *(B, D)* та *(C, D)* це $u_{BD} = 3$ та $u_{CD} = 7$, відповідно. Кожна дуга має сталу верхню границю, таку що потік через цю дугу не може бути більшим за неї. Шляхом з вершини ***v*** у вершину ***u*** є почергова послідовність вершин і ребер, яка починається з ***v*** та закінчується в ***u***. В цьому графі максимальний потік реалізується шляхом з *A* через *C* до *D*.

Введемо тепер поняття *перерізу*. *Переріз* [S,$\overline{S}$] розділяє множину вершин *N* на дві підмножини *S* та $\overline{S}$. Він складається зі всіх вершин з однією кінцевою точкою в множині *S*, та іншою в множині $\overline{S}$. Дуги напрямлені з *S* до $\overline{S}$ (записується *(S, $\overline{S}$)*) називають вперед-напрямленими в перерізі. Переріз *[S, $\overline{S}$]*

називається перерізом *s*-*t*, якщо $s \in S$ та $t \in \bar{S}$. $u[S, \bar{S}]$ – це пропускна здатність перерізу $[S, \bar{S}]$, яка визначається як $\sum_{(i,j)\in(S,\bar{S})} u_{ij}$ (це стосується тільки вперед напрямлених дуг). Мінімальний переріз в мережі *G* – це переріз *s*-*t* з мінімальною пропускною здатністю (рис.1.3.2).

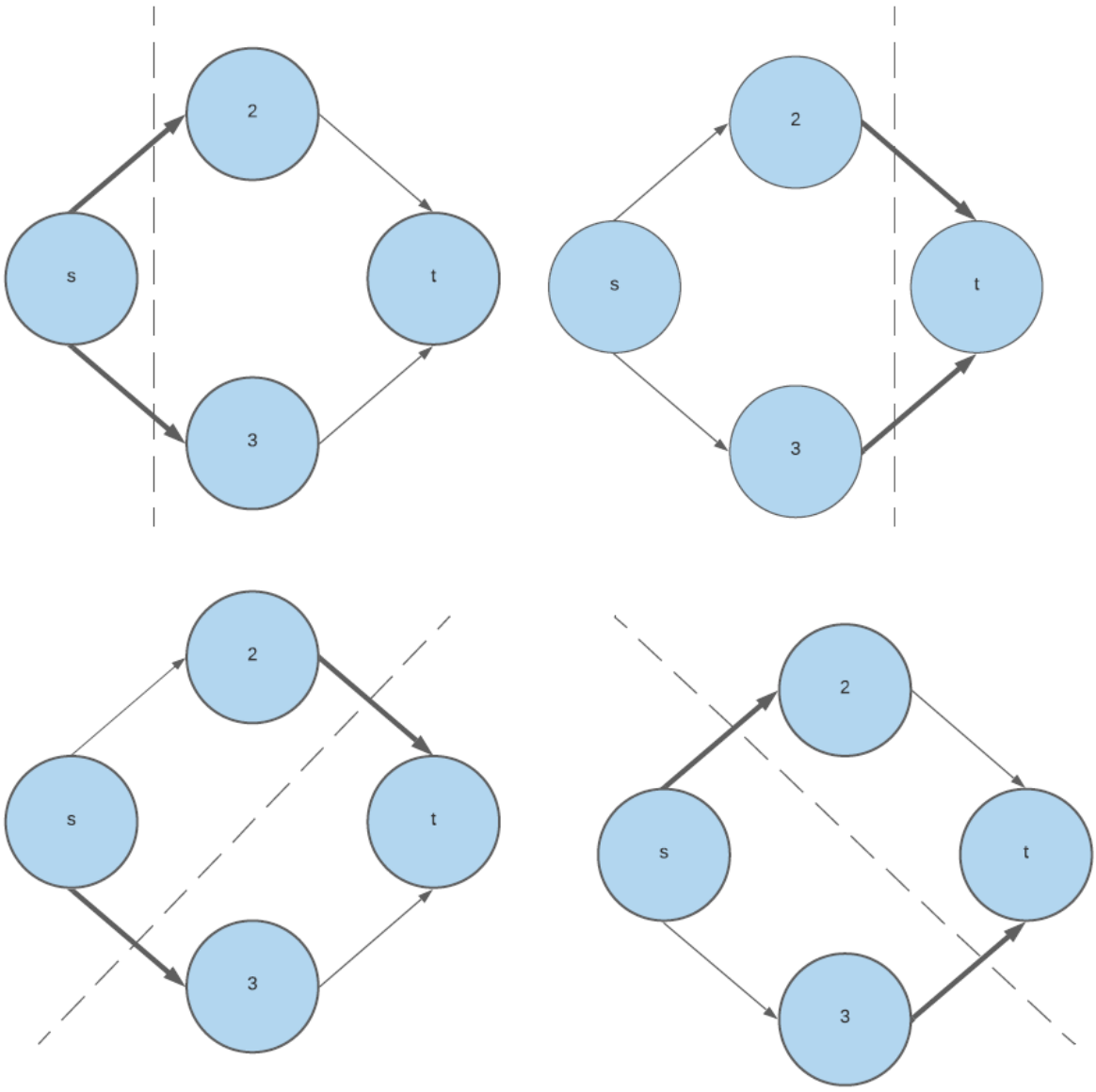
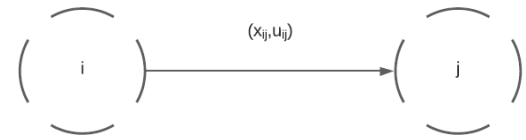


Рис.1.3.2. Всі перерізи s-t для даної мережі

1.4. Залишкова мережа

Концепція залишкової мережі відіграє ключову роль у розробці алгоритмів для знаходження максимального потоку. Для потоку x залишкова пропускна здатність будь-якої дуги $(i,j) \in A$ рівна $u_{ij} - x_{ij} + x_{ji}$. Залишкова пропускна здатність r_{ij} складається з двох компонентів: u_{ij}-x_{ij} – невикористана пропускна здатність дуги (i,j), та x_{ji} – розглядуваний потік через дугу (j,i), який ми можемо відкинути, щоб збільшити потік з вершини i у вершину j. Будемо називати мережу $G(x)$, яка складається з дуг з позитивними пропускними здатностями, як залишкову мережу у відповідності до потоку x. На рисунку 1.4.1 показано початкову мережу з пропускними здатностями дуг та потоком x, а на рисунку 1.4.2 – залишкову мережу $G(x)$.



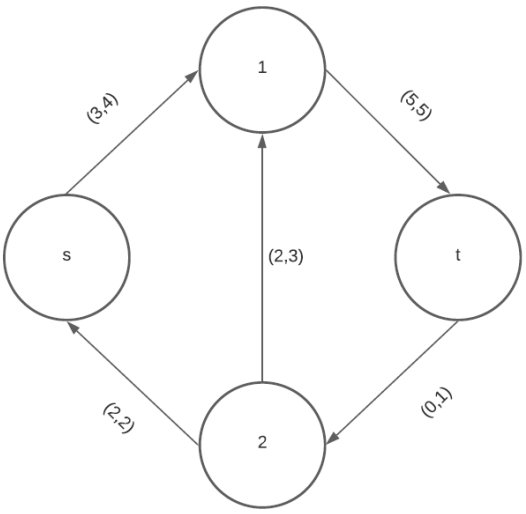
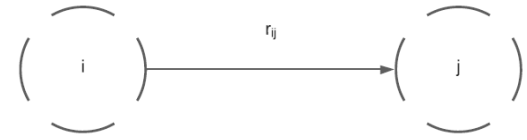


Рис. 1.4. 1



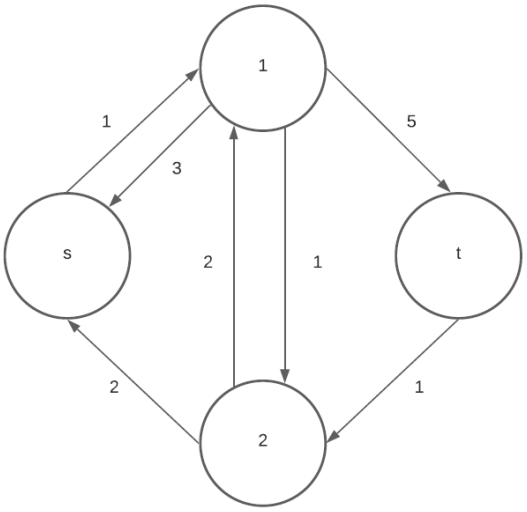


Рис. 1.4. 2

1.5. Мітки відстані

Функція відстані $d: N \rightarrow Z + \cup\{0\}$ з врахуванням залишкових пропускних здатностей r_{ij}, перетворює множину вершин у множину невід'ємних цілих чисел. Будемо називати *d(i)* міткою відстані вершини *i* і вони будуть *правильними* у відношенні до потоку *x*, якщо задовольнятимуть дві умови:

d(t)=0;

$d(i) \leq d(j) + 1$ для будь-якої дуги *(i,j).*

Дуга *(i,j)* називається *допустимою,* якщо вона задовольняє умову *d(i)=d(j)+1;* в інакшому випадку дуга буде *недопустимою*. Також шлях з вершини *i* у вершину *j*, який повністю складається з допустимих дуг,

називається допустимим шляхом. Допустимий шлях – це найкоротший шлях (за кількістю дуг) з вершини-джерела до вершини-стоку.

