Київський національний університет імені Тараса Шевченка

Факультет комп’ютерних наук та кібернетики

Кафедра теоретичної кібернетики

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня бакалавра

за спеціальністю 122 Комп’ютерні науки

на тему:

Алгоритми наближеного розв’язання задачі комівояжера з обмеженими відстанями

Студента 4-го курсу

Макарчука Івана Івановича

(підпис)

Науковий керівник:

доцент Ставровський Андрій Борисович

____

(підпис)

Робота заслухана на засіданні кафедри теоретичної кібернетики та рекомендована до захисту в ЕК, протокол № …… від .... 2021р.

Завідувач кафедри Крак Ю.В.

Київ – 2021

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 3
ВСТУП 4
РОЗДІЛ 1. АЛГОРИТМИ РОЗВ'ЯЗАННЯ ЗАДАЧІ КОМІВОЯЖЕРА 8
1.1. Метод повного перебору 8
1.2. Метод найближчого сусіда 11
1.3. Вдосконалений метод найближчого сусіда 16
1.4. Метод гілок і меж 18
РОЗДІЛ 2. ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕВРИСТИЧНИХ МЕТОДІВ РОЗВ'ЯЗАННЯ ЗАДАЧІ КОМІВОЯЖЕРА 26
2.1.Розробка скрипта для паралельних генерації і обчислень алгоритмів 26
2.2. Спосіб генерації матриць часів для завдання комівояжера в умовах міського циклу 27
ВИСНОВКИ 34
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 36
ДОДАТКИ 39

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ОП – оперативна пам’ять

ГА – генетичний алгоритм

ЗК – задача комівояжера

ВСТУП

Оцінка сучасного стану об'єкта дослідження або розробки. У житті сучасних підприємств самого різного роду істотне місце займають транспортні потоки: для кожної компанії є актуальним питання про своєчасну доставку товару споживачам в найкоротші терміни. Для здійснення цього керівництво компанії, по суті, займається розв'язанням так званої задачі комівояжера.

Уявімо собі ситуацію, в тому чи іншому вигляді виникає на будь-якому підприємстві, яке спеціалізується на доставці товарів і вантажів. Припустимо, кур'єру компанії необхідно розвезти продукцію в певну кількість місць і повернутися в офіс. Завдання формулюється так: визначити в якому порядку кур'єр повинен відвідувати клієнтів, щоб дорога зайняла найкоротший термін (всі пункти відвідуються один раз).

Задача комівояжера в такій постановці є NP-важкою. Це означає, що не існує алгоритму, який знаходив би точне розв'язок задачі комівояжера за поліноміальний час. Єдиним же алгоритмом, який в принципі може гарантувати знаходження точного розв'язання, є метод повного перебору. Однак робота програми, що реалізує цей алгоритм, займає адекватне час тільки при дуже малій розмірності вхідних даних (при числі пунктів < 20).

Через відсутність ефективних точних методів розв'язання задачі комівояжера стає необхідним використовувати евристичні методи. Евристичними називають методи, які, спираючись на певну евристику (правило), не завжди виходячи зі строгих математичних принципів, в переважній більшості випадків дають розв'язок, близьке до точного.

Постановка задачі. Розглядається наступне формулювання задачі [16]. Задана множина C, яка складається з m міст, час шляху $d(C_i, C_j) \in Z^+$ між кожною парою міст $C_i, C_j \in C$ і додатнє ціле число B.

Завдання полягає у знаходженні маршруту (перестановки міст), який пролягає через усі міста з C рівно один раз та жодна частина шляху не займає

час більший за B.

Під витратами в даній роботі мається на увазі час. Термін «витрати» був використаний відповідно до поширеного в літературі терміном «матриця витрат», що визначає матрицю, еквівалентну графу в задачі комівояжера [22].

Розв’язок задачі шукається серед гамільтонових циклів, тобто циклів, в яких не відбувається повторного відвідування пунктів.

Мережу доріг для завдання представимо графом $G = (V, E)$, де V - вершини графа, E- ребра графа. Вага ребра $C_{ij}> 0$ еквівалентна часу проїзду між суміжними вершинами графа.

Для зручності програмної реалізації граф представляється у вигляді матриці, розмірність якої відповідає кількості вершин у графі. Елемент матриці d_{ij} ідентичний вазі ребра C_{ij} у графі, він означає відстань між пунктами i та j. При $i = j$, $d_{ij} = M$. Елемент M символізує нескінченність, забороняючи перехід «в себе».

Актуальність роботи та підстави для її виконання. Характерною ознакою будь-якого сучасного мегаполісу є розвинений ринок товарів і послуг. Перед компаніями, що пропонують ці компоненти споживчого ринку своїм клієнтам, стоїть завдання грамотно організувати роботу, наявних в їх розпорядженні транспортних засобів.

У зв'язку з актуальністю проблеми, виникла велика кількість завдань, які можна об'єднати в загальний клас «транспортних задач» [22, 20, 17, 18]. У цей розділ входить величезне різноманіття різних задач, які пов'язані тим, що цільова функція в них носить той чи інший економічний сенс [20]. Серйозними роботами в цьому розділі математики відзначилися А.В. Левітін [21] і група вчених на чолі з Т.Х. Корменом [17]. Ґрунтовну класифікацію транспортних завдань у 2010 році здійснили Е.М. Бронштейн і Т.А. Заіко в статті [12].

Мета й завдання роботи. Метою роботи є реалізація різних евристичних методів розв'язання задачі комівояжера і оцінка якості

отриманого розв’язку для задач з різною кількістю відвідуваних пунктів.

Для реалізації поставленої в роботі мети ставляться такі завдання:

- вивчити єдиний точний метод розв'язання задачі комівояжера - метод повного перебору;
- дослідити деякі евристичні алгоритми та програмно реалізувати їх;
- проаналізувати час і результати роботи програмних реалізацій алгоритмів;
- поліпшити алгоритми, якщо це можливо;
- розробити спосіб обліку змінюваної дорожньої обстановки в методах;
- розробити спосіб генерації вхідних даних для задачі комівояжера з урахуванням специфіки постановки задачі;
- організувати можливість паралельної роботи методів на різних примірниках завдань;
- порівняти час і результати роботи програмних реалізацій вдосконалених алгоритмів з вихідними.

Об’єкт дослідження – процеси розв’язання задачі комівояжера

Предмет дослідження – моделі та методи налаштування параметрів алгоритмів задачі комівояжера.

Для розв’язання поставлених завдань використано комплекс методів дослідження: теоретичні - аналіз, синтез, систематизація, порівняння та узагальнення для опрацювання наукових джерел, визначення сутності та особливостей розв’язання задачі комівояжера; емпіричні - спостереження, вивчення результатів дослідження.

Практична значимість дослідження полягає в тому, що алгоритм методу найближчого сусіда може бути корисний для кур'єра в разі відсутності програмного забезпечення. Пов'язано це з простотою побудови маршруту цим методом і його задовільних результатів.

Апробація роботи та публікації з теми роботи. Основні положення дипломної роботи викладено в доповіді на XIV Міжнародній науково-практичній конференції «Умови економічного зростання в країнах з

ринковою економікою» (Переяслав, 2021 р.). та науково-практичному семінарі ««Актуальні питання туризму, сфери гостинності, обліку, фінансів та підприємництва в умовах інтеграційних та глобалізаційних процесів»» (Київ, 2020 р.).

РОЗДІЛ 1. АЛГОРИТМИ РОЗВ'ЯЗАННЯ ЗАДАЧІ КОМІВОЯЖЕРА

1.1. Метод повного перебору

У статті [12] відзначені найбільш відомі транспортні задачі. Так, наприклад, класична транспортна задача [11, 12] полягає в доставці товару декількома транспортними засобами певній кількості споживачів. Теоретичне завдання про завантаження рюкзака [26, 2, 6], що має своєю метою найбільш вигідне наповнення обмеженого простору, успішно застосовується для оптимального завантаження автотранспорту. Важливі і задачі теорії розкладів [28], роль яких у сучасних реаліях не можна недооцінювати.

Особливе місце в класі транспортних завдань займає задача комівояжера [23], яка полягає в знаходженні самого вигідного маршруту, що проходить через задані пункти. Крім величезної актуальності практичного застосування задача комівояжера має серйозний теоретичний сенс: вона використовується в якості моделі для розробки евристичних алгоритмів різних оптимізаційних задач. Особливе місце в цій галузі відводиться програмі Concorde TSP Solver, розробленої в 1990-х роках групою вчених для розв'язання задачі комівояжера з величезним числом вершин [4, 5]. В даний час велика увага приділяється можливостям розв’язок задачі комівояжера та інших транспортних завдань за допомогою геоінформаційних систем [22, 24].

Задачі комівояжера розрізняються за типом графів, які лежать в їх основі. Так, існують симетричні [22, 27] і асиметричні [16] задачі комівояжера. Також задача комівояжера діляться на статичні і динамічні, в залежності від того, чи можуть додаватися замовлення в процесі роботи алгоритму [12]. Розв’язок в задачі можуть шукатися у вигляді гамільтонових [16] або негамільтонових [13] циклів.

