Міністерство освіти і науки України
Український державний університет науки і технологій

Комп'ютерні технології і системи
(назва факультету)

Електронні обчислювальні машини
(повна назва кафедри)

19.01.2024

Пояснювальна записка

до кваліфікаційної роботи
магістра
(ступінь вищої освіти)

на тему: **Дослідження алгоритмів знаходження дерева Штейнера для побудови мереж в розподілених комп'ютерних системах**
за освітньою програмою Комп'ютерна інженерія
зі спеціальності: 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

Виконав: студент групи: КС2221

(підпис студента) / Олег ГЛУШКОВ /
(Ім'я ПРІЗВИЩЕ)

Керівник: /
(підпис) / професор, Анатолій КОСОЛАПОВ
(посада, Ім'я ПРІЗВИЩЕ)

Нормоконтролер: /
(підпис) /доцент Володимир ШАПОВАЛОВ
(посада, Ім'я ПРІЗВИЩЕ)

Консультанти:

(назва розділу) (підпис) / (посада, Ім'я ПРІЗВИЩЕ) /

(назва розділу) (підпис) / (посада, Ім'я ПРІЗВИЩЕ) /

(назва розділу) (підпис) / (посада, Ім'я ПРІЗВИЩЕ) /

(назва розділу) (підпис) / (посада, Ім'я ПРІЗВИЩЕ) /

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

Студент
(підпис)

Дніпро – 2024 рік

Ministry of Education and Science of Ukraine
Ukrainian State University of Science and Technologies

Computer Technologies and Systems
(faculty)

Electronic Computers
(department)

Explanatory Note
to Master's Thesis
(higher education degree)

on the topic: Research of finding Steiner's tree Algorithms for Building Networks in Distributed Computer Systems

according to educational curriculum Computer Engineering

in the Speciality: 123 Computer Engineering
(speciality and its code)

Done by the student of the group: KC2221 / Oleh Hlushkov /
(name, surname)

Scientific Supervisor: / Professor, Anatoliy Kosolapov /
(position, name, surname)

Normative controller: /Docent, Volodymyr Shapovalov /
(position, name, surname)

Supervisors

(Chapter title heading)	/ / (position, name, surname)
(Chapter title heading)	/ / (position, name, surname)
(Chapter title heading)	/ / (position, name, surname)
(Chapter title heading)	/ / (position, name, surname)

Dnipro – 2024

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: Комп'ютерні технології і системи
Кафедра: Електронні обчислювальні системи
Рівень вищої освіти: Другий (магістерський)
Освітня програма: Комп'ютерна інженерія
Спеціальність: 123 Комп'ютерна інженерія
(шифр та назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ЕОМ
_____ Ігор Жуковицький
(підпис) (Ім'я ПРІЗВИЩЕ)
Дата _____

ЗАВДАННЯ

на кваліфікаційну роботу Магістр
(ступінь вищої освіти)

студенту **Глушкову Олегу Володимировичу**
(Прізвище, Ім'я По батькові)

1. Тема роботи: **Дослідження алгоритмів знаходження дерева Штейнера для побудови мереж в розподілених комп'ютерних системах**

Керівник роботи: **Косолапов А. А., д.т.н., професор**
(Прізвище, Ім'я, По батькові, науковий ступінь, вчене звання)

затверджені наказом від 28. 04. 2023 р. № 333 ст

2. Строк подання студентом роботи: 15.01.2024 р.
3. Вихідні дані до роботи:
4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати):
4.1. Вступ
4.2. Аналіз відомих алгоритмів побудови дерева Штейнера та їх класифікація
4.3. Розробка вдосконаленого алгоритму знаходження точок Штейнера
4.4. Розробка експериментального зразка програми реалізації алгоритму
4.5 Дослідження характеристик розробленого підходу до розв'язання завдання
4.6 Висновки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):
5.1 Дерево Штейнера (2 шт).
5.2 Вдосконалений алгоритм знаходження точок Штейнера (2 шт).
5.3 Результати роботи алгоритму побудови дерева Штейнера (2 шт).

6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада консультанта	Завдання видав: (підпис консультанта, дата)	Завдання прийняв: (підпис студента, дата)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ	12.10.2023 – 13.10.2023	2%
2	Аналіз відомих алгоритмів побудови дерева Штейнера та їх класифікація	15.10.2023 – 25.10.2023	25%
3	Розробка вдосконаленого алгоритму знаходження точок Штейнера	26.10.2023 – 12.11.2023	30%
4	Розробка експериментального зразка програми реалізації алгоритму.	26.10.2023 – 12.11.2023	3%
5	Дослідження характеристик розробленого підходу до розв'язання завдання.	14.11.2023 – 23.11.2023	24%
6	Висновки	25.11.2023 – 30.11.2023	5%
7	Підготовка презентації та доповіді	31.11.2023 – 09.12.2023	9%
8	Подання кваліфікаційної роботи до кафедри	15.01.2024	2%
9	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії		

Студент _____ Олег ГЛУШКОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник роботи _____ Анатолій КОСОЛАПОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра:
76 с., 34 рис., 6 табл., 3 додатки, 25 джерел.

Об’єкт розробки – методи побудови мереж в розподілених комп'ютерних системах.

Предмет дослідження - розробка вдосконаленого алгоритму знаходження точок Штейнера.

Мета роботи – визначення оптимального способу організації мережі комунікації на підприємстві.

Методи дослідження – аналіз схем побудови мінімального остовного дерева Штейнера за допомогою алгоритмів та визначення способів для оптимізації та поліпшення організації мережі.

Визначено способ вдосконалення алгоритмів побудові дерева Штейнера.

Результати роботи можуть стати основою для вдосконалення не тільки існуючих комунікаційних мереж, а також в інших різноманітних сферах діяльності.

Перший розділ – описує існуючи алгоритми побудови дерева Штейнера та їх класифікація.

Другий розділ – описується вдосконалений алгоритм знаходження точок Штейнера .

Третій розділ – описує експериментальний зразок програми.