2. Дослідження алгоритмів розв'язування задачі про максимальний потік

2.1. Постановка задачі

Нехай $G = (N; A)$ – зв'язний орієнтований граф з невід'ємними значеннями пропускних здатностей u_{ij} для кожної дуги $(i, j) \in A$. Означимо дві вершини $s, t \in N$ як джерело та стік, враховуючи, що потік починається в s та прямує до t. Всі інші вершини називатимемо проміжними.

В даній мережі з джерелом s та стоком t, вектор потоку $x \in \mathbb{R}|U|$ називається *досяжним* в $G = (N; A)$, якщо кожна його компонента задовольняє обмеження пропускної здатності для будь-якого значення $v \geq 0$,

$$\sum_{j \in N\, A(i)} x_{ij} + \sum_{j \in N\, B(i)} x_{ji} = \begin{cases} v, \text{якщо } i = s \\ 0, \text{якщо } i \neq s, t \text{ та } i \in N \\ -v, \text{якщо } i = t \end{cases}$$

(рівняння неперервності потоку)

Величина потоку є сумою таких векторів:

$$v = \sum_{j \in N\, A(s)} x_{sj}$$

де $N\,A(i)$ та $N\,B(i)$ – це набір всіх вершин після i та перед i відповідно.

Ми припускаємо, що джерела *s* можуть видавати необмежену величину потоку, а стоки *t* можуть приймати необмежену величину потоку. Тепер маємо задачу знаходження максимального потоку з джерела *s* у стік *t* в *G = (N; A)*, який задовольняє рівняння неперервності потоку та відповідні обмеження:

$$\textbf{Max}\, v = \sum_{j \in N\, A(i)} x_{ij} + \sum_{j \in N\, B(i)} x_{ji} = \begin{cases} v, \text{якщо } i = s \\ 0, \text{якщо } i \neq s, t \text{ та } i \in N \\ -v, \text{якщо } i = t \end{cases}$$

При цьому, $0 \leq x_{ij} \leq u_{ij}$ для всіх $(i, j) \in U$.

Задача про пошук максимального потоку являє собою особливий вид задачі про потік мінімальної вартості і може бути поставлена як така простим додаванням однієї дуги з безмежною пропускною здатністю, що з'єднує вершину-джерело та вершину-стік (якщо такої дуги досі немає в мережі).

Для подальшого дослідження задачі про максимальний потік будемо використовувати наступні припущення:

Припущення 1. *Якщо мережа містить дугу (i, j), то мережа також містить дугу (j, i).*

Припущення 2. *Нижня межа значення потоку, що проходить крізь дугу, дорівнює 0.*

Припущення 3. *Існує лише одна вершина-стік та лише одна вершина-джерело.*

Припущення 4. *Алгоритми, складність яких включає в себе U, мають цілочисельні значення пропускних здатностей дуг.*

2.2. Критерії ефективності алгоритмів

В ідеальній ситуації ми би хотіли вимірювати ефективність алгоритму на основі його здатності розв’язувати практичну проблему. Але загалом результати роботи алгоритму можуть варіюватись в залежності від контексту задачі, тому нам потрібно опиратись на теоретичне обґрунтування.

Ефективність алгоритму вимірюють на основі результатів для найгіршого випадку, тобто для максимальної кількості операцій, яку потребує алгоритм для розв’язання проблеми для будь-якої мережі з заданим розміром. Для задач про потік на мережі розмір задається кількістю дуг, вузлів та цілими числами, що характеризують пропускну здатність ребер (дуг)

Задача про максимальний потік полягає в пошуку максимально можливого потік в упорядкованій мережі від вказаного «джерела» до вказаного «стоку» без перебільшення пропускної здатності ребра з найбільшим показником пропускної здатності. Зворотна задача про мінімальний переріз, шукає набір ребер з найменшою сумарною пропускною здатністю, які розділяють «джерело» та «стік».

Задачі максимального потоку та мінімального перерізу постають в багатьох прикладних задачах. Їх пов’язує наступна теорема про зв’язок максимального потоку та мінімального перерізу:

значення максимального потоку на мережі співпадає зі значенням пропускної здатності мінімального перерізу ***s-t.***

В кінцевому підсумку основним показником ефективності алгоритму стає його часозатратність. Алгоритм з поліноміальним часом, це алгоритм в якого найгірший час роботи обмежений зверху поліноміальною функцією вхідних параметрів. Для задачі про максимальний потік вхідними параметрами є *n, m* та log *U* (кількість бітів, яка вказує на пропускну здатність найбільшої дуги). Ми називаємо алгоритм максимального потоку алгоритмом з псевдополіноміальним часом, якщо його найгірший час роботи обмежений поліноміальною функцією *n, m,* а *U*O(nmU)* означає, що час роботи обмежений вище деякою константою кратною *nmU* та *Ω(nmU)* означає, що час роботи обмежений внизу деякою постійною кратною *nmU.*

Протягом багатьох років дослідники намагалися знайти алгоритм, який буде швидшим, ніж бар'єр декомпозиції *Ω(nmU)*. Це природна нижня межа для алгоритмів, що вимагають декомпозиції потоку, та для алгоритмів, що збільшують потік на одній дузі шляху за раз. Черіян та ін. [3] запропонували подолання цього бар’єру для щільних графів, тоді як Голдберг та Рао [4] ще більше покращили цю нижню межу.

2.3. Алгоритми доповнюючого шляху

2.3.1. Базовий алгоритм доповнюючого шляху.

Перше покоління алгоритмів, що розв’язують задачу про максимальний потік – це алгоритми доповнюючого шляху. Нехай *x* - потік у графі *G*. Ми

називаємо спрямований шлях від джерела до стоку в залишковій мережі $G(x)$ як збільшуючий шлях.

Ми визначаємо залишкову пропускну здатність $\delta(P)$ збільшуючого шляху P як максимальну кількість потоку, який може бути направлений по ньому: $\delta(P) = min\{ r_{ij}: (i, j) \in P\}$. Оскільки залишкова пропускна здатність кожної дуги в залишковій мережі є строго додатною, залишкова пропускна здатність доповнюючого шляху також є строго додатною. Надсилання δ одиниць потоку по збільшуючому шляху зменшує залишкову пропускну здатність кожної дуги (i, j) на шляху на δ одиниць і збільшує залишкову пропускну здатність зворотної дуги (j, i) на δ одиниць.

Загальний алгоритм збільшення шляху [5,6] ідентифікує шляхи збільшення в $G(x)$ і посилає потік за цими шляхами, поки мережа містить такий шлях. Ми наводимо алгоритм нижче.

Алгоритм. Базовий алгоритм доповнюючого шляху.

begin

$x := 0$;

while $G(x)$ містить доповнюючий шлях **do**

begin

знайти доповнюючий шлях P;

збільшити x на $min\{ r_{ij}: (i, j) \in P\}$ одиниць та оновити $G(x)$;

end

end

Коли алгоритм завершує роботу, залишкові пропускні здатності r_{ij} можуть бути використані для побудови максимального потоку. Оскільки $r_{ij} = u_{ij} - x_{ij} + x_{ji}$, потоки через дуги задовільняють рівняння $x_{ij} - x_{ji} = u_{ij} - r_{ij}$. Якщо $u_{ij} > r_{ij}$, можемо покласти $x_{ij} = u_{ij} - r_{ij}$ та $x_{ji} = 0$; інакше покладемо $x_{ij} = 0$ та $x_{ji} = u_{ij} - r_{ij}$.

Продуктивність алгоритму залежить від (і) кількості ітерацій збільшення, та (іі) часу ідентифікації збільшувального шляху. Обидва числа залежать від того, який збільшувальний шлях ми обираємо на кожній ітерації. Ми можемо ідентифікувати доповнюючий шлях P у $G(x)$, використовуючи алгоритм пошуку у графі. Алгоритм пошуку у графі починається з вузлів s і поступово знаходить усі вузли, до яких можна дійти за допомогою впорядкованих шляхів. Більшість алгоритмів пошуку виконуються за час $O(m)$, при цьому або знаходять доповнюючий шлях, або роблять висновок, що $G(x)$ не містить доповнюючих шляхів; останнє трапляється, коли стік недоступний з джерела.

У кожній ітерації алгоритм збільшує потік від вузла-джерела до вузла-стоку на ціле додатне значення. Щоб визначити обмеження на кількість ітерацій, ми обчислюємо границю за допомогою значення максимального потоку. За визначенням U позначає найбільшу пропускну здатність дуги, отже, пропускна здатність перерізу $[\{s\}, N-\{s\}]$ не може становити більше ніж $n \cdot U$. Оскільки значення будь-якого потоку ніколи не може перевищувати пропускну здатність будь-якого перерізу в мережі, і алгоритм посилає принаймні одну одиницю потоку на кожній ітерації, ми отримуємо межу $n \cdot U$ на кількість

ітерацій. Отже, час роботи алгоритму дорівнює $O(n \cdot m \cdot U)$, що є псевдополіноміальним часом.

2.3.2. Алгоритм найкоротшого доповнюючого шляху

Алгоритм найкоротшого доповнюючого шляху завжди збільшує потік на шляху s-t , який містить найменшу кількість дуг. Оскільки шлях, що містить мінімальну кількість дуг. Оскільки мінімальна відстань з будь-якого вузла i до стоку t монотонно неспадна, за всіх збільшень, ми зменшуємо час для кожного збільшення до $O(n)$.