Метод повного перебору, по-іншому званий методом грубої сили (англ.

Brute force), є простим, логічним і широко використовуваним математичним методом [17, 21]. Він може застосовуватись у багатьох, якщо не у всіх, областях математики: задача комівояжера також не є винятком.

Ідея brute force гранично проста: перебираються всілякі розв'язок і з них вибирається розв'язок (або безліч розв'язків), що відповідає умові завдання.

У задачі комівояжера, відповідно, потрібно з різних варіантів об'їзду пунктів вибрати маршрут, який не має у собі ребер з вагою більшою за дане число та займає найкоротший час (або мінімальний за вартістю маршрут).

Величезною перевагою методу повного перебору перед іншими методами розв'язання задачі комівояжера є гарантованість знаходження найкращого маршруту (якщо він існує). Інші методи радять лише «непоганий» маршрут, який зовсім не обов'язково є найкращим. Крім того, до переваг методу належить простота його програмної реалізації.

Однак, у зв'язку з наявністю величезного дефіциту, метод повного перебору вкрай рідко використовується на практиці. Цого недоліком є тимчасова складність алгоритму. Асиметрична задача комівояжера з n відвідуваних пунктів вимагає при повному переборі розгляду (n-1)! шляхів, а факторіал, як можна побачити з таблиці 1.1.1, зростає неймовірно швидко:

Таблиця 1.1.1. Приблизні значення факторіалу

Факторіал числа	5!	10!	15!	20!	25!	30!	40!	50!
Значення	$\sim10^{2}$	$\sim10^{6}$	$\sim10^{12}$	$\sim10^{18}$	$\sim10^{25}$	$\sim10^{32}$	$\sim10^{47}$	$\sim10^{64}$

Тому метод повного перебору може застосовуватися тільки для задач малої розмірності (при розгляді до двох десятків відвідуваних пунктів).

Для реалізації методу повного перебору необхідно навчитися виробляти генерацію всіх перестановок заданого числа елементів, які потім

можна буде використовувати для всіх матриць тієї ж розмірності. Зробити це можна кількома способами, але найпоширеніший (найбільш використовуваний в інших переборних алгоритмах) - це перебір в лексикографічному порядку.

Нехай є деякий алфавіт і набори символів цього алфавіту (букв) - слова. Букви в алфавіті впорядковані. Наприклад, в українському алфавіті порядок букв наступний: a → б → я (символ «→» використовується тут для позначення передування). Якщо впорядковані букви алфавіту, то можна впорядкувати і слова. Наприклад, дано слово u = ($u1$, $u2$, ..., um), що складається з букв $u1$, $u2$, ..., um, і слово v = ($v1$, $v2$, ..., vb). Тоді якщо $u1$ → $v1$, то і u → v. Якщо ж $u1$ = $v1$, то порівнюють другі букви і т.д. Такий порядок слів і називається словниковим. Тому в словниках слово «корова» стоїть раніше слова «корона». Пропуск вважається символом, що передує будь-якій букві алфавіту, тому слово «пар» стоїть раніше слова «парк».

Розглянемо як приклад перебір перестановок з п'яти елементів, позначених цифрами 1...5. Лексикографічно першої перестановкою буде 1-2-3-4-5, другий - 1-2-3-5-4, ..., останньою - 5-4-3-2-1. Для безпосередньої реалізації генерації перестановок потрібно визначити загальний алгоритм перетворення будь-якої перестановки в наступну.

Алгоритм виглядає наступним чином. Нехай дана перестановка 1-3-5-4-2. Потрібно рухатися по перестановці справа наліво, поки вперше не буде виявлено число, менше, ніж попереднє («3» після «5»). Припустимо, що знайдене число розташовується на позиції Pi-1. Міняємо знайдене число місцями з найменшим з великих чисел, які розташовані правіше позиції Pi-1 (таке, очевидно, буде). Після чого, числа правіше позиції Pi-1 необхідно впорядкувати по зростанню. В результаті виходить безпосередньо наступна перестановка: у прикладі це 1-4-2-3-5, за нею йде 1-4-2-5-3 і т.д.

Після генерації всіх перестановок, основна програма проходить по всім перестановкам та рахує суми проїзду між пунктами, результати

порівнюються один з одним, та знаходиться маршрут, відповідний мінімальному часу об'їзду пунктів. Якщо в усіх маршрутах присутнє ребро, що не відповідає умовам задачі, тоді оптимального маршруту не існує.

Програмна реалізація алгоритму brute force була успішно реалізована і наведена в додатку А.

1.2. Метод найближчого сусіда

Метод найближчого сусіда належить до так званих жадібних алгоритмів [1, 13]. Жадібний алгоритм (англ. Greedy algorithm) має на увазі під собою прийняття локально оптимальних розв’язань, допускаючи що остаточний розв’язок також виявиться оптимальним. Існує досить велика кількість завдань, для яких жадібні алгоритми дають оптимальні розв’язки. До таких завдань належать, наприклад, задача про розподіл заявок, задача про розмін монет. Однак для ряду завдань, що належать до класу NP, жадібні алгоритми не дають оптимального розв’язання. Однак в таких задачах, в тому числі і в задачі комівояжера, вони дають досить непогане наближене розв’язання.

Ідею алгоритму найближчого сусіда базується на простому евристичному правилі: якщо ми будемо на кожному кроці відвідувати найближчий пункт, відстань до якого не більша за B (якщо такий існує), то маршрут вийде досить хороший в цілому. Перед комівояжером ставиться завдання відвідувати найближчий з ще не відвіданих пунктів. В алгоритмі існують два важливих обмеження:

1. Недопущення повторного заїзду в пункт. Воно пов'язане з необхідністю (за умовою завдання) знаходження гамільтонового циклу, тобто циклу, в якому всі пункти відвідуються раз.

2. Недопущення передчасного повернення в початковий пункт. Ця

заборона вводиться для запобігання передчасного зациклення маршруту, яке потягне за собою неправильну роботу алгоритму.

Схема алгоритму зображена на рис. 1.2.1.

Рис. 1.2.1. Схема алгоритму найближчого сусіда

Для демонстрації роботи методу приведена реальна матриця часів проїзду між транспортними вузлами м. Київ (див. таблицю 1.2.1).

Таблиця 1.2.1. Матриця часу для демонстрації методу

Найменування	№	0	1	2	3	4
ст. м. Вокзальна	0	M	12	21	19	9
ст. м. Хрещатик	1	10	M	22	17	16
ст. м. Васильківська	2	21	24	M	36	29
ст. м. Либідська	3	24	23	39	M	30
ст. м. Площа Льва Толстого	4	11	21	30	27	M

Значення часів проїзду на автомобілі між парами пунктів були знайдені задані довільним чином. Час, що витрачається на проїзд між пунктами, дано в хвилинах. Нехай B = 25.

На першому і наступних етапах кур'єру необхідно вибирати найближчий з ще не відвіданих пунктів. Також необхідно пам'ятати, що повернення в початковий пункт завчасно неможливий.

Таблиця 1.2.2. Крок перший

Найменування	№	0	1	2	3	4
ст. м. Вокзальна	0	M	12	21	19	9
ст. м. Хрещатик	1	10	M	22	17	16
ст. м. Васильківська	2	21	24	M	36	29
ст. м. Либідська	3	24	23	23	M	30
ст. м. Площа Льва Толстого	4	11	21	30	27	M

На першому кроці (таблиця 1.2.2.) вибір проводиться лише з перевіркою на те, що час проїзду не більший за B, так як повторне відвідування поки не можливо, і повернення в стартовий пункт навіть теоретично не представляється реальним через знаходження в ньому. Тобто перебуваючи в базовому пункті 0 (ст. м. Вокзальна) кур'єр вибере для відвідування пункт 4 (ст. М. «Площа Льва Толстого»), час проїзду до якого (9 хвилин) мінімально.

Таблиця 1.2.3. Крок другий

Найменування	№	0	1	2	3	4
ст. м. Вокзальна	0	M	12	21	19	9
ст. м. Хрещатик	1	10	M	22	17	16
ст. м. Васильківська	2	21	24	M	36	29
ст. м. Либідська	3	24	23	23	M	30
ст. м. Площа Льва Толстого	4	11	21	30	27	M

Опинившись в 4-му пункті для подальшого відвідування кур'єр обере 1-ий пункт. Можливість повернення в «найбільш вигідний» стартовий пункт буде проігнорована з огляду на те, що не можна повернутися на базу, ще не обслуживши всіх клієнтів.

Таблиця 1.2.4. Крок третій

Найменування	№	0	1	2	3	4
ст. м. Вокзальна	0	M	12	21	19	9
ст. м. Хрещатик	1	10	M	22	17	16
ст. м. Васильківська	2	21	24	M	36	29
ст. м. Либідська	3	24	23	23	M	30
ст. м. Площа Льва Толстого	4	11	21	30	27	M

З 1-го пункту кур'єр піде в 3-й, ігноруючи повернення на базу і повторне відвідування 4-го пункту.