Четвертий розділ – дослідження характеристик алгоритма реалізації знаходження мінімального остовного дерева за допомогою генетичного алгоритма.

Ключові слова: ДЕРЕВО ШТЕЙНЕРА, МІНІМАЛЬНЕ ОСТОВНЕ ДЕРЕВО, ГЕНЕТИЧНИЙ АЛГОРИТМ, ВДОСКОНАЛЕННЯ МЕРЕЖІ

ЗМІСТ

ВСТУП

Актуальність роботи. Мережа комунікацій на будь якому підприємстві відіграє критичну роль у підтримці різних аспектів діяльності. Особливо це важливо при такому масштабному, як організації залізничного сполучення, де відстані між вузлами сягають сотень кілометрів. Мережа повинна ефективно зв'язуватися між різними відділами та співробітниками всередині компанії, а також із зовнішніми сторонами, такими як клієнти, постачальники, партнери, надавати платформи для спільної роботи, обміну файлами, відеоконференцій, що є важливим для підтримки продуктивності роботи відділов. Також забезпечує доступ до загальних баз даних, програмного забезпечення та інших цифрових ресурсів, необхідних для повсякденної роботи працівників.

Зважаючи на все перелічене, особливо великі відстані між центрами, оптимальне проектування мережі є дуже важливим. Існують різні способи з’єднання вузлів та центрів керування в єдину мережу. Одним з таких є проектування мережі за допомогою дерева Штейнера.

Мета та завдання дослідження. Метою даної кваліфікаційної роботи є дослідження алгоритмів знаходження дерева Штейнера для побудови мереж в розподілених комп'ютерних системах на основі знайдених бібліографічних посилань за допомогою розробленого програмного забезпечення.

Для досягнення поставленої мети у роботі необхідно виконати низку завдань:

- Аналіз відомих алгоритмів побудови дерева Штейнера та їх класифікація
- Розробка вдосконаленого алгоритму знаходження точок Штейнера.
- Розробка експериментального зразка програми реалізації алгоритму.
- Дослідження характеристик розробленого підходу до розв'язання завдання.

Об’єкт і предмет дослідження. Об’єктом дослідження є методи побудови мереж в розподілених комп'ютерних системах. Предметом виступає дослідження вдосконаленого алгоритму знаходження точок Штейнера.

Методи дослідження. Для вирішення поставлених задач було використано метод оптимізації існуючих алгоритмів пошуку точки Штейнера та застосування розробленого програмного забезпечення для оптимізації та поліпшення організації мережі.

Наукова новизна. Розроблено вдосконалений алгоритм знаходження точок Штейнера. Результати дослідження дозволяють зробити висновки щодо доцільності подальшої його оптимізації та впровадження.

Практична значимість. Практична значимість даної роботи полягає в тому, що результат досягнення поставленої мети може бути застосованим при проектуванні та оптимізації мереж в розподілених комп'ютерних системах.

Апробація результатів дослідження та публікації. Основні положення магістерської роботи доповідалися та були схвалені на кафедрі ЕОМ, а також було опубліковано тези доповіді в збірці XVII Міжнародної науково-практичної конференції «СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ ТА ОСВІТІ» від 13-14.12.2023 р.

1 АНАЛІЗ ВІДОМИХ АЛГОРИТМІВ ПОБУДОВИ ДЕРЕВА ШТЕЙНЕРА ТА ЇХ КЛАСИФІКАЦІЯ

У цьому розділі будуть розглянуті алгоритми, які застосовуються для побудови дерева Штейнера та що собою представляє ця задача. Особлива увага буде приділена аналізу їх ефективності, точності та практичності застосування. Також ми розглянемо питання NP-складності проблеми дерева Штейнера та обговоримо, як різні алгоритми справляються з цим викликом.

Цей аналіз включатиме в себе порівняння між різними методами, оцінку їх масштабованості та придатності до різних типів мережевих структур.

1.1 Задача Штейнера

Задача Штейнера (або задача дерева Штейнера) - це комбінаторна оптимізаційна задача в графах, яка полягає в пошуку найкоротшого можливого дерева (графа) для з'єднання заданого набору вершин, які називаються терміналами. Відрізняючись від більш традиційних мінімальних остовних дерев, дерево Штейнера може включати додаткові точки, які не є частиною первісного набору, але які допомагають скоротити загальну довжину дерева. Додаткові точки відомі як точки Штейнера.

Ця задача є комбінаторною оптимізаційною задачею NP-складності і може бути дуже складною для розв'язання, особливо на великих графах. Тому існують різні алгоритми, які намагаються наблизити оптимальне рішення або знайти розв'язок в прийнятний час.

Метод Штейнера має безліч застосувань у різних галузях, включаючи телекомунікації (для з'єднання мереж з мінімальною довжиною кабелів), маршрутизацію мереж, виробництво печатних плат, дизайн логістичних систем і багато інших областей.

Ось деякі з основних переваг:

- Скорочення витрат: Наприклад, у телекомунікаційних мережах дерево Штайнера дозволяє мінімізувати витрати на прокладання кабелів або встановлення точок бездротового доступу.

- Підвищення ефективності: За рахунок оптимізації маршрутизації трафіку або сигналів дерево Штейнер може підвищити ефективність мереж і систем. Наприклад, у транспортних мережах дерево Штайнера може оптимізувати маршрути транспортних засобів або мінімізувати час у дорозі між пунктами призначення.

- Підвищення надійності: Дерево Штейнер також може підвищити надійність систем за рахунок забезпечення резервування та резервних шляхів. Наприклад, в електромережах або водопровідних мережах дерево Штайнер може забезпечити альтернативні маршрути у разі збоїв або збоїв.

- Підвищення продуктивності: Дерево Штайнер може підвищити продуктивність систем за рахунок оптимізації розміщення компонентів або пристроїв. Наприклад, при проектуванні схем дерево Штейнер може оптимізувати розміщення компонентів на друкованій платі, щоб мінімізувати затримку сигналу і підвищити продуктивність схеми.