Алгоритм найкоротшого доповнюючого шляху працює завдяки збільшенню потоків уздовж допустимих шляхів. Він створює допустимий шлях, поступово збільшуючи частково допустимий шлях, тобто шлях від s до вузла i, що складається виключно з допустимих дуг, і ітеративно виконує операції проходження або повернення від останньої вершини частково допустимого шляху , який ми називаємо поточною вершиною. Якщо поточна вершина i має (тобто є інцидентною) допустиму дугу (i, j), тоді ми виконуємо операцію проходження і додаємо дугу (i, j) до частково допустимого шляху; в іншому випадку ми виконуємо операцію повернення і повертаємо одну дугу назад. Ми повторюємо ці операції до тих пір, поки частково допустимий шлях не досягне стоку, і тоді ми збільшуємо потік на $min\{ r_{ij}: (i, j) \in P\}$ одиниць. Ми повторюємо цей процес, поки потік не буде максимальним. Алгоритм

продемонстрований нижче. В алгоритмі ми вводимо *pred(i)* визначається як вершина що була попередньою до вершини і на поточному шляху.

Алгоритм. Алгоритм найкоротшого доповнюючого шляху

begin

x:=0,
поточна вершина := *s*;

while *d(s)<n* **do**

begin

if поточна вершина має вихідну допустиму дугу

then проходження(поточна вершина)

else повернення(поточна вершина);

if поточна вершина=*t* **then** збільшення;

end

end

procedure *проходження(i)*

begin

нехай *(i, j)* допустима дуга;

pred(*j*):= *i* та *i* := *j;*

end

procedure повернення(*i*)

begin

d(i):=min{d(j)+1: (i,j) ∈ A та *rij>0};*

if *i≠s* **then** *i*:=pred(*i);*

end

procedure збільшення
begin
використовуючи індекси попередніх вершин знаходимо доповнюючий шлях P;
збільшуємо x на $min\{ r_{ij}: (i, j) \in P\}$
end

Цей алгоритм легко реалізувати на практиці. Отримані алгоритми можуть знайти не більше $O(n \cdot m)$ доповнюючих шляхів, і ця межа є сталою; тобто, зокрема, в цих прикладах ці алгоритми виконують збільшення $O(n \cdot m)$ раз. Єдиний спосіб покращити час роботи алгоритму найкоротшого доповнюючого шляху - це виконувати менше обчислень за одне доповнення. Можна показати, що використання більш складної структури даних, так званих динамічних дерев, зменшує середній час кожного збільшення з $O(n)$ до $O(\log n)$. Ця реалізація алгоритму найкоротшого доповнюючого шляху працює за час $O(n \cdot m \cdot \log n)$.

2.3.3. Алгоритм доповнюючого шляху з масштабуванням пропускної спроможності

Ми почнемо з пояснення алгоритму доповнюючого шляху максимальної пропускної здатності. Алгоритм доповнюючого шляху максимальної пропускної здатності, який був спочатку розроблений Едмондсом та Карпом [11], завжди збільшує потік лише вздовж шляху з максимальною залишковою пропускною здатністю. Нехай x - будь-який потік, v - його значення потоку, v^* -

максимальне значення потоку. Теорія декомпозиції потоку показує, що в залишковій мережі $G(x)$ ми можемо знайти m або менше напрямлених шляхів від джерела до стоку, залишкова пропускна здатність якої дорівнює $(v^* - v)$. Отже, доповнюючий шлях з максимальною пропускною здатністю має залишкову пропускну здатність щонайменше $(v^* - v) / m$. Тепер розглянемо послідовність з $2m$ поспіль збільшень до максимальної пропускної здатності, починаючи з потоку x. Припустимо, що v' - результуюче значення потоку. Якщо при кожному збільшенні ми додамо щонайменше $(v^* - v) / 2m$ одиниць потоку, тоді можна буде встановити максимальний потік протягом $2m$ або менше ітерацій. Однак, якщо одне з цих $2m$ збільшень додає менше ніж $(v^* - v) / 2m$ одиниць потоку, то $(v^* - v)$ не більше $(v^* - v) / 2$. Оскільки значення залишкової пропускної здатності будь-якого збільшуючого шляху лежить між 1 і $2U$, потік повинен досягнути максимального за $O(m \log U)$ ітерацій.

Алгоритм доповнюючого шляху максимальної пропускної здатності зменшує кількість ітерацій у базовому алгоритмі збільшення шляху від $O(n \cdot U)$ до $O(m \cdot \log U)$. Однак алгоритм виконує більше обчислень за ітерацію, оскільки йому потрібно знайти не просто будь-який збільшуючий шлях, але той, що має максимальну залишкову пропускну здатність.

Зараз ми представимо варіацію алгоритму збільшення маршруту збільшення максимальної пропускної здатності, який не виконує більше обчислень за ітерацію, ніж базовий алгоритм збільшення маршруту, і все ж

встановлює максимальний потік в межах ітерацій $O(m \cdot \log U)$. Оскільки цей алгоритм неявно масштабує потужність дуги, ми називаємо його алгоритмом масштабування пропускної здатності. Алгоритм збільшує потік через шлях з достатньо великою залишковою пропускною здатністю, замість шляху з максимальною збільшуючою здатністю. Ми можемо отримати шлях із достатньо великою залишковою пропускною здатністю досить легко за час $O(m)$. Щоб визначити алгоритм масштабування пропускної здатності, введемо параметр Δ та визначимо *Δ-залишкову мережу*, як мережу, що містить дуги, залишкова пропускна здатність яких принаймні Δ. Нехай $G(x, \Delta)$ це Δ-залишкова мережа. $G(x, 1) = G(x)$ та $G(x, \Delta)$ це підграф $G(x)$. Алгоритм наведено нижче.

Алгоритм. Алгоритм масштабування пропускної спроможності

begin

$x:=0$; $\Delta:=2^{[\log U]}$;

while $\Delta \geq 1$ **do**

begin

if $G(x, \Delta)$ містить шлях з вершини s до вершини t **then**

begin

знайти шлях P в $G(x, \Delta)$;

збільшити x на $\{ r_{ij}: (i, j) \in P\}$ одиниць на P;

оновити $G(x, \Delta)$;

end

$\Delta := \Delta/2$;

end

end

Алгоритм масштабування пропускної спроможності розв'язує задачу пошуку максимального потоку за $O(m \cdot \log U)$ ітерацій і має тривалість $O(m^2 \cdot \log U)$. Ба більше Ахуджа та Орлін [9] покращили час роботи алгоритму до $(n \cdot m \cdot \log U)$ використовуючи найкоротший доповнюючий шлях, як підпрограму для кроку, в якому доповнюючий шлях визначений.

2.4. Алгоритми блокуючого потоку

2.4.1. Базовий алгоритм блокуючого потоку

Замість того, щоб шукати доповнюючі шляхи один за одним, Дініц [10] запропонував шукати набір доповнюючих шляхів на кожній ітерації.

Нехай $d(i)$ - довжина шляху з найменшою кількістю дуг від вузла i до вузла t. Багатошаровий граф - це підграф $G(x)$, який містить лише допустимі дуги, тобто (i, j) з $d(i)=d(j)+1$ (рис. 2.4.1.1).

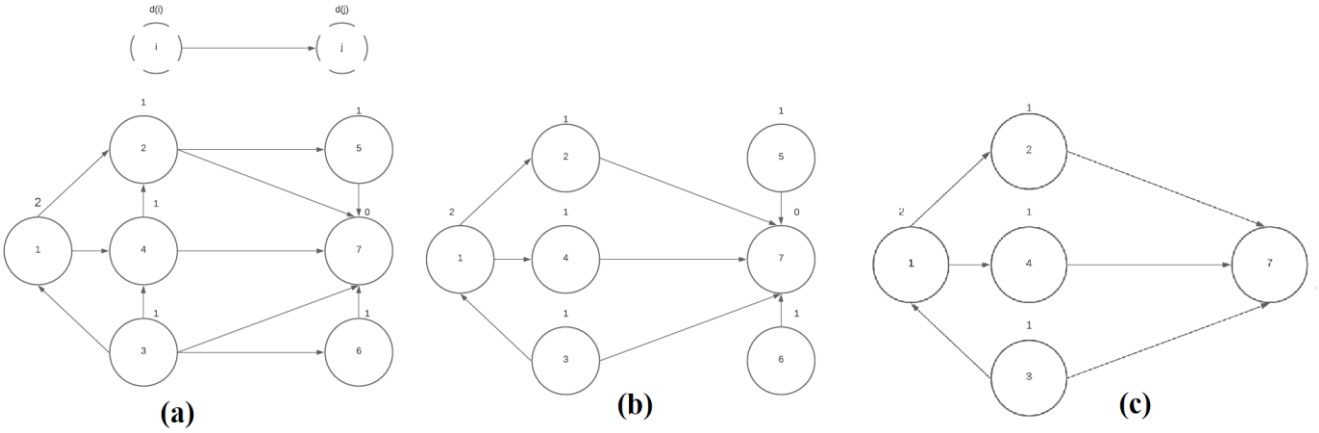


Рис. 2.4.1.1. (a) залишкова мережа, (b) відповідна шарувата мережа, (c) шарувата мережа після видалення залишкових дуг.

Блокуючий потік - це максимальний потік у багатошаровому графі, і він може включати не більше ніж *m* доповнюючих шляхів. Потік буде максимальним після *n-1* операцій блокування потоку. Блокуючий потік можна знайти за час $O(n \cdot m)$ шляхом пошуку в глибину або за час $O(m \cdot \log n)$ використовуючи структуру динамічного дерева. Загальний час роботи цих алгоритмів рівний $O(n^2 \cdot m)$ та $O(n \cdot m \cdot \log n)$ відповідно. Загальний вигляд алгоритму наведено нижче.