Таблиця 1.2.5. Крок четвертий

Найменування	№	0	1	2	3	4
ст. м. Вокзальна	0	M	12	21	19	9
ст. м. Хрещатик	1	10	M	22	17	16
ст. м. Васильківська	2	21	24	M	36	29
ст. м. Либідська	3	24	23	23	M	30
ст. м. Площа Льва Толстого	4	11	21	30	27	M

На цьому етапі так само, як і раніше буде проігнорований передчасний повернення в стартовий пункт і повторні відвідування 1-го і 4-го пунктів. В результаті чого кур'єр піде в єдиний досі не відвіданий пункт 2.

Таблиця 1.2.6. Крок п’ятий

Найменування	№	0	1	2	3	4
ст. м. Вокзальна	0	M	12	21	19	9
ст. м. Хрещатик	1	10	M	22	17	16
ст. м. Васильківська	2	21	24	M	36	29
ст. м. Либідська	3	24	23	23	M	30
ст. м. Площа Льва Толстого	4	11	21	30	27	M

Як тільки всі клієнти обслужені, кур'єр може повертатися в базовий пункт. Це і відбувається на останньому етапі.

В результаті кур'єр пройшов за маршрутом 0-4-1-3-2-0, витративши на об'їзд 9 + 21 + 17 + 23 + 21 = 91 хвилину.

Метод є дуже швидким зважаючи на надзвичайно мале число операцій, необхідних для здійснення його роботи. Однак у випадку, якщо в якомусь пункті найменший час до відвіданого пункта буде більший за B, алгоритм зупинить роботу та шлях не буде знайдений. Окрім цього результати в порівнянні з, наприклад, методом гілок і меж він показує досить скромні.

Час роботи алгоритму для різного числа пунктів наведено в таблиці 1.2.7.

Таблиця 1.2.7. Час роботи програмної реалізації методу найближчого сусіда

Число пунктів	10	25	50	75	100	125	150
Час роботи, с	Менше 0,1 с						

Програмна реалізація алгоритму метода найближчого сусіда була

успішно реалізована і наведена в додатку Б.

1.3. Вдосконалений метод найближчого сусіда

Для того, щоб метод давав більш точні результати, можна ввести в алгоритм невелике вдосконалення, яке дозволить обирати розв’язок з більшого числа маршрутів для однієї матриці часів.

Можливість цього удосконалення можлива в зв'язку з тим, що об'єктом дослідження є задача комівояжера, розв’язок якої має на увазі знаходження замкненого циклу. У завданнях, присвячених знаходженню незамкненого маршруту об'їзду пунктів таке вдосконалення алгоритму неможливо.

Ідея полягає в тому, щоб запускати алгоритм, циклічно змінюючи початковий пункт (початковий для алгоритму, реальний же стартовий пункт залишається колишнім).

Для демонстрації роботи методу була згенерована матриця 7-го порядку. Спосіб генерації буде описаний далі.

Таблиця 1.3.1. Матриця часів для демонстрації вдосконаленого методу

№	0	1	2	3	4	5	6
0	M	75	64	69	60	63	29
1	66	M	35	59	96	80	74
2	48	30	M	26	110	111	57
3	64	53	21	M	14	15	86
4	51	94	109	8	M	45	95
5	57	70	106	6	35	M	76
6	22	71	55	76	92	67	M

Звичайний алгоритм найближчого сусіда для даного екземпляра задачі пропонує наступний маршрут 0-6-2-3-4-5-0, об'їзд якого займе 305 хвилин.

Сформуємо новий ланцюжок з умовним стартовим пунктом 1 (замість 0). Для цього скористаємося методом найближчого сусіда з вхідними даними з таблиці 1.3.1, маючи на увазі перший пункт початковим. В результаті

отримаємо ланцюжок 1-2-3-4-5-0-6-1, час об'їзду якої займе всього 277 хвилин. Легко переконатися, що цей ланцюжок є розв’язанням для вихідної завдання з реальним стартовим пунктом 0. У зв'язку з замкнутістю маршруту, нам потрібно лише пересунути стартовий пункт. Таким чином, 1-2-3-4-5-0-6-1 перетворюється в 0-6-1-2-3-4-5-0.

Ті ж дії повторюються і для інших фіктивних стартових пунктів (див. таблицю 1.3.2).

Таблиця 1.3.2. Утворення нових ланцюжків

Нові ланцюжки	Час
1-2-3-4-5-0-6-1 → 0-6-1-2-3-4-5-0	277
2-3-4-5-0-6-1-2 → 0-6-1-2-3-4-5-0	277
3-4-5-0-6-2-1-3 → 0-6-2-1-3-4-5-0	289
4-3-5-0-6-2-1-4 → 0-6-2-1-4-3-5-0	290
5-3-4-0-6-2-1-5 → 0-6-2-1-5-3-4-0	265
6-0-4-3-5-1-2-6 → 0-4-3-5-1-2-6-0	267

Можна побачити, що для наведеного прикладу завдання маршрут, запропонований модифікованим методом, займає на 40 хвилин менше (відносний виграш у часі -15%). Вдосконалений метод найближчого сусіда за результатами тестування на різних матрицях розмірностями від 10 до 150 дає результати в середньому на 16% краще звичайного методу.

При малих розмірностях (до 10-15 розглянутих в задачі пунктів) метод може не давати поліпшення, зважаючи на малу кількість розглянутих ланцюжків. При розмірностях матриці від 50 відносний виграш стабілізується в межах 16%.

Для того, щоб додатково збільшити ймовірність отримання більш вигідного маршруту, можна розглядати також зворотні до ланцюжків маршрути (просто змінюючи напрямок обходу циклу). В рамках даної роботи

це поліпшення не проводилося.

У зв'язку з незначним часом роботи вихідного методу збільшення кількості операцій в 2n разів на підсумковому часу роботи програми не позначається критично.

Час роботи алгоритму для різної кількості пунктів наведено в таблиці 1.3.3.

Таблиця 1.3.3. Утворення нових ланцюжків

Кількість пунктів	10	25	50	75	100	125	150
Час роботи, с	0,02	0,14	0,5	2,0	4,5	9	15

Програмна реалізація алгоритму метода найближчого сусіда була успішно реалізована і наведена в додатку В.

1.4. Метод гілок і меж

Метод гілок і меж (англ. Branch and bound) - загальний алгоритмічний метод, який широко застосовується для розв'язання завдань комбінаторної оптимізації [18; 7; 8].

По суті, метод є вдосконаленим методом повного перебору з послідовним відсівом рішень, що здаються невигідними. Внаслідок того, що в процесі роботи методу деякі розв'язання не розглядаються, метод гілок і меж не може гарантувати знаходження точного розв'язку задачі. Відкинутий на початковому етапі «невигідний» розв'язок може виявитися в кінці кінців кращим.

Метод вперше був запропонований Лендом і Дойг в 1960 році для розв'язання завдань цілочисельного програмування [7]. У 1963 році групою авторів (Дж. Літтл, К. Мурті, Д. Суїні, К. Керол) була запропонована модифікація методу гілок і меж, розроблена спеціально для розв'язання задачі комівояжера [8]. Згодом цей алгоритм отримав назву «алгоритм

Літтла».

В основі методу гілок і меж лежить ідея послідовного розбиття множини рішень шляхом розгалуження і знаходження оцінок (границь). Схема методу приведена на рис. 1.4.1.

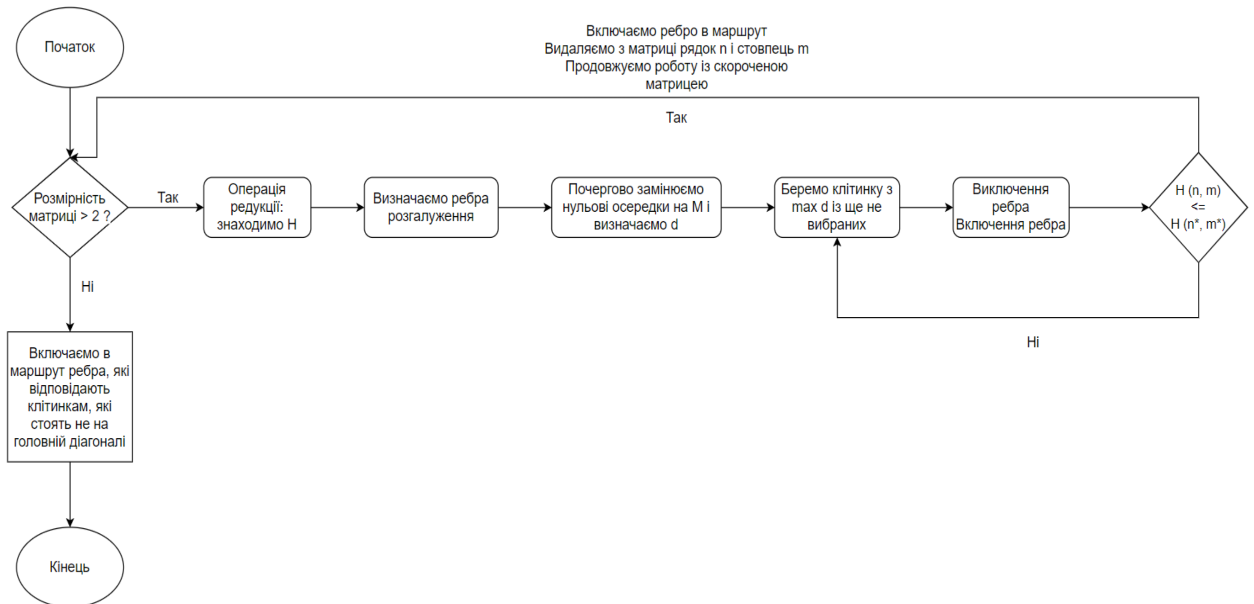

Рис. 1.4.1. Схема методу гілок і меж

Метод гілок і меж вважається універсальним методом, так як він добре підходить для розв'язання антисиметричної задачі комівояжера. У той час як інші методи пристосовані в основному для розв'язання симетричних задач.