Хоча дерево Штейнер має ряд переваг, воно також має деякі недоліки, які слід взяти до уваги:

- NP-Труднощі: Відомо, що проблема дерева Штейнера є NP-складна, а це означає, що знайти точне рішення для великих випадків завдання неможливо обчислити.

- Алгоритми апроксимації: Хоча існує кілька алгоритмів апроксимації завдання дерева Штейнера, вони не завжди можуть забезпечити оптимальні рішення. У деяких випадках якість рішення може бути значно гіршою за оптимальне рішення, що може вплинути на продуктивність і надійність системи.

- Чутливість до змін рішення «Дерево Штейнера» може бути дуже чутливим до змін вхідних параметрів, таких як розташування терміналів або вартість ребер. Невелика зміна вхідних параметрів може призвести до значної зміни рішення дерева Штейнера, що може ускладнити порівняння різних рішень або оптимізацію системи.

- Обмежена застосовність: Дерево Штейнера в основному застосовується до завдань, пов'язаних зі з'єднанням набору терміналів у графі. Він може не підійти для інших типів оптимізації завдань, таких як планування або розподіл ресурсів [17].

1.2 Геометричне визначення точки Штейнера

Найпростіший приклад задачи Штейнера коли кількість точок дорівнює трьом. Рішення задачі Штейнера для трьох точок було знайдене ще в XVII столітті. П. Ферма, Є. Торричеллі та Б. Кавальєрі в свій час довели, що дерево Штейнера для трикутнику будується за допомогою однієї додаткової точки (при умові що всі кути трикутника менші 120°), котра називається точкою Ферма-Торричелли-Штейнера. Якщо один з кутів трикутника більше або дорівнює 120°, то дерево Штейнера складається зі сторін цього кута [1].

На Рис. 1 показан процес побудови точки Штейнера S для трьох точок A, B, C. Для пошуку точки Штейнера треба на будь-якій стороні начального трикутника добудувати назовні рівносторонній трикутник A’BC. Точка S перетин кола, описаного навколо трикутника A’BC, та відрізка AA’ буде являтися точкою Штейнера. Також цю точку можно знайти, якщо добудувати рівносторонні трикутники к будь-яким іншим сторонам.

Рішення задачі Штейнера для трьох точок є основою для побудови найкоротшої сітки Штейнера для великої кількості точок. Відомо, що для будь-якої кінцевой системи точок на площини існує кінцева множина сіток

Штейнера. Для находження найкоротшого шляху серед множини точок треба знайти всі сітки Штейнера, а потім вибрати найкоротшу. Однак точне рішення задачі потребує розгляду великої кількості варіантів, так що навіть найкращі алгоритми, що виконуються на самих бистродіючих комп'ютерах, не в змозі надати рішення для великої множини заданих точок за реально прийнятний час. Більше того, задача Штейнера належить до класу задач, для яких, на думку багатьох сучасних дослідників, ефективні алгоритми ніколи не будуть знайдені. З огляду на затребуваність задачі для практичних додатків, актуальною є побудова нових алгоритмів, у тому числі дають наближене рішення задачі Штейнера.

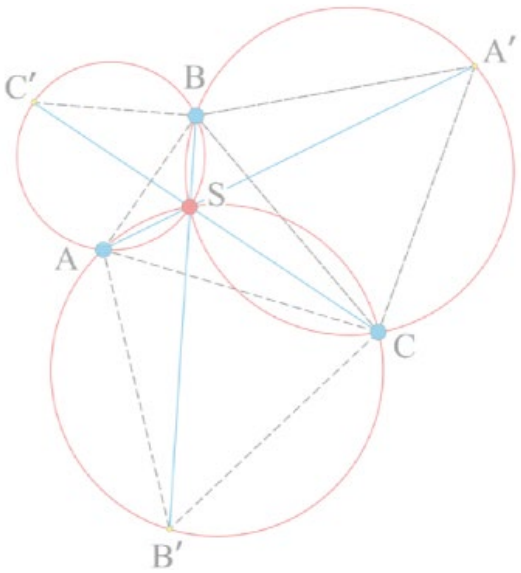

Рис. 1.1 Побудова точки Штейнера S для трикутника ABC геометричним методом[1]

Завдання Штейнера для трьох точок також дає деяку інформацію про геометрію найкоротших дерев Штейнера. По-перше, кожен кут дорівнює 120° або більше, а це означає, що кожна точка з'єднується з рештою дерева не більше ніж трьома ребрами. По-друге, у кожній точці Штейнера сходяться рівно три ребра, утворюючи один з одним кути, точно рівні 120°. По-третє, число ребер дерева завжди на одиницю менше сумарного числа заданих

вихідних точок та точок Штейнера. І нарешті, остання властивість: оскільки в кожній точці Штейнера сходяться рівно три ребра і принаймні одне ребро має стосуватися кожної із заданої множини точок, максимальна кількість точок Штейнера для будь-якої задачі на дві менше, ніж число заданих вихідних точок.

На Рис. 2 показано приклад пошуку сітки для 3, 4 та 6 точок, розташованих у вершинах рівностороннього трикутника, прямокутника та «сходів» і має різні рішення. У випадку трьох точок трикутник рівнобедрений, де всі кути 120°.Задача пошуку за точками, що розташовані у вершинах рівностороннього трикутника має різні рішення. У випадках, a, d та g точки з'єднуються без допоміжних точок і це рішення називається мінімальне основне дерево (МОД). Дерева Штейнера, отримані шляхом додавання вузлових точок, випадки представлені: b, c, e, f, h та i. Єдиним прикладом найкоротшого дерева Штейнера є приклади c, f і i. Числа під кожним рішенням вказують зразкову сумарну довжину відрізків мережі.