Алгоритм. Алгоритм базового блокуючого потоку

begin

x:=0;

while стік досяжний з джерела **do**

begin

знайти поточну ** мережу;

знайти відповідний блокуючий потік *f*;

збільшити *x* на *f*;

оновити залишкову мережу;

end

end

2.4.2. Алгоритм двійкового блокуючого потоку

В класичних варіантах алгоритмів блокуючого потоку кожна дуга має одиничну довжину. Голдберг та Рао [6] продемонстрували алгоритм блокуючого потоку. в якому довжини дуг залежать від залишкової пропускної здатності. Введемо означення довжини дуги (i, j) як

$$l_{ij} = \begin{cases} 0, \text{якщо } 2\Delta \leq r_{ij} < 3\Delta, \;\; d(i) = d(j), r_{ji} \geq 3\Delta \\ l_{ij}, \qquad \text{в інакшому випадку} \end{cases}$$

де

$$l_{ij} = \begin{cases} 0, \quad r_{ji} \geq 3\Delta \\ 1, \text{в інакшому випадку} \end{cases}$$

Алгоритм складається з $O(n \log U)$ Д-залишкових фаз. [Sk,Tk]-канонічний переріз, якщо $Sk=\{v \in N;\ d(v) \geq k\}$, $Tk=N-Sk$. Позначимо $[\overline{S,T}]$ як канонічний переріз з найменшою пропускною здатністю, а $u[\overline{S,T}]$ як пропускну здатність цього перерізу. d_l та $d_{l'}$ позначають змінні відстаней в термінах функій відстаней l та l' відповідно. F це верхня межа значення потоку. Алгоритм наведено нижче.

Алгоритм. Алгоритм двійкового блокуючого потоку.

begin

поточний переріз := [{*s*},{*N*-*s*}], *F*:=*nU*, Δ:=min{[$F/m^{1/2}$],[$F/n^{2/3}$]}

while Δ≥1 **do**

begin

while $[\overline{S},\overline{T}]$ > F/2 **do**

begin

обчислюємо *l’*;

стягуємо сильно зв’язані компоненти індуковані дугами нульової довжини;

знаходимо значення потоку Δ або блокуючого потоку;

оновлюємо залишкову мережу;

обчислюємо *l* та d_l ;

end

F:= u$[\overline{S},\overline{T}]$, поточний переріз:= *u*$[\overline{S},\overline{T}]$, *Δ*:=min{[$F/m^{1/2}$],[$F/n^{2/3}$]}

end

end

Стягування сильно пов’язаних компонентів – це дуже важливий крок алгоритму, оскільки найкоротший шлях на залишковому графі може мати допустимі цикли через дуги нульової довжини. Алгоритм стягує вершини, що сильно зв’язані дугами нульової довжини. Після цього потік циклує всередині стягнутих вершин. Алгоритм двійкового блокуючого потоку завершує роботу за час $O(min\{n^{2/3},\sqrt{m}\}m\log(n^2/m)\log U)$.

2.5. Алгоритми проштовхування предпотоку

2.5.1. Загальний алгоритм проштовхування предпотоку

Інший клас алгоритмів для вирішення задачі про максимальний потік називається алгоритмами проштовхування предпотоку, які працюють з окремими дугами, а не з цілими шляхами, як це робить алгоритм доповнюючого шляху. Посилання потоку потужності д вздовж шляху який складається з *k* дуг розкладається на *k* базових операцій посилання потоку потужності д вздовж кожної дуги на шляху. Назвемо кожну з таких операцій поштовхом.

Алгоритм доповнюючого шляху підтримує збереження потоку у всіх вершинах, тоді як алгоритм проштовхування предпотоку не дозволяє збереження потоку на всіх кроках окрім найостанніших, натомість підтримує предпотік на кожній ітерації. Предпотік – це вектор *x*, який задовольняє обмеження пропускної здатності та наступне спрощене рівняння неперервності потоку:

$\sum_{(j,i)\in A} x_{ji} - \sum_{(i,j)\in A} x_{ij} \geq 0$, для всіх i ∈ N – {s,t}

Кожен елемент вектору предпотоку – це або невід'ємне дійсне число або +∞. Для даного предпотоку *x*, означуємо надлишковий потік для кожної вершини $i \in N$ -{*s,t*} як

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

Вершину з додатнім надлишковим потоком будемо називати активною вершиною. Також вказуємо те, що вершина-джерело та вершина-стік ніколи не можуть бути активними. В алгоритмі проштовхування предпотоку наявність активної вершини вказує на те, що розв'язання проблеми поки недоступне. Звідси слідує, що базова операція алгоритму вибирає активну вершину *i* та намагається позбутись надлишкового потоку шляхом «виштовхування» потоку з неї.

Якщо ми будемо просто «виштовхувати» потік до сусідньої вершини у випадковому порядку та будемо робити це з іншими вершинами тоді, можливо, що деякі вершини будуть виштовхувати потік поміж один одним та утворювати нескінченний цикл операцій. Оскільки ми хочемо скерувати потік у вершину-стік, то логічним кроком буде направити надлишок у вершину, що знаходиться «ближче» до вершини-стоку.

Якщо це правило буде виконуватись для всіх вершин, то алгоритм ніколи не зіткнеться з проблемою нескінченного циклу. Алгоритм проштовхування предпотоку використовує так звані *мітки відстані* для кожної вершини.

Базова операція в алгоритмі проштовхування предпотоку вибирає активну вершину *i* та намагається позбутись надлишку, проштовхуючи потік до вершини, в якої мітка є меншою. Якщо вершина *i* має допустиму дугу (*i*,*j*),тоді *d(j)=d(i)*-1 і алгоритм направляє потік через допустимі дуги, щоб позбутись надлишкового потоку у вершині.

Якщо вершина *i* не має допустимих дуг, то алгоритм збільшує мітку відстані вершини, щоб у вершини з'явилась допустима дуга. Алгоритм завершує свою роботу тоді, коли у графі не залишається активних вершин, тобто надлишковий потік присутній лише у вершині-джерелі та вершині-стоці.

Проштовхування ∂ одиниць потоку з вершини *i* у вершину *j* знижує надлишок *e(i)* у вершині *i* та залишок r_{ij} дуги *(i,j)* на ∂ одиниць та збільшує *e(j)* та r_{ji} на ∂ одиниць. Якщо $\partial = r_{ij}$,, то проштовхування на дузі *(i,j)* буде *насиченим*. В інакшому випадку $\partial = e(i)$,тоді проштовхування буде ненасиченим. Алгоритм наведено нижче.

Алгоритм. Загальний алгоритм проштовхування предпотоку

begin

$x := 0$, $d(i) := 0$ для всіх $i \in N$;

$x_{sj} := u_{sj}$, $e(j) = u_{sj}$ у всіх дуг $(s,j) \in A$, $e(s) := 0$, $d(s) := n$;

while *G(x)* містить активну вершину **do**

begin

обираємо будь-яку активну вершину *k;*

if мережа містить допустиму дугу *(k,j)* **then**

проштовхуємо $\{e(k), r_{kj}\}$ одиниць з *k* до *j;*

else

міняємо $d(k)$ *на* $min\{d(j)+1: (k, j)\in A$ та $r_{kj} > 0\}$;

end

end

Складність алгоритму залежить від того, як ми обираємо активну вершину. Можливо показати, що алгоритм проштовхування предпотоку слідкує за мітками відстаней на кожному кроці алгоритму, та збільшує мітки відстані кожної вершини щонайбільше $2n$ раз.

Якщо ми оберемо будь-яку активну вершину, алгоритм здійснить $O(nm)$ насичених проштовхувань та $O(n^2m)$ ненасичених. Отже час роботи алгоритму складає $O(n^2m)$.

2.5.2. FIFO-алгоритм проштовхування предпотоків

FIFO алгоритм проштовхування предпотоків досліджує активні вершини у порядку «першим прийшов – першим пішов» (англ. first-in-first-out). Алгоритм працює з набором активних вершин, як з чергою. Він обирає вершину i на початку черги, виконує проштовхування для цієї вершини і тоді додає нові активні вершини в кінець черги. Алгоритм завершує свою роботу коли черга стає пустою.

Перша фаза перевіряє вершини, що стали активними під час попередньої обробки. Друга фаза складається з перевірки вершин, які потрапили до черги

під час проходження першого етапу. Подібним чином, третій етап складається з перевірки вершин, що потрапили до черги під час другого етапу і так далі.

Можливо показати, що алгоритм проводить $2n^2+n$ фаз. Під час кожної фази будь-яка вершина перевіряється щонайбільше один раз, і кожна перевірка вершини виконує щонайбільше один ненасичений поштовх. Отже, ліміт на кількість фаз $2n^2+n$ встановлює ліміт $O(n^3)$ на кількість ненасичених проштовхувань. Звідси час роботи FIFO-алгоритму проштовхування предпотоків буде $O(n^3)$, тому що «вузьким місцем» цього алгоритму якраз і являється кількість ненасичених проштовхувань.