Алгоритм складається з двох етапів:

Перший етап (попередній):

Операція приведення матриці і обчислення нижньої оцінки вартості маршруту H.

1. Елементи матриці, що є більшими за B, замінюються на M.
2. Знаходження мінімальних елементів в кожному рядку (константи приведення d_i.

$$d_i = \min_i d_{ij}$$

Таблиця 1.4.1. Крок перший.

№	0	1	2	3	4	d_i

0	M	12	21	19	9	9
1	10	M	22	17	16	10
2	21	24	M	36	29	21
3	24	23	39	M	30	23
4	11	21	30	27	M	11

3. З елементів рядків матриці віднімаються константи приведення di.

Таблиця 1.4.2. Крок другий.

№	0	1	2	3	4
0	M	3	12	10	0
1	0	M	12	7	6
2	0	3	M	15	8
3	1	0	16	M	7
4	0	10	19	16	M

4. Знаходження мінімального елемента в кожному стовпці (константи приведення d_j).

$$d_j = \min_j d_{ij}$$

Таблиця 1.4.3. Крок третій.

№	0	1	2	3	4
0	M	3	12	10	0
1	0	M	12	7	6
2	0	3	M	15	8
3	1	0	16	M	7
4	0	10	19	16	M
d_j	0	0	12	7	0

5. З елементів стовпців матриці віднімаються константи приведення d_j.

Таблиця 1.4.4. Крок четвертий.

№	0	1	2	3	4

0	M	3	0	3	0
1	0	M	0	0	6
2	0	3	M	8	8
3	1	0	4	M	7
4	0	10	7	9	M

В результаті отримуємо наведену матрицю, в якій в кожному рядку і в кожному стовпці є хоча б один нульовий елемент.

6. Обчислення H (нижньої межі) як суму констант приведення d_i і d_j.

$$H = \sum d_i + \sum d_j$$

$$H = 9 + 10 + 21 + 23 + 11 + 0 + 0 + 12 + 7 + 0 = 93$$

Другий етап (основний):

1. Обчислення штрафу за невикористання для кожного нульового елемента наведеної матриці витрат.

Штраф за невикористання елемента з індексом (i, j) означає, що відповідне ребро не буде включено в маршрут, а значить мінімальна вартість невикористання цього ребра дорівнює сумі мінімальних елементів в рядку i і стовпці j.

а) Шукаються всі нульові елементи в наведеній матриці

Таблиця 1.4.5. Крок п'ятий.

№	0	1	2	3	4
0	M	3	0	3	0
1	0	M	0	0	6
2	0	3	M	8	8
3	1	0	4	M	7
4	0	10	7	9	M

б) Нульові елементи по черзі замінюються на M (нескінченність).

Кожному нульового елементу зіставляється штраф (сума констант приведення) за його невикористання.

Таблиця 1.4.6. Крок шостий.

№	0	1	2	3	4	d_i
0	M	3	0 (0)	3	0(6)	0
1	0	M	0	0	6	0
2	0 (3)	3	M	8	8	3
3	1	0 (4)	4	M	7	1
4	0 (7)	10	7	9	M	7
d_j	0	3	0	3	6	

в) Вибирається елемент (з ще не обраних), якому відповідає максимальний штраф.

Вибір елемента з максимальним штрафом обумовлений тим, що виключення з маршруту цього ребра призведе до максимального збільшення вартості оптимального маршруту, знаходження якого є нашою метою.

Найбільша сума констант приведення дорівнює (7 + 0) = 7 для ребра (4, 0), отже, вибираємо цей елемент.

2. Тепер вихідна множина розбивається на дві підмножини - (i *, j *) (містить ребро (i, j)) і (i, j) (що не містить ребро (i, j)). Проводиться обчислення нижніх меж для підмножин.

У нашому випадку множина розбивається на підмножини (4,0) і (4*,0*).

а) Виключення ребра (i *, j *)

Нижня межа для підмножини (i *, j *) дорівнює сумі нижньої межі H вихідної множини і штрафу за невикористання ребра (i, j).

Видалення ребра (4, 0) проводиться шляхом заміни елемента d_{40} = 0 на M, після чого здійснюється чергове приведення матриці відстаней для

утвореної підмножини (4 *, 0 *), в результаті отримаємо скорочену матрицю.

Таблиця 1.4.7. Крок сьомий.

№	0	1	2	3	4	d_i
0	M	3	0	3	0	0
1	0	M	0	0	6	0
2	0	3	M	8	8	0
3	1	0	4	M	7	0
4	M	10	7	9	M	7
d_j	0	0	0	0	0	

Нижня межа гамільтонових циклів цієї підмножини:

$$H\,(4\ *, 0\ *) = 93 + 7 = 100$$

б) Включення ребра (i, j)

При обчисленні нижньої межі для множини (i, j) необхідно взяти до уваги, що якщо ребро (i, j) входить в маршрут, то симетричне ребро (j, i) в маршрут входити не може, тому в матриці цей елемент замінюється на M .

А в зв'язку з тим, що з пункту i ми «вже пішли», а в пункт j ми «вже прийшли», то жодне ребро, що виходить з i, і жодне ребро, що приходить в j, вже використовуватися не можуть, тому викреслюємо з матриці рядок i і стовпець j.

Проводимо операцію приведення для отриманої матриці. Нижня межа для (i, j) буде дорівнювати нижній межі H вихідної множини плюс сума констант приведення для отриманої скороченої матриці.

Включення ребра (4, 0) проводиться шляхом виключення всіх елементів 4-го рядка і 0-го стовпця, в якій елемент d_{04} замінюється на M, для запобігання утворення негамільтонового циклу.

В результаті виходить інша скорочена матриця (4 × 4), яка підлягає операції приведення.

Після операції приведення скорочена матриця буде мати вигляд:

Таблиця 1.4.8. Крок восьмий.

№	1	2	3	4	d_i
0	3	0	3	M	0
1	M	0	0	6	0
2	3	M	8	8	3
3	0	4	M	7	0
d_j	0	0	0	6	

Сума констант приведення скороченої матриці:

$$\sum d_i + \sum d_j = 9$$

3. Якщо $(i, j) \leq (i *, j *)$, то ребро включається в маршрут, і подальша робота методу відбувається з матрицею меншої розмірності. В іншому випадку відбувається повернення до пункту 1в, і розглядається елемент з наступним за спаданням штрафом за невикористання.

Нижня межа підмножини (4,0) дорівнює:

$$H\,(4,0) = 93 + 9 = 102 > 100$$

Оскільки нижня межа цієї підмножини (4, 0) більше, ніж підмножини (4 *, 0 *), то ребро (4, 0) на даний момент в маршрут не включається. Відбувається відкат до пункту 1в, вибирається елемент з наступним за спаданням штрафом.

Процес розгалуження шляхом включення і виключення ребер триває до тих пір, поки не буде розглядатися матриця розмірності 2. З неї додавання ребер в маршрут відбувається тривіальним чином.

Зауваження. При розв’язанні задачі комівояжера методом гілок і меж необхідно перед кожною ітерацією алгоритму забороняти для відвідування ребра, які можуть привести до передчасного зациклення алгоритму. Робиться це шляхом присвоєння відповідного елементу матриці значення M (нескінченність).

В даному випадку алгоритмом в кінці роботи був запропонований

наступний маршрут: 0-4-3-1-2-0. Час об'їзду маршруту, запропонованого методом гілок і меж, займе 102 хвилини.

Середній час роботи програмної реалізації алгоритму для різного числа пунктів представлено в таблиці 1.4.9.

Таблиця 1.4.9. Час роботи програмної реалізації методу гілок і меж.

Кількість пунктів	10	25	50	75	100	125	150
Час роботи, с	0,03	0,2	3,4	35	85	202	436

Програмна реалізація методу гілок та меж була успішно реалізована і наведена в додатку Г.