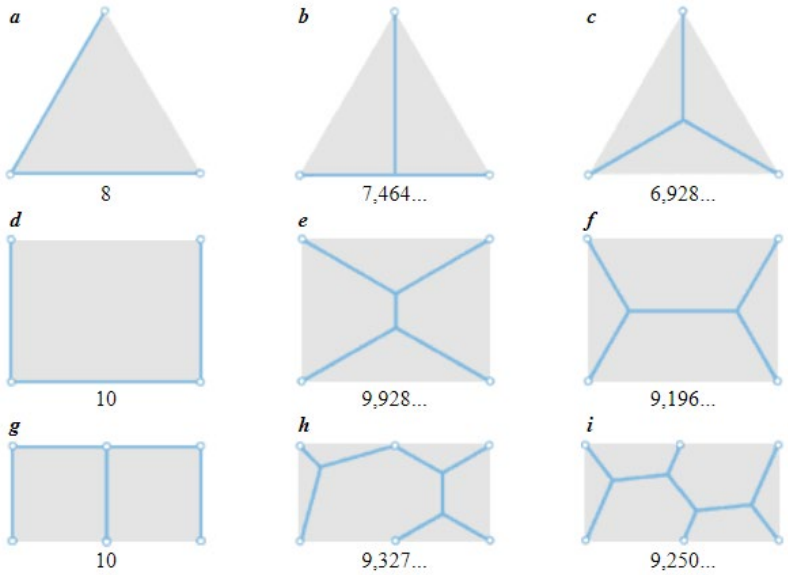

Рис. 1.2 Рішення для точок, розташованих у вершинах рівностороннього трикутника, прямокутника та «сходів» [1]

При однаковій кількості і розташування вихідних точок можна побудувати багато різних дерев Штейнера, які задовольняють переліченим

вище умовам. Деякі з цих дерев, які називають локально мінімальними рішеннями, неможливо скоротити за рахунок дрібномасштабних змін, таких як невелике переміщення ребра або розщеплення точки Штейнера. Однак не всяке локально мінімальне дерево Штейнера дає найкоротше з можливих рішень завдання. Для того, щоб перетворити мережу на найкоротше дерево, зване глобально мінімальним деревом Штейнера, можуть знадобитися великомасштабні переміщення точок Штейнера[1].

Стислі характеристики алгоритмів представлені в таблиці 1.[18, 19, 20]

Таблиця 1

Тип алгоритму	Опис	Приклад
1. Жадібни алгоритми	Використовують прості жадібні стратегії для побудови дерева Штейнера, додаючи краї за критерієм мінімальної ваги або іншими локально оптимальними рішеннями.	Алгоритм Крускала та Пріма, адаптовані для дерева Штейнера [23].
2. Евристичні алгоритми	Використовують евристичні методи для приблизного розв'язку проблеми дерева Штейнера, забезпечуючи достатньо хороші рішення за менший час.	Алгоритм Штейнера в графах, Евристика Штейнера.
3. Метаевристичні	Використовують більш загальні стратегії пошуку, такі як симульований відпал або алгоритми рою частинок, для знаходження оптимальних або майже оптимальних рішень.	Генетичні алгоритми, Алгоритми рою частинок для дерева Штейнера, Адаптивні алгоритми.
4. Точні алгоритми	Розв'язують проблему дерева Штейнера точно, але можуть бути неефективними для великих графів через високу обчислювальну складність.	Цілочисельне лінійне програмування, Алгоритми повного перебору, Мелзака. Ханана.
5. Алгоритми апроксимації	Надають гарантовані межі точності для рішень, які вони генерують, часто швидше, ніж точні алгоритми.	Апроксимаційні алгоритми для дерева Штейнера.

Продовження Таблиці 1

Тип алгоритму	Особливості	Переваги	Недоліки
1. Жадібни алгоритми	Використовують прості жадібні стратегії для побудови дерева, додаючи краї за мінімальною вагою.	Простота реалізації; Швидкість виконання.	Можуть не знаходити оптимальне рішення; Обмежена ефективність[24].
2. Евристичні алгоритми	Використовують приблизні методи для знаходження достатньо хороших рішень.	Ефективність для великих графів; Хороший баланс між швидкістю та якістю.	Не гарантують оптимального рішення; Можуть бути складні у налагодженні[22].
3. Метаевристичні	Використовують загальні стратегії пошуку, такі як генетичні алгоритми або алгоритми рою частинок.	Гнучкість; Потенційно висока якість рішень.	Висока обчислювальна складність; Потребують тонкого налагодження параметрів[16].
4. Точні алгоритми	Розв'язують проблему точно, часто за допомогою методів цілочисельного лінійного програмування.	Гарантія знаходження оптимального рішення.	Висока обчислювальна складність; Неефективні для великих графів[1.4].
5. Алгоритми апроксимації	Надають гарантовані межі точності для рішень, які вони генерують.	Ефективність для великих графів; Гарантовані межі точності.	Не завжди знаходять абсолютно оптимальне рішення[13].

Вибір конкретного алгоритму залежить від специфіки задачі та вимог до точності та швидкості виконання.

1.3 Алгоритм Мелзака

Алгоритм Мелзака відноситься до типу екзактних або точних алгоритмів для розв'язання задачі Штейнера. Цей алгоритм забезпечує точне рішення задачі, але його ефективність обмежена великими обсягами даних або високою складністю задачі через високий обчислювальний час, особливо в випадках, коли задача є NP-повною. Тому алгоритм Мелзака частіше використовується для менших або більш обмежених наборів даних, де його висока точність може бути вигідною без значного збільшення часу обчислень.

Діючи методом повного перебору, можна знайти найкоротшу мережу шляхом побудови всіх можливих локально мінімальних дерев Штейнера, обчисленням їхньої довжини та вибором найкоротшого. Але оскільки розташування точок Штейнера є неоднозначним, виникає сумнів у тому, що обчислити всі локально мінімальні дерева Штейнера можна за кінцевий час. З. Мелзак з Університету Британської Колумбії зумів подолати цю перешкоду і склав перший алгоритм для вирішення завдання Штейнера.