2.5.3. Алгоритм проштовхування предпотоків з найвищої мітки

Алгоритм проштовхування предпотоків з найбільшої мітки, завжди проштовхує потік з активної вершини, значення мітки відстані якої найбільше. Нехай $h^*=max\{d(i)$: i активна вершина$\}$. Алгоритм завжди в першу чергу перевіряє вершини зі значенням мітки h^* та проштовхує потік у вершину зі значенням мітки h^*-1. Потім потік з цих вершин проштовхується до вершин з мітками зі значенням h^*-2, і так продовжує до тих пір поки алгоритм починає перепомічати вершини, або закінчуються всі активні вершини. Коли алгоритм перепомічає вершину, він запускає такий самий процес. Якщо алгоритм не перепомічає будь-яку вершину за n послідовних перевірок вершин, то весь надлишок надходить до вершини-стоку(або вершини-джерела) і алгоритм

завершує свою роботу. Так як алгоритм виконує щонайбільше $2n^2$ операцій перепомітки, ми отримуємо межу $O(n^3)$ на кількість перевірок вершин. Оскільки кожна перевірка вершин проводить щонайбільше один ненасичений поштовх, то алгоритм виконує $O(n^3)$ ненасичених поштовхів.

Для вибору вершини з найбільшою міткою відстані будемо використовувати наступну структуру даних, яка дозволяє уникнути використання великої кількості ресурсів. Для кожного

$$k=1, 2, \ldots, 2n-1,$$

створюємо список

$$\text{LIST}(k)=\{i : i \text{ є активною та } d(i)=k\}$$

у формі з'єднаних списків, або з'єднаних стеків. Вводимо поняття змінної *рівень*, яка характеризує верхню межу найбільшого значення k, для якого LIST(k) непорожній. Для того щоб визначити вершину з найбільшою міткою відстані, перевіряємо списки LIST(*рівень*), LIST(*рівень*-1), поки не знаходимо непорожній список, нехай це буде LIST(p). Встановлюємо значення *рівень*=p та вибираємо будь-яку вершину з LIST(p). Також якщо при перевірці вершини мітка відстані збільшується, тоді ми встановлюємо значення рівня, яке буде дорівнювати новому найбільшому значенню мітки відстані вершини. Варто зазначити, що максимальне загальне збільшення рівня дорівнює $2n^2$, а зменшення $2n^2+n$. Внаслідок цього перевірка списків LIST(*рівень*), LIST(*рівень*-

1), … для знаходження першого непорожнього списку не є більше найвужчим місцем алгоритму.

Черіян та Манешварі [16] показали, що кількість ненасичених поштовхів дорівнює $O(n^2\sqrt{m})$. Алгоритм проштовхування предпотоків з найбільшої мітки, на даний момент, являється найбільш ефективним методом для розв'язання задачі про максимальний потік на практиці, тому що він виконує найменшу кількість ненасичених поштовхів.

2.5.4. Алгоритм масштабування надлишкового потоку

Алгоритм масштабування надлишкового потоку – це одна з модифікацій алгоритму проштовхування предпотоку. Алгоритм наведено нижче.

Алгоритм. Алгоритм масштабування надлишкового потоку.

begin

$x:=0$, $d(s):=n$, $d(t):=0$, $d(i):=1$ для $i \in N - \{s,t\}$;

покладемо x_{sj} та $e(j)$ як u_{sj} для всіх дуг $(s,j) \in A$, та $\Delta:=2^{[\log U]}$;

while $\Delta \geq 1$ **do**

begin

if $G(x, \Delta)$ містить активні вершини з надлишком більшим ніж $\Delta /2$ **then**

begin вибираємо вершину *i* з надлишком більшим ніж *Δ/2* та найменшою міткою відстані;

if мережа містить допустиму дугу *(k,j)* **then**

проштовхуємо $min\{e(k), r_{kj}, \Delta\text{-}e(j)\}$ одиниць потоку з *k* до *j*;

else оновлюємо $d(k)=min\{d(j)+1: (k,j) \in A$ та $r_{kj}>0\}$

end

$\Delta := \Delta /2;$

end

end

Алгоритм масштабування надлишкового потоку працює за час $O(nm+n^2\log U)$. Ахуджа та інші [19] отримали покращену версію алгоритму, збільшивши фактор масштабування з 2 до *k* (*Δ* стає *Δ* / k в наступній фазі), та обираючи надлишкову вершину з найбільшою міткою відстані. Ця версія працює за час $O(nm\log(n/m\sqrt{\log U})$

2.6. Інші алгоритми

В цьому розділі ми швидко проглянемо інші підходи до розв’язання задачі про максимальний потік

2.6.1. Алгоритм Малхотри, Прамод-Кумара та Махешварі (МПМ)

Цей алгоритм базується на ідеї яка обирає вершину з найнижчим потенціалом потоку. $p_x(i)$ – це максимальна величина потоку яка може пройти крізь вершину.

$$p_x(i) = min\left\{\sum_{(j,i)\in A}(u_{ji} - x_{ji}), \sum_{(i,k)\in A}(u_{ik} - x_{ik})\right\}$$

Алгоритм. МПМ алгоритм

begin

x:=0;

while є вершини в цій мережі **do**

begin

обираємо вершину k з найменшим потенціалом потоку;

надсилаємо $p_x(k)$ одиниць потоку з вершини *k* до вершин *s* та *t*;

збільшуємо *x* на $p_x(k)$, оновлюємо *G(x);*

видаляємо всі насичені дуги, вершину *k*, та всі вершини до яких надходять тільки вхідні або вихідні дуги;

end

end

2.6.2. Алгоритм впорядкування за максимальною суміжністю

Впорядкування вершин $v_1, v_2, \ldots, v_n$ називається впорядкуванням за максимальною суміжністю (МС), якщо для всіх i, v_i – це вершина в $N - \{v_1, v_2, \ldots, v_{i-1}\}$, яка має найбільшу суму залишкових пропускних здатностей дуг до вершин $v_1, v_2, \ldots, v_{i-1}$.

Алгоритм впорядкування за максимальною суміжністю був запропонований Фуджишіге [20] та працює за час $O(n(m+n\log n)\log nU$, а його версія з масштабуванням працює за час $O(nm\log U)$. Мацуока та Фуджишіге запропонували покращену версію алгоритму Фуджишіге, використовуючи предпотоки. Попри те, що в теорії час роботи обох алгоритмів однаковий, останній алгоритм швидший на практиці.

2.6.3. Алгоритм збалансування дуг

Тар'ян та інші представили круговий алгоритм балансування дуг, який знаходить максимальний потік за час $O(n^2 m\log(Nu))$. Попри те, що цей алгоритм повільніший ніж інші відомі алгоритми, він надзвичайно простий. Цей алгоритм використовує псевдопотік, а не предпотік. Нехай x це псевдопотік тоді, $x_{ij}=-x_{ji}$ та $x_{ij} \leq u_{ij}$ для всіх дуг $(i,j) \in A$. Більше того, баланс у вершині i можна розрахувати за формулою

$$b(i) = \sum_{(j,i)\in A} x_{ji}$$

де *b(i)* може приймати як додатні (надлишок) так і від'ємні (нестача) значення.

Алгоритм. Алгоритм збалансування дуг

begin конструюємо початковий псевдопотік насичуючи кожну дугу, яка виходить з вершини s, та кожну дугу, що входить у вершину та присвоюєм нульовий потік до кожної іншої дуги.

while (сума всіх додатніх балансів>1) та (сума всіх негативних балансів < -1) **do**

begin

вибираємо дугу (v,w) з $b(v)>b(w)$, $w\neq s$ та $v\neq t$;

збільшуємо потік на (v,w) на $\min\{r_{vw}, e(v)/2 - e(w)/2\}$;

end

end

2.6.4. Алгоритм псевдопотоку Гохбаума

Гохбаум [24] запропонував алгоритм псевдопотоку. який працює за час $O(nm\log n)$. Чандаран та Гохбаум [25] показали, що алгоритм псевдопотоку через найбільшу мітку працює швидше, ніж аналог проштовхування предпотоку в більшості випадків для різних варіацій задачі. Гохбаум та Орлін [26] довели,

що ця версія алгоритму працює за час $O(mn\log n^2/m)$ з використанням динамічних дерев та час $O(n^3)$ без їх використання.

2.7. Максимальний потік на спеціальних графах

В дуже багатьох випадках, алгоритми розв'язання проблеми про максимальний потік в спеціальних графах виявляються більш ефективними ніж ті самі алгоритми на звичайних графах. В цьому розділі розгдянемо дві спеціальні мережі.

2.7.1. Потік в мережах з одиничними пропускними здатностями

Якщо в умовах задачі про максимальний потік всі ребра розглядуваних графів мають пропускну здатність рівну одиниці, то такі мережі називаються мережами з одиничними пропускними здатностями.

В мережі з одиничними пропускними здатностями значення максимального потоку має свою верхню межу яка дорівнює n, оскільки пропускна здатність перерізу s-t дорівнює n. Базовий алгоритм доповнюючого шляху знаходить максимальний потік за n збільшень та потребує часу $O(nm)$.

Алгоритм найкоротшого доповнюючого шляху також розв'язує цю задачу за час $O(nm)$, оскільки його вузьке місце (крок збільшення) потребує $O(nm)$ часу замість $O(n^2m)$. Алгоритм знаходження максимального потоку в одиничній мережі, який ми описуємо - це гібрид двох вище згаданих алгоритмів та потребує лише $O(min\{n^{2/3}m, m^{3/2}\})$ часу, що є значно кращим часом ніж $O(nm)$. Цей алгоритм працює за час $O(n^{1/2}m)$ у простих мережах. Мережа з одиничними

пропускними здатностями називається простою, якщо з вершин виходить лише по одній вхідній та вихідній дузі.