РОЗДІЛ 2. ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕВРІСТИЧНИХ МЕТОДІВ РОЗВ'ЯЗАННЯ ЗАДАЧІ КОМІВОЯЖЕРА

2.1. Паралелізація генерації і обчислень алгоритмів

Однією з головних цілей даної випускної кваліфікаційної роботи є порівняння і аналіз доцільності застосування алгоритмів розв'язання задачі комівояжера для конкретних постановок задачі. Для того щоб робити такі висновки, необхідно мати у своєму розпорядженні широку вибірку примірників задачі.

Однак генерація і подальша робота алгоритму навіть на одному екземплярі задачі, наприклад, для методу гілок і меж може займати тривалий час. У зв'язку з цим виникає необхідність прискорити процес генерації вибірки. Паралельне обчислення різних примірників завдань дозволяє в рази скоротити час на створення вибірки.

Тому було поставлено завдання зробити так, щоби програма могла одночасно виробляти генерацію декількох екземплярів задачі і виконання алгоритму для них.

Поставлена задача була виконана. Був успішно написаний код (див. додаток Д), за допомогою якого може здійснюватися генерація і подальша робота методу гілок і меж на кількох примірниках завдання у фоновому режимі.

Порівняння часів виконання послідовного алгоритму і алгоритму, що запускається у фоновому режимі наведено в таблиці 2.1.1.

Таблиця 2.1.1. Порівняння часу роботи програми на кластері з використанням скрипта і сумарного часу роботи програми для 8 примірників завдання на 8 потоках.

Кількість пунктів	Паралельно, час роботи	Послідовно, час роботи
100	53 с	7 хв 57 с
125	6 хв 4 с	29 хв 58 с

150	15 хв 47 с	1 год 17 хв
200	29 хв 7 с	3 год 50 хв

Як видно з таблиці, використання кластера для паралельних обчислень дозволило виконувати необхідні дії в середньому в 5-10 разів швидше.

У разі доступності більшого числа нод кластера, можна розраховувати для виконання завдання за рахунок паралельного обчислення більшого числа примірників завдання.

2.2. Порівняльний аналіз евристичних методів розв’язання задачі комівояжера

Для якісного порівняння алгоритмів відповідно до способу, наведеного вище, було згенеровано по 5 матриць розмірності 10, 25, 50, 75, 100, 125 і 150. Часи роботи програмних реалізацій алгоритмів наведені в таблиці 2.2.1. без урахування часу на генерацію матриць.

За підсумками роботи програмних реалізацій в консоль виводився час роботи програми і сумарний час проїзду, запропонованого програмою оптимального маршруту. Також виводилась сама матриця та оптимальний маршрут.

Часи і результати роботи кожного алгоритму для матриць кожної розмірності були усереднені. На підставі результатів були сформовані три таблиці, кожній з яких відповідає графік.

Порівняння часу роботи. У таблиці 2.2.1. порівнюються часи роботи програмних реалізацій алгоритмів і демонструється залежність часу роботи алгоритмів від розмірності вхідних даних.

З таблиці видно, що при числі пунктів більше 75 метод гілок і меж може працювати кілька хвилин. Однак, варто зазначити, що, наприклад, п'ять

хвилин очікування результату програми прийнятні для користувача, якому потрібно об'їхати близько ста пунктів.

Час роботи методу найближчого сусіда становить менше 0,1 с для задач наведених розмірностей, з подальшим збільшенням числа пунктів, істотного збільшення часу роботи не відбудеться.

Таблиця 2.2.1. Порівняння часів роботи програмних реалізацій алгоритмів.

Кількість пунктів	Час роботи програмної реалізації, с		
	Програмна реалізація методу гілок і меж	Програмна реалізація методу найближчого сусіда	Програмна реалізація удосконаленого методу найближчого сусіда
10	0,03	0,01	0,02
25	0,24	0,04	0,14
50	3,44	0,03	0,55
75	35,47	0,05	2,00
100	84,91	0,03	4,57
125	201,62	0,09	8,71
150	436,28	0,10	15,12

Вдосконалений метод найближчого сусіда працює прийнятний час, однак при подальшому збільшенні числа пунктів час роботи алгоритму буде рости істотніше.

Відповідно до таблиці 2.2.1 був побудований графік (див. рис. 2.2.1.)

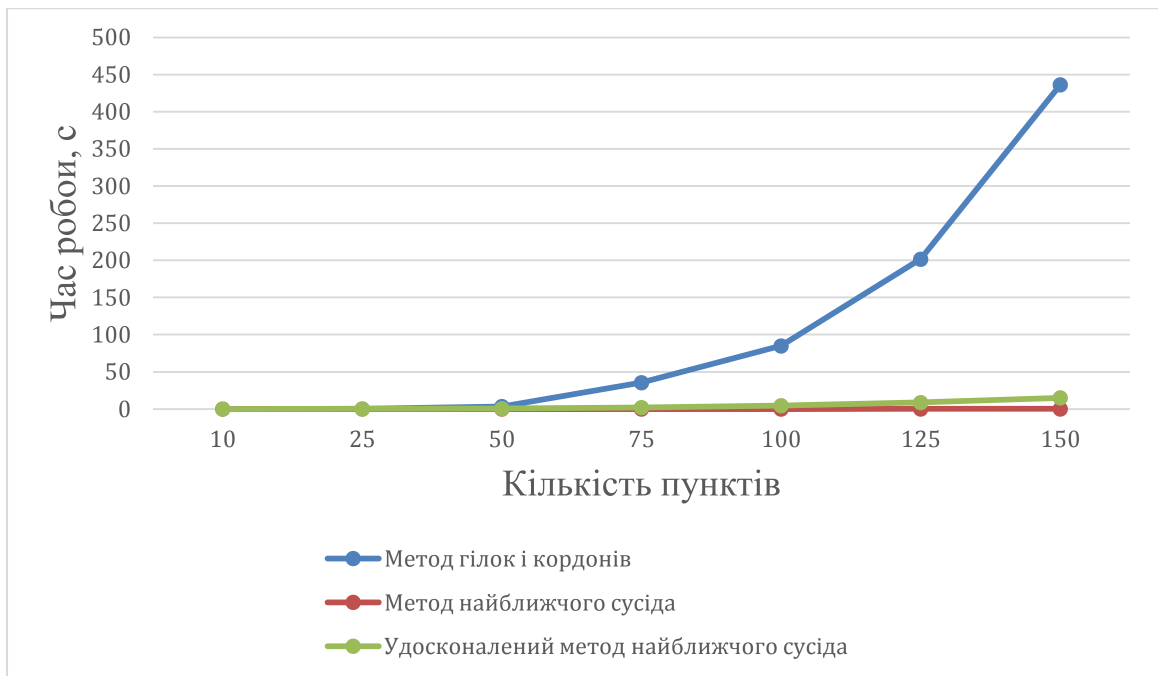

Рис. 2.2.1. Графік залежності часу роботи методів від числа пунктів

З графіку видно, що залежність часу роботи методу гілок і меж від числа пунктів близька до експоненційної. Залежність часів роботи методів найближчого сусіда близька до лінійної, однак, в разі вдосконаленого методу зростання часу роботи при істотному збільшенні числа пунктів буде рости поліноміально.

Порівняння результатів роботи. У наступній таблиці результати роботи програмних реалізацій звичайного і вдосконаленого методів найближчого сусіда порівнюються з результатами роботи програмної реалізації методу гілок і меж. У таблиці наводиться відносний програш в часі об'їзду методів найближчого сусіда методу гілок і меж у відсотковому відношенні.

Таблиця 2.2.2. Відносний програш результатів роботи програмних реалізацій вихідного і вдосконаленого методів найближчого сусіда програмної реалізації методу гілок і меж

Кількість	Програш, %

пунктів	Програмна реалізація методу найближчого сусіда	Програмна реалізація удосконаленого методу найближчого сусіда
10	31,08	9,98
25	29,77	9,98
50	44,13	23,07
75	39,01	21,97
100	37,86	23,16
125	43,17	23,62
150	45,83	26,56
Середнє	38,69	19,76

За даними таблиці 2.2.2. був побудований графік (див. рис. 2.2.2.)

З наведених таблиці і графіка видно, що обидва методи найближчого сусіда досить суттєво програють методу гілок і меж. Однак програш вдосконаленого методу не так важливий. Цікаво, що для одного з примірників завдань (при 25 відвідуваних пунктах) вдосконалений метод найближчого сусіда запропонував більш вигідний маршрут, ніж метод гілок і меж. Також варто відзначити тенденцію до збільшення відносного програшу методів найближчого сусіда зі збільшенням числа розглянутих в задачі пунктів.