В алгоритмі Мелзака розглядаються багато можливих з'єднань між заданими точками і багато можливих розташування точок Штейнера. Алгоритм можна умовно розбити дві частини. У першій його частині безліч вихідних точок просто поділяється на всілякі підмножини. У другій частині для кожного такого підмножини створюється ряд можливих дерев Штейнера з використанням побудови, аналогічної до того, яке ми застосували до завдання з трьома точками.

Так само як і для трьох точок, замість двох вихідних точок можна підставити одну точку, що їх замінює, не змінюючи результату (довжини мережі) рішення. Однак у випадку алгоритм повинен вгадати, яку пару слід замінити, і тому він перебирає всі можливі пари. Більш того, точка, що

замінює, може розміщуватися по будь-який бік від прямої, що з'єднує дві замінні точки, оскільки рівносторонній трикутник, що використовується при побудові, може бути орієнтований в одному з двох напрямків. Після того як одна з точок в підмножині замінена однією з двох можливих точок, що замінюють, на кожному наступному кроці алгоритму заміщаються або дві інші вихідні точки, або одна вихідна і одна заміщаюча, або дві заміщають іншою заміщувальною точкою; і так до тих пір, поки все підмножина не буде зведено до трьох точок. Як тільки для цих трьох точок знайдено точку Штейнера, алгоритм починає працювати у зворотному напрямку, намагаючись визначити точку Штейнера, відповідну кожній точці, що заміщає (див. рис. 3).

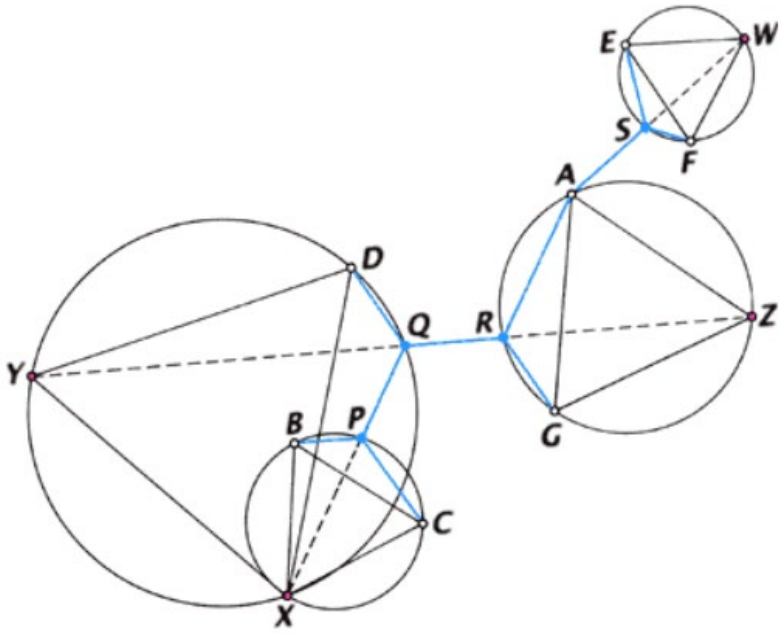

Рис. 1.3 Схема алгоритму Мелзака [2]

Алгоритм Мелзака розбиває завдання пошуку найкоротшої мережі на підзавдання. Точка A підходить для розбиття завдання на підзавдання з 3 та 5 точок. Щоб побудувати можливі дерева Штейнера для 5 точок, пару точок (наприклад, B і C) можна замінити на одну (тут X), побудувавши рівносторонній трикутник з основою BC. Тепер завдання зведено до 4 точок:

X, D, G і A. Пару точок знову можна замінити - спочатку D і X на Y, а потім G і A на Z. Навколо кожного з отриманих рівносторонніх трикутників (XDY, AGZ та BCX) описуємо кола. Точки Q і R, у яких пряма YZ перетинає два кола, - це точки Штейнера, а перетин прямий XQ з третім колом визначає точку Штейнера P. Оскільки неможливо заздалегідь передбачити найкраще розбиття на підзавдання та угруповання на пари, необхідно розглянути всі варіанти, щоб знайти найкоротше дерево.

Спроба може закінчитися невдачею, оскільки на розташування точок Штейнера накладаються обмеження, що суперечать один одному. Однак успішна спроба призводить до виникнення дерева Штейнера, що з'єднує кожну вихідну точку підмножини з одним ребром деревом. Розглянувши, таким чином, всі послідовності, що заміщають, алгоритм вибирає найкоротше з цих дерев Штейнера для підмножини. Комбінуючи між собою усілякими способами найкоротші дерева Штейнера для підмножин так, щоб охопити вихідну множину точок, можна побудувати всілякі локально мінімальні дерева Штейнера і визначити геометрію найкоротшої мережі.

Алгоритм Мелзака може вимагати колосального часу навіть для невеликих завдань, оскільки в ньому розглядається дуже багато варіантів. Наприклад, завдання для 10 точок може бути розподілена на 512 підмножин вихідних точок. І хоча двоточкові підмножини не вимагають великого обсягу роботи, кожна з 45 підмножин з вісьмома точками має два мільйони послідовностей, що заміщають. Крім того, існують ще більше 18 000 способів поєднати ці підмножини в дерева [2].

1.4 Алгоритм Ханана

Алгоритм Ханана також відноситься до типу точних або екзактних алгоритмів для розв'язання задачі Штейнера. Він використовує ґраткову структуру для визначення можливих місць розташування точок Штейнера та

знаходження мінімального дерева Штейнера. Цей алгоритм, хоча і забезпечує точне рішення, має високу обчислювальну складність, особливо при великій кількості термінальних точок, що робить його менш практичним для великих задач.

Можливо, найважливішим практичним застосуванням завдання Штейнер є конструювання інтегральних електронних схем. Коротша мережа провідних ліній на інтегральній схемі вимагає меншого часу зарядки-розрядки в порівнянні з більш довгою мережею і підвищує таким чином швидкодія схеми. Однак завдання відшукання найкоротшої мережі на інтегральній схемі має іншу геометрію, оскільки провідники на ній зазвичай проходять лише у двох напрямках – горизонтальному та вертикальному.