Алгоритм. Алгоритм для мереж з одиничними пропускними здатностями.

begin

застосовуємо алгоритм найкоротшого доповнюючого шляху поки $d(s) \geq min\{2n^{2/3}m^{1/2}\};$

потім застосовуємо загальний алгоритм доповнюючого шляху поки не знайдемо максимальний потік;

end

2.7.2. Потік в дводольних графах

Дводольний граф, це граф множину вершин якого можна розбити на дві частини (N_1, N_2) таким чином, що кожне ребро графу з'єднує вершину з однієї частини множини з вершиною з іншої частини, тобто ребер, що з'єднують вершини з однієї частини множини не існує. Для позначення дводольних графів використвують позначення $G= (N_1 \cup N_2, A)$. Нехай $n_1=|N_1|$ *та* $n_2=|N_2|$.

В цьому параграфі, ми описуємо адаптацію алгоритму проштовхування предпотоку. Найгірший сценарій розвитку цих спеціальних алгоритмів подібний до найгірших випадків базового алгоритму, якщо множини вершин N_1, N_2 подібні за розміром. Припускаємо, що $n_1 \leq n_2$ та вершина-джерело

знаходиться в N_2. Як приклад використаємо модифікацію загального алгоритму проштовхування предпотоку. Час роботи цього алгоритму $O(n_1^2 m)$, та якщо $n_1 < n_2$, то нова імплементація алгоритму працює значно швидше за загальний алгоритм. Також для цієї модифікації ми розглядаємо ідеї, які використовують алгоритми FIFO, найбільшої мітки, проштовхування предпотоку, масштабування надлишкового потоку та інших, які покращують час роботи в найгірших випадках. Ми включаємо операції проштовхувань та перепомічування, а інші частини алгоритму лишаються незмінними.

Алгоритм. Алгоритм для дводольних графів.

procedure дводольне проштовхування/перепомічування;

begin

if залишкова мережа містить допустиму дугу *(i, j)* **then**

if залишкова мережа містить допустиму дугу *(j, k)* **then**

проштовхуємо δ = $min\{e(i), r_{ij}, r_{jk}\}$ одиниць потоку через шлях $i \to j \to k$ та збільшуємо x на δ;

else замінюємо $d(j)$ на $min\{d(k)+1: (j,k) \in A$ та $r_{jk}>0\}$;

else замінюємо $d(i)$ на $min\{d(j)+1: (i,j) \in A$ та $r_{ij}>0\}$;

end

2.8. Підсумок аналізу алгоритмів

Більшість алгоритмів з найкращим часом для знаходження максимального потоку базуються на алгоритмах проштовхування предпотоку: алгоритм Голдеберга та Тар'яна працює за час $O(nm \log(n^2/m))$; алгоритм Ахуджі за час $O(nm \log(n/m\sqrt{\log U+2}))$; алгоритм Черіяна з правилом випадкового вибору дуги працює за час $O(n^3/\log n)$ або з високою ймовірністю $O(nm+(n\log n)^2)$. Кінг продемонстрував детермінований варіант алгоритму Черіяна, який працює за час $O(nm+n^{2+\varepsilon})$ для будь-якого $\varepsilon>0$.

Сьогодні найкращі алгоритми проштовхування предпотоку перевершують найкращі алгоритми доповнюючого шляху як на практиці, так і в теорії. Чандран та Гохбаум показали, що алгоритми псевдопотоку працюють швидше ніж алгоритми проштовхування предпотоку для більшості задач.

Представлені у роботі алгоритми підсумовані таблицею в Додатку А. Для кожного алгоритму ми виділили основні показники роботи та найкращий час роботи. Алгоритми, в яких виділено більше ніж один показник, з часом покращувались та модифікувались.

3. Результати роботи практичної реалізації алгоритмів

В цій частині роботи будемо розглядати результати виконання програми для наступних алгоритмів:

1. Дініца (Додаток Б)
2. Форда-Фалкерсона (Додаток В)
3. Едмонда-Карпа (Додаток Г)
4. Проштовхування предпотоку з найвищої мітки (Додаток Д)

Ефективність того чи іншого алгоритму сильно залежить від того, наскільки складною є геометрія мережі та її насиченості параметрами. Обрані нами алгоритми мають наступні оцінки порядку величини часозатратності:

$O(nmU)$ – алгоритм Форда-Фалкерсона,

$O(n^2m)$ – алгоритм Дініца),

$O(n^2\sqrt{m})$ – алгоритм проштовхування предпотоку з найвищої мітки,

$O(nm^2)$ – алгоритм Едмонда-Карпа.

Легко бачити, що в більшості мереж на час роботи алгоритму найбільше впливає кількість дуг у мережі, оскільки $m>n$ завжди, а в більш складних мережах $m>>n$. Для початку розглянемо результати роботи алгоритмів для невеликої мережі, представленої на рисунку 3.1.

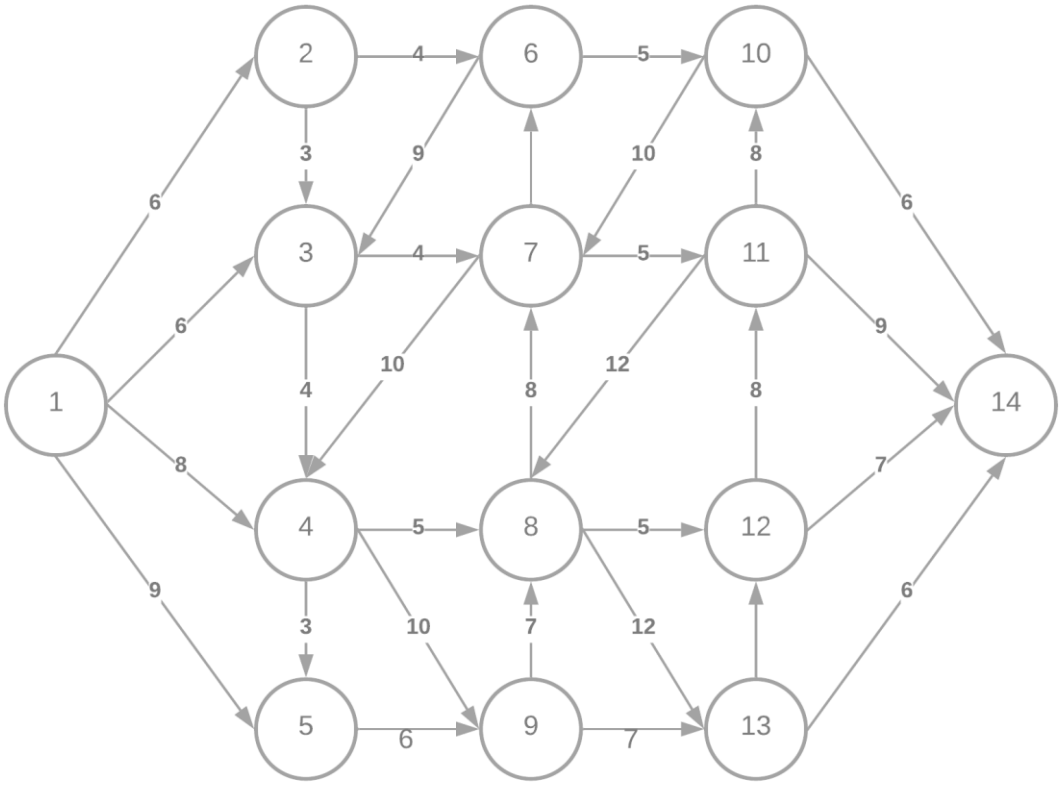


Рис. 3. 1

Результати роботи програми для цієї мережі наводимо у таблиці нижче.

Назва алгоритму	**Час виконання**	
	секунди	**% від макс. часу**
Дініца	0,001	7%
Форда-Фалкерсона	0,0015	100%
Проштовхування предпотоку	0,0008	5,3%
Едмондса-Карпа	0,0012	80%

Значення мксимального потоку = 22. Кількість вершин=14, кількість дуг=21.

В таблиці наведено середні значення часу роботи програми, оскільки для опрацювання такої мережі витрачається мала кількість часу і похибка часу обчислень відіграє значну роль, тому робити повноцінні висновки на основі роботи алгоритмів у цій мережі неможливо.

Для отримання більш об’єктивних результатів нам необхідно розглянути значно більші мережі, для обробки яких потрібно більше часу.