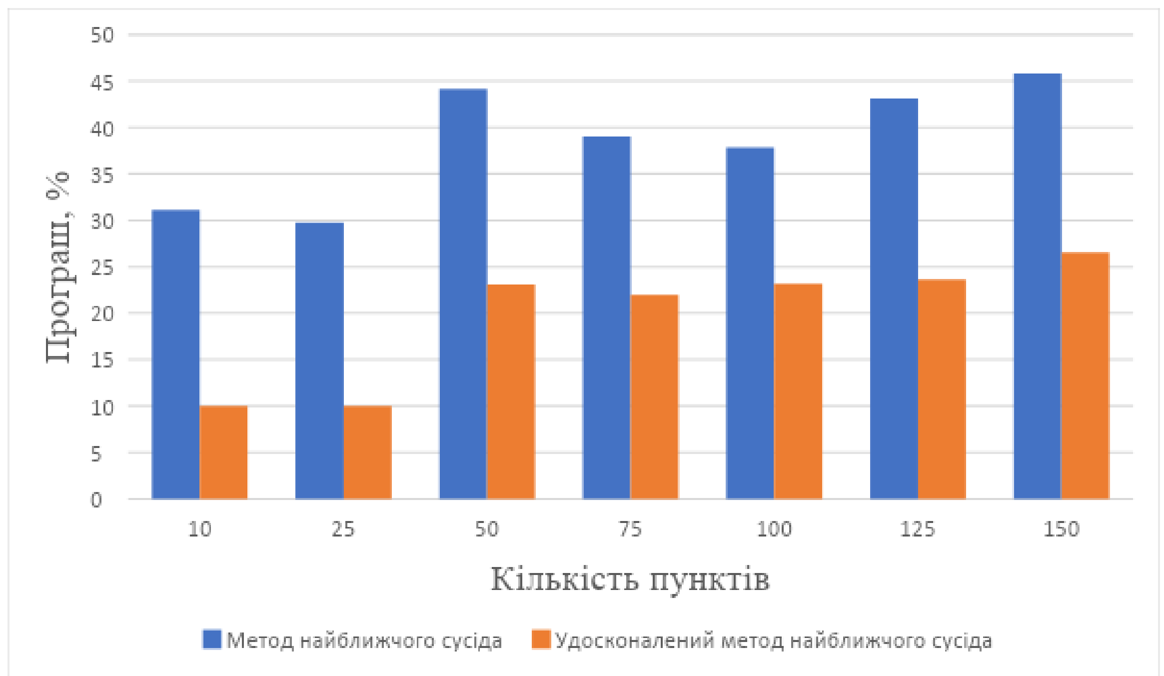

Рис. 2.2.2. Графік залежності відносних програшів результатів роботи рограмних реалізацій вихідного і вдосконаленого методів найближчого сусіда програмної реалізації методу гілок і меж від числа розглянутих в задачі пунктів

У наступній таблиці порівнюються вдосконалений і вихідний методи найближчого сусіда. Результати представлені у вигляді програшу за часом об'їзду в процентному відношенні звичайного методу вдосконаленому.

Таблиця 2.2.3. Відносний програш результатів роботи програмної реалізації вихідного методу найближчого сусіда програмної реалізації вдосконаленого методу.

Кількість пунктів	Програш, %
10	19,93
25	17,96
50	16,93
75	14,30
100	12,04
125	15,81

150	15,17
Середнє	16,00

Відповідно до таблиці 2.2.3 був побудований графік (див. рис. 2.2.3.).

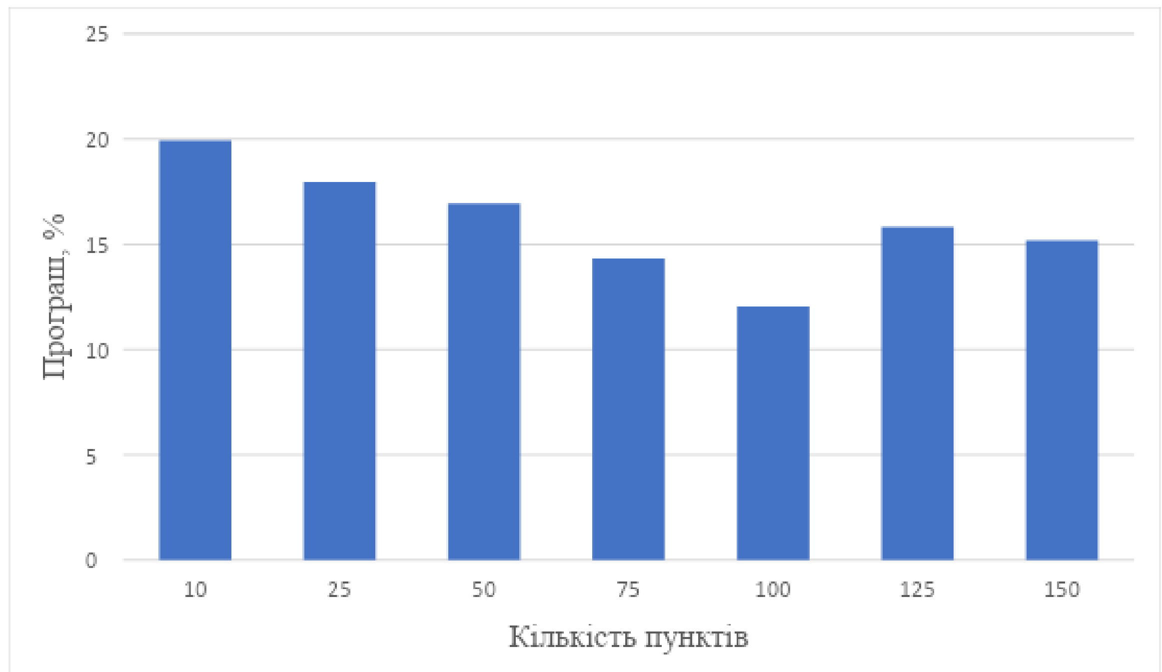

Рис. 2.2.3. Графік залежності відносного програшу результатів роботи програмної реалізації вихідного методу найближчого сусіда програмної реалізації вдосконаленого методу від числа розглянутих в задачі пунктів.

З наведених таблиці і графіка видно, що вдосконалений метод найближчого сусіда дає точніші результати, ніж вихідний метод. Також варто відзначити, що виграш вдосконаленого методу фактично не залежить від розмірності задачі і становить близько 16 відсотків.

Всі програмні реалізації тестувалися на переносному персональному комп'ютері Dell Vostro 15 3000 з наступними технічними характеристиками:

Таблиця 2.2.4. Технічні характеристики переносного ПК

Процесор	Intel Core i5-1135G7
Тактова частота	2.40 ГГц
Кількість ядер	4

Об’єм ОП	8 Гб
Тип ОП	DDR4
Частота ОП	2 133 МГц

ВИСНОВКИ

В дипломній роботі досліджено алгоритми наближеного розв’язання задачі комівояжера з обмеженими відстанями.

Проаналізовано наукову літературу, яка присвячена цій проблемі, та виявлено, що алгоритми наближеного розв’язання задачі комівояжера з обмеженими відстанями є актуальними, адже це питання ще не є достатньо розглянутим та дослідженим.

Успішно розроблений метод моделювання завантаженості транспортної мережі, який згодом може застосовуватися і для програмних реалізацій інших транспортних завдань, що мають на увазі зміну дорожньої ситуації.

Метод гілок і меж підтвердив, що він є одним з найуспішніших методів для розв’язання задачі комівояжера. Той факт, що метод працює трохи довше своїх суперників на практиці істотно не позначиться, тому що час роботи програми при великій кількості пунктів мізерно мало в порівнянні з виграшем в часі об'їзду, яке він дає.

Програмна реалізація методу найближчого сусіда, як і передбачалося, працює швидко, однак маршрути, запропоновані методом, можуть бути корисні тільки при надзвичайно малих розмірності вхідних даних. Тільки в цих випадках маршрут, пропонований методом найближчого сусіда, показує адекватний в абсолютному значенні час. Також алгоритм методу найближчого сусіда може бути корисний для кур'єра в разі відсутності програмного забезпечення. Пов'язано це з простотою побудови маршруту цим методом і його задовільних результатів.

З результатів роботи програмної реалізації вдосконаленого методу найближчого сусіда видно, що модифікація дійсно зробила метод більш ефективним. Вдосконалений метод дає приблизно на 16% кращі результати і на деяких екземплярах завдання може скласти конкуренцію методу гілок і

меж. Час роботи програмної реалізації методу робить його зручним у використанні мобільними користувачами.

Продемонстровано, що евристичні методи дозволяють успішно вирішувати завдання будь-яких розмірностей. Поставлені в даній роботі мети і завдання були успішно виконані:

- вивчена література, присвячена завданню, відзначені основні напрямки сучасних досліджень в даній області;
- вивчений метод повного перебору; показано, що він не може успішно застосовуватися на практиці з огляду на швидке зростання часу роботи його програмної реалізації при збільшенні кількості відвідуваних пунктів;
- детально описані метод найближчого сусіда і метод гілок і меж, проілюстровані кроки роботи алгоритмів, наведені їх логічні схеми, обидва методи успішно реалізовані на мові C ++;
- на основі широкої вибірки згенерованих матриць побудовані порівняльні таблиці і графіки, на підставі яких були зроблені висновки про корисність застосування алгоритмів на практиці;
- вироблено удосконалення методу найближчого сусіда шляхом введення в розгляд фіктивних стартових пунктів; на підставі застосування вдосконаленого методу до різних екземплярів завдання, зроблено висновок про те, що ефективність методу підвищилася;
- розроблена модель обліку зміни дорожньої ситуації, проведена демонстрація роботи моделі на прикладі;
- розроблений зручний спосіб генерації вхідних даних, обґрунтована його адекватність специфіці даної постановки завдання; розроблений скрипт, що дозволяє генерувати вхідні дані і здійснювати програмне розв’язання задачі одночасно для декількох екземплярів задачі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bang-Jensen J., Gutin G., Yeo A. When the greedy algorithm fails // Discrete Optimization. 2004 Vol. 1, No 2. P. 121-127.