Різновиди задачі про найкоротшу мережу застосовувалися при конструюванні електронних інтегральних схем, щоб підвищити їхню швидкодію. Найкоротша мережа з вертикальних та горизонтальних провідників, що пов'язують безліч висновків, виділена червоним кольором. Тут показані також провідники та висновки у глибших шарах схеми (Рис. 6).

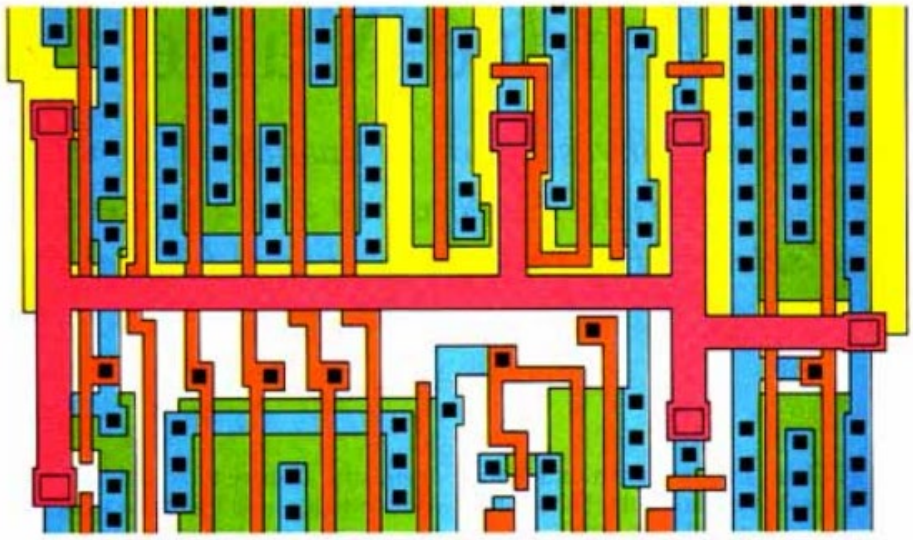

Рис. 1.4 Електронна інтегральна схема[4].

Така задача, яка отримала назву прямокутної задачі Штейнера, було вперше вивчено в 1965 році Морісом Хенаном з Дослідницького центру ім.

Томаса Вотсона корпорації IBM у Йорктаун-Хейтсі (шт. Нью-Йорк). Як і в класичній задачі Штейнера, рішення для прямокутної її версії також містить точки Штейнера і вихідні точки, але ребра зустрічаються в них під кутом або 90 °, або 180 °. Хоча точки Штейнера можуть, здавалося б, лежати повсюдно у прямокутній задачі, так само як і в класичній задачі Штейнера, Хенан показав, що в найкоротшій прямокутній мережі на розташування точок Штейнера можна накласти певні обмеження. Через кожну вихідну точку проводяться горизонтальна і вертикальна прямі, і кожне перетин двох ліній дає можливе положення точки Штейнера. Щоб знайти найкоротшу мережу, алгоритм може розглянути всі підмножини можливих точок Штейнера. Однак у міру того, як кількість вихідних точок зростає, час розв'язання для кожного алгоритму, що здійснює повний перебір варіантів, зростає експоненційно. Більш тонкі, але все ж таки експоненційні алгоритми здатні вирішувати прямокутні завдання Штейнера розміром близько 40 точок.

Прямокутна версія задачі пошуку мінімального кістяка може бути ефективно вирішена алгоритмом, що вибирає на кожному кроці найкоротше з'єднання, якщо це з'єднання не утворює замкнутого шляху. Ф. Хванг з фірми Bell Laboratories показав, що прямокутне дерево Штейнера не буває коротше прямокутного остовного дерева більш ніж на одну третину[4].

1.5 Адаптивний алгоритм

Адаптивний алгоритм - це метаевристичний тип алгоритму, який здатен змінювати свою поведінку або стратегію вирішення задачі в залежності від вхідних даних або умов середовища, в якому він працює. Ця здатність алгоритму "адаптуватися" дозволяє йому бути більш гнучким та ефективним у вирішенні проблем, особливо в динамічних або складних умовах.

Основні характеристики адаптивних алгоритмів:

- Гнучкість: Адаптивні алгоритми можуть змінювати свою стратегію в залежності від поточної ситуації або вхідних даних.

- Оптимізація: Часто вони спрямовані на знаходження найкращого можливого рішення в мінливих умовах.

- Навчання з досвіду: Деякі адаптивні алгоритми використовують методи машинного навчання або штучного інтелекту для вдосконалення своїх стратегій на основі попереднього досвіду.

- Реагування на зміни: Адаптивні алгоритми добре працюють у середовищах, де умови часто змінюються або непередбачувані.

Адаптивні алгоритми знаходять застосування в різних областях, включаючи оптимізацію мереж, обробку сигналів, робототехніку, економічне моделювання, і навіть у таких складних задачах, як управління штучним інтелектом або машинне навчання.

Адаптивний алгоритм побудови дерева Штейнера основан на побудові МОД Пріма та його ортогоналізації з використанням решітки Ханана. Для вибору ортогональної реалізації ребер Прима дерева використовується генетичний алгоритм. У розробленому адаптивному алгоритмі побудови дерева Штейнера використовується параметрична адаптація, яка полягає у виборі значення параметра адаптації на основі аналізу зовнішніх умов вирішення задачі та інформації, що зберігається в базі даних. Як зовнішні умови виконання завдання побудови дерева Штейнера запропоновано використовувати: розмірність задачі; ресурс часу, відведений рішення завдання, і продуктивність комп'ютера, у якому вирішується завдання. Як параметр адаптації запропоновано використовувати кількість ітерацій в генетичному алгоритмі при застосуванні операцій кросинговера над варіантами реалізації ребра дерева Пріма.

Розглянутий алгоритм побудови дерева Штейнера відноситься до класу адаптивних алгоритмів з керованою точністю рішення та використанням параметричної адаптації до зовнішніх умов його виконання: розмірності

завдання, ресурсу часу, відведеного для виконання проектної операції, а також швидкодії комп'ютера, на якому виконується програма алгоритму. Він спрямований на отримання найточнішого рішення за відведений час на основі вибору значення параметра адаптації[9].