Порівняємо результати виконання програм для різних *n* при постійно зростаючих m>>n:

n	m	Одиниці вимірювання часу роботи алгоритмів	Діниця	Форда-Фалкерсона	Проштовхування предпотоку	Едмондса-Карпа
200	35464	сек	0.55	0.24	0.11	1
		% від макс. часу	55%	24%	11%	100%
300	80111	сек	1,15	2,55	30	5
		% від макс. часу	3,8%	8,5%	100,0%	16,7%
400	143727	сек	3,7	5,2	85	10,7
		% від макс. часу	4,4%	6,1%	100,0%	12,6%
600	322238	сек	11	27,7	317	39
		% від макс. часу	3,5%	8,7%	100,0%	12,3%

Результати виконання програм для відносно малих значень m:

n	m	Одиниці вимірювання часу роботи алгоритмів	Дініца	Форда-Фалкерсона	Проштовхування предпотоку	Едмондса-Карпа
200	14425	сек	0,1	0,2	0,08	0,36
		% від макс. часу	27,8%	55,6%	22,2%	100,0%
300	3115	сек	0,55	0,75	0,2	0,82
		% від макс. часу	67,1%	91,5%	24,4%	100,0%
400	54241	сек	1,5	1,75	75	2,5
		% від макс. часу	2,0%	2,3%	100,0%	3,3%
500	30335	сек	0,9	1,2	135	2,15
		% від макс. часу	0,7%	0,9%	100,0%	1,6%
600	47642	сек	2,1	1,6	148	2,6
		% від макс. часу	1,4%	1,1%	100,0%	1,8%

Для випадків, що ми розглядали можна скласти такий рейтинг теоретичної ефективності алгоритмів:

1. Проштовхування предпотоку найвищої мітки
2. Дініца
3. Форда-Фалкерсона
4. Едмонда-Карпа

Цей рейтинг буде справедливим, оскільки

$$O(nm^2) > O(nmU) > O(n^2m) > O(n^2\sqrt{m}).$$

Проте з результатів роботи програм можна помітити, що алгоритм проштовхування предпотоку, який має найкращий теоретичний час виконання, зі зростанням кількості вершин сильно програє в ефективності іншим алгоритмам з гіршими теоретичними оцінками. Це пов’язано з тим, що для алгоритму проштовхування предпотоків потрібно багато додаткових ресурсів для зберігання в пам’яті інформації про вершини. Для оптимізації цього алгоритму на практиці потрібен комплексний підхід, в той час як простіші алгоритми не використовують таку велику кількість пам’яті і їм не потрібно зберігати та зчитувати інформацію про кожну вершину та їхня практична імплементація набагато простіша. Проте для невеликих мереж алгоритм проштовхування предпотоку показує найкращий час виконання, бо виграш в часі більший за час, який додатково витрачається для зберігання інформації про вершини.

Висновок

В ході роботи ми розглянули принципи роботи більшості основних алгоритмів та продемонстрували найпростіші імплементації алгоритмів (псевдокод). На цій основі мовою пайтон була реалізована програма для послідовного розв’язування задачі про максимальний потік на графах (мережах) різної конфігурації і складності цими алгоритмами почергово.

Під час застосування алгоритмів на практиці дізнались, що теоретичний час їх роботи може сильно відрізнятись від часу роботи реальної програми із застосуванням цього алгоритму через те, що деякі з сучасніших алгоритмів використовують складні структури даних, які потребують додаткових ресурсів для їхнього успішного виконання.

Тобто використання алгоритмів із хорошими теоретичними оцінками не завжди є вдалим вибором для розв’язування задач на практиці. При виборі алгоритмів для конкретного застосування треба завжди звертати увагу на початкові параметри мереж, які будуть досліджуватись.

При розв’язанні задачі у малих мережах суттєвої різниці між часом знаходження розв’язку тим чи іншим алгоритмом немає. Більше того, відмінності в часі виконання для малих мереж настільки мізерні, що їх можна вважати похибкою часу обчислень.

У випадках, коли розміри мережі збільшуються (наприклад при розв’язанні складних транспортних проблем для вуличних мереж, масштабу країни або глобальної логістики), важливу роль відіграє кількість вершин і ребер, адже, як було вище згадано, зберігання інформації про кожну вершину потребує великої кількості ресурсів і це особливо важливо для алгоритмів, що використовують мітки відстані вершин.

Тобто для малих мереж немає різниці, який застосовувати алгоритм, а при розв’язанні задачі для великих мереж треба відштовхуватись під конкретної конфігурації мережі, її параметрів та від того, на що націлена оптимізація того чи іншого алгоритму.

Список використаної літератури

1. О.М. Іксанов, В.І Шевченко. Потоки на мережах. − Наукове видавництво "ТВіМС", 2010. − 46с.
2. Ahuja RK, Magnanti TL, Orlin JB. Network flows: theory, algorithms, and applications. Englewood Cliffs (NJ): Prentice Hall; 1993.
3. Ford LR, Fulkerson DR. Flows in networks. Princeton (NJ): Princeton University Press; 1962.
4. Cheriyan J, Hagerup T, Mehlhorn K. Can a maximum flow be computed in O(nm) time. Proceedings of the ICALP. 1990.
5. Dinic, E. A. (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". Soviet Mathematics - Doklady. Doklady. 11: 1277–1280.
6. Goldberg AV, Rao S. Beyond the flow decomposition barrier. J ACM 1998;45(5):783 –797. DOI: 10.1145/290179.290181.
7. Ford LR, Fulkerson DR. Maximal flow through a network. Can J Math 1956;8:399 –404.
8. Elias P, Feinstein A, Shannon CE. Note on maximum flow through a network. IRE Trans Inf Theory 1956;IT-2:117 –119.
9. Ahuja RK, Orlin JB. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. Nav Res Log Q 1991;38:413 –430.
10. Sleator DD, Tarjan RE. A data structure for dynamic trees. J Comput Syst Sci 1983;26(3):362 –391. DOI: 10.1016/0022- 0000(83)90006-5.
11. Edmonds J, Karp RM. Theoretical improvements in algorithmic efficiency for network flow problems. J ACM 1972;19:248 –264.
12. Dinic EA. Algorithm for solution of a problem of maximum flow in networks with power estimation. Sov Math Dokl 1970;11:1277 –1280.
13. Goldberg AV, Tarjan RE. A new approach to the maximum flow problem. Proceedings of the 19th ACM Symposium on the theory of Computing. 1986. pp. 136 –146.
14. Ahuja RK, Kodialam M, Mishra AK, et al. Computational investigations of maximum flow algorithms. 1997;97:509 –542. DOI: 10.1016/S0377- 2217(96)00269-X.
15. Cormen TH, Leiserson CE, Rivest RL, et al. Introduction to algorithms. 2nd ed. MIT Press
    and McGraw-Hill; 2001. Q10
16. Cheriyan J, Maheshwari SN. Analysis of preflow-push algorithms for maximum network flow. SIAM J Comput 1989;18:1057 –1086.

17. Cherkassky BV, Goldberg AV. On implementing the push-relabel method for the maximum flow problem. Algorithmica 1997;19:390 –410. DOI: 10.1007/PL00009180.
18. Ahuja RK, Orlin JB. A fast and simple algorithm for the maximum flow problem. Oper Res 1989;37:748 –759. DOI: 10.1287/opre.37.5.748.
19. Ahuja RK, Orlin JB, Tarjan RE. Improved time bounds for the maximum flow problem. SIAM J Comput 1989;18:939 –954.
20. Malhotra VM, Pramodh Kumar M, Maneshwari SN. An O(V3) algorithm for finding maximum flows in networks. Inf Process Lett 1978;7:277 –278.
21. Shioura A. The MA-ordering max-flow algorithm is not strongly polynomial for directed networks. Oper Res Lett 2004;32(1):31 –35. DOI: 10.1016/S0167-6377(03)00070-1.
22. Matsuoka Y, Fujishige S. Practical efficiency of maximum flow algorithms using MA orderings and preflows. J Oper Res Soc Jpn 2005;48(4):297 –307.
23. Tarjan R, Ward J, Zhang B, et al. Balancing applied to maximum network flow problems. In: Azar Y, Erlebach T, editors. Volume 4168, ESA 2006. LNCS. Heidelberg: Springer; 2006. pp. 612 –623.
24. Hochbaum DS. The pseudoflow algorithm: a new algorithm for the maximum flow problem. Oper Res 2008;56(4):992 –1009.
25. Chandran BG, Hochbaum DS. A computational study of the pseudoflow and push- relabel algorithms for the maximum flow problem. Oper Res 2009;57(2):358 –376. DOI: 10.1287/opre.1080.0572.
26. Hochbaum DS, Orlin JB. The pseudoflow algo rithm in O(mn log(n2/m)) and O(n3). Berkeley: University of California; 2007.
27. Itai A, Shiloach Y. Maximum flow in planar networks. SIAM J Comput 1979;8:135 –150.
28. Borradaile G, Klein P. An O(n log n) algorithm for maximum st flow in a directed planar graph. J ACM 2009;56(2):1 –30. DOI: 10.1145/1502793.1502798.
29. Fujishige S. A maximum flow algorithm using MA ordering. Oper Res Lett 2003;31:176 –178.
30. Ahuja RK, Orlin JB, Stein C, et al. Improved algorithms for bipartite network flows.
SIAM J Comput 1994;23(5):906 –933. DOI: Q111137/S0097539791199334.
31. King V, Rao S, Tarjan RE. A faster deterministic maximum flow algorithm. Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms. 1992. pp. 157 – 164. DOI: 10.1006/jagm.1994.1044.
32. Goldberg AV. A new max-flow algorithm. Technical Report, MIT/LCD/TM-291. Cambridge (MA): Laboratory for Computer Science, MIT; 1985.