2. Berbeglia G., Cordeau J.F., Gribkovskaia I., Laporte G. Static pickup and delivery problems: A classification scheme and survey // TOP. 2007. V. 15. No 1. P. 1-31.

3. M. Dorigo, V. Maniezzo & A. Colorni. Ant System: optimization by a colony of cooperating agents // IEEE transactions on systems, man, and cybernetics-part B. 1996 No. 26 (1). P. 29-41.

4. Gutin G., Jakubowicz H., Ronen, Sh., Zverovitch A. Seismic vessel problem, Communications in DQM, 2005 No. 8. P. 13-20.

5. Hahsler M., Hornik K. TSP – Infrastructure for the traveling salesperson problem // Journal of statistical software, 2007 No. 23 (2). P. 1-21.

6. Kellerer H., Pferschy U., Pisinger D. Knapsack problems. // Springer-Verlag, 2003. 548 c.

7. Land A. H., Doig A. G. An automatic method of solving discrete programming problems // Econometrica. 1960 Vol. 28. P. 497-520.

8. Little J. D. C., Murty K. G., Sweeney D. W., Karel C. An algorithm for the traveling salesman problem // Operations Research. 1963 Vol. 11, No 6. P. 972-989.

9. Paar C., Pelzl J. Understanding cryptography: a textbook for students and practitioners. Berlin: Springer, 2010. 372 c.

10. Parragh S., Doerner K., Hartl R. A survey on pickup and delivery problems. Part II: Transportations between customers and depot // J. Betriebswirtschaft. 2008. V. 58. No 2. P. 81-117.

11. Большакова Е. И., Мальковский М. Г., Пильщиков В. Н. Искусственный интеллект. Алгоритмы эвристического поиска [Електронний ресурс] URL: http://recyclebin.ru/BMK/II/ii.html

12. Бронштейн Е. М., Заико Т. А. Детерменированные

оптимизационные задачи транспортной логистики // Автоматика и телемеханика, 2010. №10. С. 133-147.

13. Вишняков П.О. Планирование маршрутов с использованием модифицированного метода «ближайшего соседа» //Математические методы в технике и технологиях - ММТТ. 2014. № 6 (65). С. 63-67.

14. Гладков Л.А., Баринов С.В., Разработка новых подходов к решению транспортной задачи с использованием геоинформационных технологий // Известия ТРТУ. Тематический выпуск «Интеллектуальные САПР», 2009. №2. С. 141-144.

15. Гладков Л.А., Гладкова Н.В. Особенности использования нечетких генетических алгоритмов для решения задач оптимизации и управления // Известия ЮФУ. Технические науки. 2009. № 4 (93). С. 130-136.

16. Гончарова А.Б., Поборчий И.В. Исследование методов решения задачи коммивояжера при управлении транспортными потоками предприятия // Процессы глобальной экономики: Сборник научных трудов XX Всероссийской научно-практической конференции с международным участием. СПб.: Издательство Политехнического университета, 2015. С. 318-324.

М.Гэри, Д.Джонсон Построение сетей. Задачи о маршрутах //Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982. С. 267

17. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Р., Штайн К. Алгоритмы. Построение и анализ. 2 изд. М.: Вильямс, 2012. 1296 с.

18. Костюк Ю. Л. Эффективная реализация алгоритма решения задачи коммивояжера методом ветвей и границ // Прикладная дискретная математика. Вычислительные методы в дискретной математике, 2010. №2 (20). С. 78-90.

19. Курейчик В.В., Курейчик В.М. Генетический алгоритм определения пути коммивояжера // Известия Российской академии наук. Теория и системы управления. 2006. № 1. С. 94-100.

20. Курейчик В.М., Кажаров А.А. Муравьиные алгоритмы для решения

транспортных задач. // Известия РАН. Теория и системы управления. – 2010. № 1. С. 32-45.

21. Левитин А. В. Алгоритмы: введение в разработку и анализ. М.: Вильямс, 2006. 576 с.

22. Маций О.Б. Повышение точности симметричной задачи класса коммивояжера большой размерности // Вестник Харьковского национального автомобильно-дорожного университета. 2011. № 55. С. 100-102.

23. Мудров В. И. Задача о коммивояжёре. М.: Знание, 1969. 62 с.

24. Пекарская С.С. Расчёт весовых коэффициентов для нахождения кратчайшего по времени пути // Сборник докладов Всероссийской научно-практической конференции «Современные техника и технологии», Томск, 2012.

25. Пекарская С.С. Построение рационального маршрута при решении задач транспортной логистики с учетом нагрузки в дорожной сети // Сборник «Перспективы развития информационных технологий» Труды Всероссийской молодежной научно-практической конференции. Кузбасский государственный технический университет имени Т.Ф. Горбачева, Международный научно-образовательный центр КузГТУ-Arena Multimedia. 2014. С. 382-383.

26. Седжвик Р. Фундаментальные алгоритмы на C++. Части 1-4. Анализ. Структуры данных. Сортировка. Поиск = Algorithms in C++, СПб: ДиаСофт, 2002. 688 с.

27. Ураков А.Р., Михтанюк А.А. Оценка количества вариантов обхода в задаче коммивояжера с дополнительными условиями //Глобальный научный потенциал, 2012. № 21. С. 82-86.

28. Хухрянская Е.С., Юдина Н.Ю., Ющенко Е.В. Анализ возможностей применения методов теории расписаний к задачам деревообрабатывающих производств и их формализация // Лесотехнический журнал. 2011. № 3. С. 37-40.

ДОДАТКИ

ДОДАТОК А

Код для запуску алгоритму brute force.

```
public class BruteForceAlgorithm {

   public Path findBestPath(Graph graph) {
        Path shortestPath = graph.getPermutations().get(0);

        int minTime = Integer.MAX_VALUE;

        for (Path currentPath : graph.getPermutations()) {
            currentPath.setTime(graph.calculatePathTime(currentPath.getEdges()
));

            if (currentPath.getTime() > 0 && currentPath.getTime() < minTime)
{
                shortestPath = currentPath;
                minTime = currentPath.getTime();
            }
        }

        return minTime == Integer.MAX_VALUE ? null : shortestPath;
    }
}
```

ДОДАТОК Б

Код для запуску алгоритму найближчого сусіда.

```
public class NearestNeighbourAlgorithm {

   public Path findBestPath(Graph graph) {
       return findBestPath(graph, 0);
   }

   public Path findBestPath(Graph graph, int startPosition) {
       List<Integer> path = new ArrayList<>();
       path.add(startPosition);

       int pathTime = 0;
       int next = startPosition;

       for (int i = 0; i < graph.getCount() - 1; i++) {
           int minEdge = Integer.MAX_VALUE;
           int current = next;
           graph.enter(current);

           for (int j = 0; j < graph.getCount(); j++) {
               int edge = graph.getEdge(current, j);

               if (!graph.getMarks()[j] && edge < minEdge && edge <=
graph.getUpperBound()) {
                   minEdge = edge;
                   next = j;
                   graph.leave(j);
               }
           }
           if (current == next) {
               return null;
           }

           graph.enter(next);
           path.add(next);
           pathTime += minEdge;
       }

       int egde = graph.getEdge(startPosition, next);
       pathTime += egde;

       graph.leaveAll();

       if (egde > graph.getUpperBound()) {
           return null;
       }

       int zeroPosition = path.indexOf(0);

       for (int i = 0; i < zeroPosition; i++) {
           path.add(path.remove(0));
       }

       return new Path(path, pathTime);
```

```
    }
}
```

ДОДАТОК В

Код для запуску удосконаленого алгоритму найближчого сусіда.

```
public class NearestNeighbourEnhancedAlgorithm {

   public Path findBestPath(Graph graph) {
       NearestNeighbourAlgorithm nearestNeighbourAlgorithm = new
NearestNeighbourAlgorithm();

       Path shortestPath = nearestNeighbourAlgorithm.findBestPath(graph);
       int minTime = shortestPath == null ? Integer.MAX_VALUE :
shortestPath.getTime();

       for (int i = 1; i < graph.getCount(); i++) {
           Path currentPath = nearestNeighbourAlgorithm.findBestPath(graph,
i);

           if (currentPath != null && currentPath.getTime() < minTime) {
               shortestPath = currentPath;
               minTime = currentPath.getTime();
           }
       }

       return minTime == Integer.MAX_VALUE ? null : shortestPath;
   }
}
```