1.6 Жадібни алгоритми

Жадібні алгоритми (Greedy algorithms) — це клас алгоритмів оптимізації, які, просто кажучи, вибирають локально оптимальне рішення на кожному кроці з надією знайти глобально оптимальне рішення. Для задачі побудови дерева Штейнера це означає вибір шляхів або вузлів, які здаються найбільш вигідними в даному контексті, не беручи до уваги всю картину в цілому. Жадібні алгоритми (ЖА) в контексті побудови дерева Штейнера часто використовують підходи, які є концептуально схожими на ті, що використовуються в алгоритмах Пріма та Крускала.

Жадібні алгоритми для побудови дерева Штейнера зазвичай працюють за наступним принципом. Ініціалізація: Починають з множини термінальних точок, що потрібно з'єднати. На кожному кроці алгоритм вибирає край, вузол або піддерево, яке виглядає найбільш вигідним (наприклад, має найменшу вагу або довжину).Оцінюється, чи цей вибір приближає до рішення задачі (наприклад, зменшення загальної довжини дерева). Ітерація: Процес повторюється, доки не буде знайдено рішення або поки не будуть розглянуті всі можливості. Жадібні алгоритми не переглядають або не скасовують раніше прийняті рішення, навіть якщо це могло б покращити кінцевий результат. Це означає, що вони не використовують техніку "backtracking". Визначення жадібного критерію є ключовим для успішної роботи алгоритму. Він визначає, як робити вибір на кожному кроці, та є фундаментальним для стратегії алгоритму.

Хоча ці алгоритми можуть бути ефективними для деяких задач, вони не завжди забезпечують глобально оптимальне рішення. Для деяких задач (зокрема, NP-складних, до яких належить задача побудови дерева Штейнера), знайти глобально оптимальне рішення може бути вкрай складно або навіть неможливо за розумний час [24].

1.7 Методи усічення

Методи усічення (pruning methods) можуть бути використані в різних видах алгоритмів, зокрема в жадібних алгоритмах та в інших евристичних підходах. Ці методи використовуються для скорочення простору пошуку. Коли алгоритм розглядає різні можливі рішення, він може "усікати" ті гілки пошуку, які вже не можуть призвести до оптимального рішення. Це робиться на основі певних критеріїв, наприклад, якщо поточний найкращий варіант вже кращий, ніж максимально можливий результат в даній гілці. Наприклад, можуть ігноруватися ті краї або точки Штейнера, які значно збільшують загальну довжину або вартість дерева.

Пошук найбільш ефективни шляхів організації обчислень дозволили підвищити швидкодію алгоритму. Замість розглядання геометрію завдання, увагу фокусують на можливих конфігураціях з'єднань у мережі, тобто, її топології. Топологія вказує, які точки з'єднані одна з одною, а чи не дійсні розташування точок Штейнера. Прийнявши певну топологію, можна знайти відповідний ланцюжок, що заміщає, відносно швидко. За такої організації процесу швидкість обчислення найкоротших дерев Штейнера для підмножин трохи зростає. Наприклад, для підмножини з 8 точок алгоритм має розглянути лише близько 10 000 різних топологій замість двох мільйонів різних послідовностей заміщення.

Так як кількість можливих топологій швидко зростає з розміром підмножини, завдання Штейнера можуть стати менш трудомісткими лише в

тому випадку, якщо потрібно розглядати дуже невеликі підмножини вихідної множини точок. Експерименти, проведені з алгоритмом Мелзака, показали, що найкоротша мережа числа випадкових точок більше 6 зазвичай може бути розбита на найкоротші мережі для менших наборів точок. Проте, розглянувши спеціальні зміни точок, звані сходами, Ф. Чанг з фірми Bell Communications Research та Рональдом Л. Гремом показав, що є нескінченно великі безлічі точок, котрим найкоротше дерево Штейнера неможливо розчленувати. Сходи - це конфігурація, в якій вихідні точки розташовані рівномірно вздовж двох паралельних ліній. Для цього дуже приватного завдання Штейнера було знайдено загальне рішення. Воно показало, що число точок Штейнера в найкоротшому дереві Штейнера для сходів з непарною кількістю «сходинок» максимально: воно дорівнює кількості вихідних точок мінус 2. Таке дерево Штейнера неможливо розчленувати, тому що для кожної точки Штейнера потрібно одночасно враховувати всі вихідні точки. Отже, який завжди можна скоротити розмір підмножин, аналізованих алгоритмом Мелзака. Деяким дослідникам вдалося покращити ефективність алгоритмів у порівнянні з алгоритмом Мелзака за рахунок застосування більш тонких способів, що дозволяють зменшити обсяг обчислень.

Методи усічення підвищують ефективність алгоритмів пошуку найкоротших мереж. Один із прийомів усічення, або виключення, можливих мереж (винайдений Кокейном) полягає в тому, щоб розглянути порядок, в якому гумове кільце (червоний колір), натягнуте навколо заданої множини точок, стосується їх. Гумка стосується всіх точок, за винятком C і H, але C можна включити в послідовність, оскільки кут, що утворюється точкою C з двома сусідніми точками, що знаходяться в контакті з гумкою, не менше 120°. Тоді порядок точок буде ABCDEFG. Безперервний шлях (чорний колір), що проходить уздовж можливої мережі (синій колір), стосується точок у порядку ACBDEFHG. Оскільки B і C тут переставлені місцями

стосовно послідовності, утвореної гумкою, цю мережу можна виключити з розгляду (Рис. 4).