Додаток А

Підсумок аналізу алгоритмів

Алгоритми доповнюючого шляху (ДШ)	**К-сть ітерацій доповнення**	**Час для знаходження ДШ**	**Найкращий час виконання**
Базова реалізація [7,8]	$O(nU)$	$O(m)$	$O(nmU)$
Найкоротшого ДШ [9,10]	$O(nm)$	$O(\log n)$	$O(nm \log n)$ Імплементація динамічного дерева
Масштабування пропускної здатності [9,11]	$O(m \log U)$	$O(n)$	$O(nm \log U)$
Алгоритми блокуючого потоку	**К-сть блокуючих потоків**	**Час знаходження блокуючого потоку**	**Найкращий час виконання**
Блокуючого потоку [12]	$O(n)$	$O(nm)$	$O(n^2m)$
Слітор і Тар'ян [10]	$O(n)$	$O(m \log n)$	$O(nm \log n)$ Імплементація динамічного дерева
Двійковий блокуючий алгоритм	$O(\min(n^{2/3}, \sqrt{m}) \log U)$	$O(m \log(n^2/m))$	$O(\min(n^{2/3}, \sqrt{m})m \log n^2/m \log U)$
Алгоритми проштовхування предпотоку	**Примітки**	**К-сть ненасичуючих проштовхувань**	**Найкращий час виконання**
Базова реалізація [13]	К-сть насичуючих проштов хувань рівна $O(nm)$	$O(n^2m)$	$O(n^2m)$
FIFO [14]	Вибирає активні вузли в порядку перший зайшов – перший вийшов	$O(n^3)$	$O(n^3)$
Найвищої мітки [13]	Вибирає активний вузол з найвищою міткою імплементуючи динамічне дерево	$O(n^3)$	$O(nm \log(n^2/m))$

Найвищої мітки [16]	Вибирає активний вузол з найвищою міткою	$O(n^2\sqrt{m})$	$O(n^2\sqrt{m})$
Масштабування надлишкового потоку [18]	Вибирає вузол великого надлишкового потоку з міткою найменшої відстані		$O(nm+n^2\log U)$
Масштабування надлишкового потоку з динамічними деревами [19]	Розширення викладок з [18]	**Хвильове масштабування:**	**Хвильове масштабування:**
	Використовує масштабуючий фактор *(k) >2*	$O(n^2\sqrt{\log U})$	$O(nm + n^2\sqrt{\log U})$
		Стекове масштабування:	**Стекове масштабування:**
	Вибирає вузол великого надлишкового потоку з міткою найбільшої відстані	$O(\frac{n^2 \log U}{\log \log U})$	$O(nm + \frac{n^2 \log U}{\log \log U})$ Імплементація динамічного дерева виконується за час $O(nm \log(n\sqrt{\log U} / m + 2))$

Додаток Б

Алгоритм Дініца

```
# Алгоритм Дініца
import time


def BFS(C, F, s, t):  # C це матриця пропускних здатностей
    n = len(C)
    queue = []
    queue.append(s)
    global level
    level = n * [0]  # ініціалізація
    level[s] = 1
    while queue:
        k = queue.pop(0)
        for i in range(n):
            if (F[k][i] < C[k][i]) and (level[i] == 0):  # не
проглянуті
                level[i] = level[k] + 1
                queue.append(i)
    return level[t] > 0


# Шукаємо доповнюючий шлях використовуючи пошук в глубину
def Dfs(C, F, k, cp):
    tmp = cp
    if k == len(C) - 1:
        return cp
    for i in range(len(C)):
        if (level[i] == level[k] + 1) and (F[k][i] < C[k][i]):
            f = Dfs(C, F, i, min(tmp, C[k][i] - F[k][i]))
            F[k][i] = F[k][i] + f
            F[i][k] = F[i][k] - f
            tmp = tmp - f
    return cp - tmp


# виконуємо пошук максимального потоку
def MFlow(C, s, t):
    n = len(C)
    F = [n * [0] for i in range(n)]  # F це матриця потоків
    flow = 0
    while (BFS(C, F, s, t)):
        flow = flow + Dfs(C, F, s, 100000)
    return flow
```

```
# матриця з пропускними здатностями

filename = "351.txt"

file = open(filename, encoding="utf-8-sig")

matrix = []
for line in file.readlines():
   temp = line.split(" ")
   try:
      temp = list(map(lambda x: int(x), temp))
   except ValueError:
      print('error')
   matrix.append(temp)



C=matrix

source = 0
sink = 349
print("Алгоритм Дініца")
start_time = time.time()
max_flow_value = MFlow(C, source, sink)
print("Значення максимального потоку", max_flow_value)
print(f"Алгоритм виконав роботу за {time.time() - start_time}
секунд")
```

Додаток В

Алгоритм Форда-Фалкерсона

```
# Алгоритм Форда-Фалкерсона


import time
#шукаємо шлях використовуючи пошук в ширину
def DFS(C, F, s, t):
    stack = [s]
    paths = {s: []}
    if s == t:
        return paths[s]
    while (stack):
        u = stack.pop()
        for v in range(len(C)):
            if (C[u][v] - F[u][v] > 0) and v not in paths:
                paths[v] = paths[u] + [(u, v)]
                if v == t:
                    return paths[v]
                stack.append(v)
    return None


def max_flow(C, s, t):
    n = len(C)  # C це матриця пропускних здатностей
    F = [[0] * n for i in range(n)]
    path = DFS(C, F, s, t)
    while path != None:
        flow = min(C[u][v] - F[u][v] for u, v in path)
        for u, v in path:
            F[u][v] += flow
            F[v][u] -= flow
        path = DFS(C, F, s, t)
    return sum(F[s][i] for i in range(n))


# граф з пропускними здатностями
filename = "351.txt"

file = open(filename, encoding="utf-8-sig")

matrix = []
for line in file.readlines():
   temp = line.split(" ")
   try:
      temp = list(map(lambda x: int(x), temp))
   except ValueError:
```

```
      print('error')
   matrix.append(temp)


C=matrix

source = 0  # A
sink = 349  # F
start_time = time.time()
max_flow_value = max_flow(C, source, sink)
print("Алгоритм Форда-Фалкерсона")
print("Значення максимального потоку", max_flow_value)
print(f"Алгоритм виконав роботу за {time.time() - start_time }
секунд")
```

Додаток Г

Алгоритм Едмонда-Карпа

```
# Алгоритм Едмонда-Карпа
import time
def max_flow(C, s, t):
    n = len(C)  # C це матриця пропускних здатностей
    F = [[0] * n for i in range(n)]
    path = bfs(C, F, s, t)
    #  print path
    while path != None:
        flow = min(C[u][v] - F[u][v] for u, v in path)
        for u, v in path:
            F[u][v] += flow
            F[v][u] -= flow
        path = bfs(C, F, s, t)
    return sum(F[s][i] for i in range(n))


# шукаємо шлях використовуючи пошук в ширину
def bfs(C, F, s, t):
    queue = [s]
    paths = {s: []}
    if s == t:
        return paths[s]
    while queue:
        u = queue.pop(0)
        for v in range(len(C)):
            if (C[u][v] - F[u][v] > 0) and v not in paths:
                paths[v] = paths[u] + [(u, v)]

                if v == t:
                    return paths[v]
                queue.append(v)
    return None



filename = "351.txt"

file = open(filename, encoding="utf-8-sig")

matrix = []
for line in file.readlines():
   temp = line.split(" ")
   try:
      temp = list(map(lambda x: int(x), temp))
   except ValueError:
```

```
      print('error')
   matrix.append(temp)


C=matrix

source = 0  # A
sink = 349  # F
start_time = time.time()
max_flow_value = max_flow(C, source, sink)
print("Алгоритм Едмонда-Карпа")
print("Значення максимального потоку: ", max_flow_value)
print(f"Алгоритм виконав роботу за {time.time() - start_time}
секунд")
```

Додаток Д

Алгоритм проштовхування предпотоку з найвищої мітки

```
# Проштовхування предпотоку з найвищої мітки
import time

def MaxFlow(C, s, t):
    n = len(C)  # C це матриця пропускних здатностей
    F = [[0] * n for i in range(n)]

    # залишкова пропускна здатність з u до v це C[u][v] - F[u][v]
    height = [0] * n  # висота вершини
    excess = [0] * n  # потік у вершину мінус потік з вершини
    seen = [0] * n  # кількість проглянутих сусідв від останього
перепомічання
    #  "черга" для вершин
    nodelist = [i for i in range(n) if i != s and i != t]

    # проштовхування
    def push(u, v):
        send = min(excess[u], C[u][v] - F[u][v])
        F[u][v] += send
        F[v][u] -= send
        excess[u] -= send
        excess[v] += send

    # перепомічання
    def relabel(u):
        # знаходимо нову наменшу мітку для проштовхування,
        # якщо проштовхування взагалі можливо
        min_height = float('inf')
        for v in range(n):
            if C[u][v] - F[u][v] > 0:
                min_height = min(min_height, height[v])
                height[u] = min_height + 1

    def discharge(u):
        while excess[u] > 0:
            if seen[u] < n:  # перевіряємо наступного "сусіда"
                v = seen[u]
                if C[u][v] - F[u][v] > 0 and height[u] > height[v]:
                    push(u, v)
                else:
                    seen[u] += 1
            else:  # перевірили всіх сусідів. перепомічаємо
                relabel(u)
                seen[u] = 0
```

```
    height[s] = n  # найдовший шлях з джерела до стоку має як
мінімум довжину n
    excess[s] = float("inf")  # надсилаємо максимально можливу
кількість потоку до "сусідів" джерела
    for v in range(n):
        push(s, v)

    p = 0
    while p < len(nodelist):
        u = nodelist[p]
        old_height = height[u]
        discharge(u)
        if height[u] > old_height:
            nodelist.insert(0, nodelist.pop(p))  # переміщаємо на
початок списку
            p = 0  # починаємо спочатку списку
        else:
            p += 1
    return sum(F[s])



filename = "351.txt"

file = open(filename, encoding="utf-8-sig")

matrix = []
for line in file.readlines():
   temp = line.split(" ")
   try:
      temp = list(map(lambda x: int(x), temp))
   except ValueError:
      print('error')
   matrix.append(temp)



C=matrix
source = 0
sink = 349
start_time = time.time()
max_flow_value = MaxFlow(C, source, sink)
print("Алгоритм проштовхування предпотоку з найвищої мітки")
print("Значення максимального потоку ", max_flow_value)
print(f"Алгоритм виконав роботу за {time.time() - start_time }
секунд")
```