ДОДАТОК Г

Код для запуску методу гілок і меж.

```
public class BranchAndBoundAlgorithm {

    private final List<Edge> edges = new ArrayList<>();
    private boolean[][] edgesAboveBound;

    public Path findBestPath(Graph graph) {
        int lowerBound = 0;
        int[][] matrix = copyMatrix(graph.getMatrix());

        removeEdgesBiggerThanUpperBound(matrix, graph.getUpperBound());

        List<Integer> availableRows = new ArrayList<>();
        List<Integer> availableCols = new ArrayList<>();

        for (int i = 0; i < graph.getCount(); i++) {
            availableRows.add(i);
            availableCols.add(i);
        }

        handleMatrix(matrix, availableRows, availableCols, lowerBound);

        if (edges.size() != graph.getCount()) {
            return null;
        }

        List<Integer> path = new ArrayList<>();
        int last = 0;

        for (Edge e : edges) {
            if (e.x == 0) {
                path.add(e.x);
                last = e.y;
                break;
            }
        }

        while (last != 0) {
            path.add(last);

            for (Edge e : edges) {
                if (e.x == last) {
                    last = e.y;
                    break;
                }
            }
        }

        return new Path(path, graph.calculatePathTime(path));
    }

    private void handleMatrix(int[][] matrix,
                              List<Integer> availableRows,
                              List<Integer> availableCols,
                              int lowerBound) {
        if (availableRows.size() == 2) {
```

```
            for (int i : availableRows) {
                for (int j : availableCols) {
                    if (matrix[i][j] != Integer.MAX_VALUE) {
                        edges.add(new Edge(i, j));
                    }
                }
            }

            return;
        }

        lowerBound += reduceMatrix(matrix, availableRows, availableCols);

        Queue<Edge> zeros = new PriorityQueue<>((a, b) -> b.coef - a.coef);

        for (int i : availableRows) {
            for (int j : availableCols) {
                if (matrix[i][j] == 0) {
                    int coef = getCoefficient(matrix, availableRows,
availableCols, i, j);

                    zeros.add(new Edge(i, j, coef));
                }
            }
        }

        while (!zeros.isEmpty()) {
            Edge edge = zeros.poll();

            int[][] matrixIncluding = copyMatrix(matrix);

            List<Integer> availableRowsIncluding = new
ArrayList<>(availableRows);
            List<Integer> availableColsIncluding = new
ArrayList<>(availableCols);

            availableRowsIncluding.remove(Integer.valueOf(edge.x));
            availableColsIncluding.remove(Integer.valueOf(edge.y));

            int tmp = matrix[edge.x][edge.y];
            matrix[edge.x][edge.y] = Integer.MAX_VALUE;

            int x = findIndexOfRowOrColumnWithoutInfinities(matrixIncluding,
                availableRowsIncluding, availableColsIncluding, true);
            int y = findIndexOfRowOrColumnWithoutInfinities(matrixIncluding,
                availableColsIncluding, availableRowsIncluding, false);

            if (x >= 0 && y >= 0) {
                matrixIncluding[x][y] = Integer.MAX_VALUE;
                edgesAboveBound[x][y] = false;
            }

            int excludingSubsetBound = lowerBound +
                reduceMatrix(matrix, availableRows, availableCols);
            int includingSubsetBound = lowerBound +
                reduceMatrix(matrixIncluding, availableRowsIncluding,
availableColsIncluding);

            if (includingSubsetBound <= excludingSubsetBound) {
                edges.add(edge);
```

```
                handleMatrix(matrixIncluding, availableRowsIncluding,
availableColsIncluding, includingSubsetBound);
                return;
            }

            matrix[edge.x][edge.y] = tmp;
        }
    }

    private int findIndexOfRowOrColumnWithoutInfinities(int[][] matrix,
                                                        List<Integer> list1,
                                                        List<Integer> list2,
                                                        boolean
isList1ForRows) {
        for (int i : list1) {
            boolean isWithoutInfinities = true;

            for (int j : list2) {
                int x = isList1ForRows ? i : j;
                int y = isList1ForRows ? j : i;

                if (matrix[x][y] == Integer.MAX_VALUE &&
!edgesAboveBound[x][y]) {
                    isWithoutInfinities = false;
                    break;
                }
            }

            if (isWithoutInfinities) {
                return i;
            }
        }

        return -1;
    }

    private int reduceMatrix(int[][] matrix,
                             List<Integer> availableRows,
                             List<Integer> availableCols) {
        int subtractSum = 0;
        int size = matrix.length;

        int[] minRow = new int[size];
        int[] minCol = new int[size];

        Arrays.fill(minRow, Integer.MAX_VALUE);
        Arrays.fill(minCol, Integer.MAX_VALUE);

        for (int i : availableRows) {
            for (int j : availableCols) {
                if (matrix[i][j] < minRow[i]) {
                    minRow[i] = matrix[i][j];
                }
            }

            for (int j : availableCols) {
                if (matrix[i][j] < Integer.MAX_VALUE) {
                    matrix[i][j] -= minRow[i];
                }
```

```
                if ((matrix[i][j] < minCol[j])) {
                    minCol[j] = matrix[i][j];
                }
            }
        }

        for (int j : availableCols) {
            for (int i : availableRows) {
                if (matrix[i][j] < Integer.MAX_VALUE) {
                    matrix[i][j] -= minCol[j];
                }
            }
        }

        for (int i : minRow) {
            if (i < Integer.MAX_VALUE) {
                subtractSum += i;
            }
        }

        for (int i : minCol) {
            if (i < Integer.MAX_VALUE) {
                subtractSum += i;
            }
        }

        return subtractSum;
    }

    private int getCoefficient(int[][] matrix,
                               List<Integer> availableRows,
                               List<Integer> availableCols,
                               int row,
                               int col) {
        int rowMin = Integer.MAX_VALUE;
        int colMin = Integer.MAX_VALUE;

        for (int i : availableRows) {
            if (i != row) {
                rowMin = min(rowMin, matrix[i][col]);
            }
        }

        for (int i : availableCols) {
            if (i != col) {
                colMin = min(colMin, matrix[row][i]);
            }
        }

        return (rowMin == Integer.MAX_VALUE ? 0 : rowMin) + (colMin ==
Integer.MAX_VALUE ? 0 : colMin);
    }

    private void removeEdgesBiggerThanUpperBound(int[][] matrix, int
upperBound) {
        edgesAboveBound = new boolean[matrix.length][];

        for (int i = 0; i < matrix.length; i++) {
            edgesAboveBound[i] = new boolean[matrix.length];
```

```
            for (int j = 0; j < matrix.length; j++) {
                if (matrix[i][j] > upperBound && i != j) {
                    matrix[i][j] = Integer.MAX_VALUE;
                    edgesAboveBound[i][j] = true;
                }
            }
        }
    }

    private int[][] copyMatrix(int[][] from) {
        int[][] to = new int[from.length][];

        for (int i = 0; i < from.length; i++) {
            to[i] = from[i].clone();
        }

        return to;
    }

    private static class Edge {
        int x;
        int y;
        int coef;

        public Edge(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public Edge(int x, int y, int coef) {
            this.x = x;
            this.y = y;
            this.coef = coef;
        }
    }
}
```

ДОДАТОК Д

Код для паралельного запуску розв’язання екземплярів задач у фоновому режимі.

```
List<Thread> threads = new ArrayList<>();

for (int i = 0; i < threadCount; i++) {
   threads.add(new Thread(() -> {
       String name = Thread.currentThread().getName();
       System.out.println(name + ": started");

       for (int j = 0; j < graphCount; j++) {
           long start;

           String logTemplate = name + ": graph " + j + " ";
           System.out.println(logTemplate + "started");

           Graph graph = new Graph(size, maxDistance, upperBound,
graphForPermutations.getPermutations());
           graph.print(j);

           start = System.currentTimeMillis();
           System.out.println(logTemplate +
               new NearestNeighbourAlgorithm().findBestPath(graph) +
               " - Nearest Neighbour, time: " + (System.currentTimeMillis() -
start) + "ms");

           start = System.currentTimeMillis();
           System.out.println(logTemplate +
               new NearestNeighbourEnhancedAlgorithm().findBestPath(graph) +
               " - Nearest Neighbour Enhanced, time: " +
(System.currentTimeMillis() - start) + "ms");

           start = System.currentTimeMillis();
           System.out.println(logTemplate +
               new BranchAndBoundAlgorithm().findBestPath(graph) +
               " - Branch and Bound, time: " + (System.currentTimeMillis() -
start) + "ms");

           start = System.currentTimeMillis();
           System.out.println(logTemplate +
               new BruteForceAlgorithm().findBestPath(graph) +
               " - Brute Force, time: " + (System.currentTimeMillis() - start) +
"ms");
```

```
            System.out.println(logTemplate + "finished");
        }

        System.out.println(name + ": finished");
    }));

    threads.get(i).start();
}

for (Thread t : threads) {
    t.join();
}
```