В цих алгоритмах виробляється усічення обчислювальної процедури, тобто, припиняються ті гілки обчислення, які свідомо мають призвести до порівняно довгих мереж. Нові методи усічення справді значно скорочують обсяг обчислень. Програми, засновані на алгоритмі Мелзака, могли вирішити будь-яке завдання 9 точок і деякі завдання 12 точок приблизно півгодини. Програма, написана Кокейном та його колегою з Університету Вікторії Д. Хьюджіллом, використовує потужний метод усічення, винайдений Р. Вінтером з Копенгагенського університету. Ця програма спромоглася вирішити всі завдання для 17 точок і більшість випадково згенерованих задач для 30 точок всього за кілька хвилин. Метод усічення Вінтера виявився настільки вдалим, що завдяки усуненню більшості можливих топологій, основний обсяг обчислювальної роботи пов'язаний із комбінуванням рішень, отриманих для окремих підмножин[4].

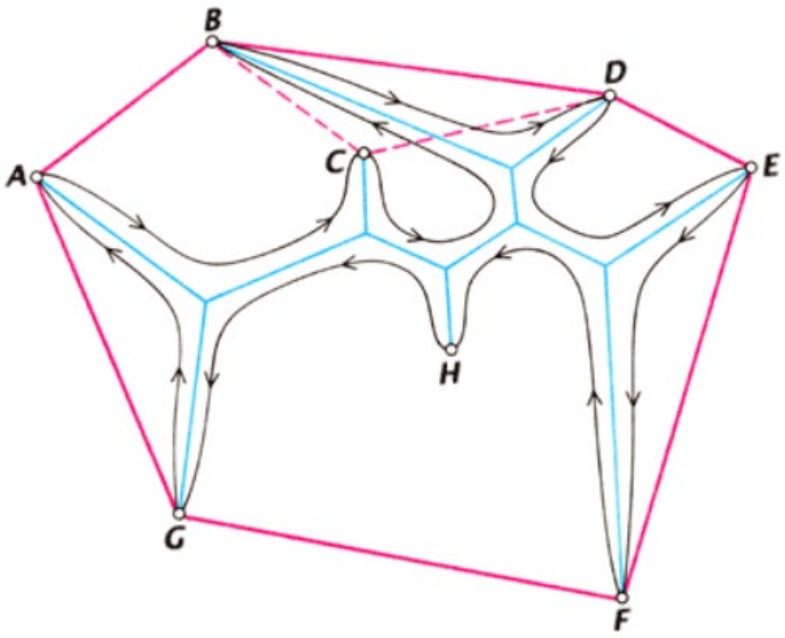

Рис. 1.5 Метод усічення [4]

Висновки до першого розділу

1. NP-складність задачі: Задача побудови дерева Штейнера є NP-складною, що означає, що на сьогоднішній день не існує відомих алгоритмів, які можуть вирішити цю задачу в поліноміальний час для великих вхідних даних. NP-складність вказує на те, що перевірка правильності рішення може бути здійснена набагато швидше, ніж його знаходження.

2. Складність знаходження оптимального рішення: Однією з основних причин, чому задача побудови дерева Штейнера є вкрай важкою, є необхідність визначення оптимальних позицій для точок Штейнера, які не задані заздалегідь і можуть бути будь-де у просторі рішень. Це створює величезну кількість потенційних конфігурацій, які необхідно розглянути.

3. Жадібні та евристичні методи як альтернативи: У зв'язку з NP-складністю задачі, алгоритми, які гарантують знаходження абсолютно оптимального рішення, зазвичай є непрактичними для великих графів. Тому на практиці часто використовуються жадібні алгоритми та різноманітні евристичні підходи, які забезпечують відносно ефективне знаходження рішень, що є "достатньо хорошими", хоча й не обов'язково оптимальними.

4. Важливість дослідження у цій області: Попри складності та обмеження, дослідження алгоритмів для побудови дерева Штейнера має велике значення для теоретичної інформатики та багатьох практичних застосувань. Оптимізація мережевих структур, зокрема в телекомунікаціях та транспортних мережах, є ключовою для підвищення ефективності та зниження витрат.

Цей розділ демонструє глибину та складність задачі побудови дерева Штейнера, а також підкреслює важливість подальших досліджень для розробки більш ефективних алгоритмів, що можуть знайти застосування у різних областях.

2 РОЗРОБКА ВДОСКОНАЛЕНОГО АЛГОРИТМУ ЗНАХОДЖЕННЯ ТОЧОК ШТЕЙНЕРА

Попри значну актуальність та важливість побудови дерева Штейнера, варто зазначити, що ця задача належить до складних проблем. Це означає, що не існує відомого алгоритму, який здатен знайти точне оптимальне рішення для довільного вхідного набору даних у поліноміальний час. В результаті, більшість існуючих алгоритмів зосереджуються на наближених або евристичних рішеннях, які можуть не гарантувати оптимальності, але забезпечують практично прийнятні результати у розумний час.

Однак, навіть ці методи часто стикаються з обмеженнями у вигляді високих обчислювальних витрат та недостатньої точності, особливо при роботі з великими або складними графами. Таким чином, розробка альтернативних алгоритмів, які здатні ефективніше вирішувати цю задачу, є критично важливою для подальшого прогресу в областях, де використовується дерево Штейнера.

Одним з таких методів може бути метод кластеризації точок на площині. Основна ідея методу полягає у використанні специфічного підходу до визначення точок Штейнера, що оптимізують під'єднання заданих точок у просторі. Метою цього підходу є зменшення обчислювальної складності, зберігаючи при цьому адекватну точність у побудові дерева Штейнера. Через спрощенний підхід до кластеризації та визначення точок Штейнера, цей метод має потенціал значно оптимізувати процес вирішення поставленої задачи, особливо в сценаріях з великою кількістю точок. Для визначення його ефективності проведемо дослідження на графах з чотирма, п’яти та шести вершинами розташованими на площині випадковим чином.

2.1 Для чотирьох точок

Вибір Точок: Візьмемо чотири точки, розміщених випадково на площині з такими координатами (Рис. 2.1):

A (2, 3); B(4, 9); C(13, 7); D(10, 1)

Довжина ребер рахується за формулою :

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (4.1)$$

AB=6,32; BC= 9.22; CD=6,71; DA=8,25.

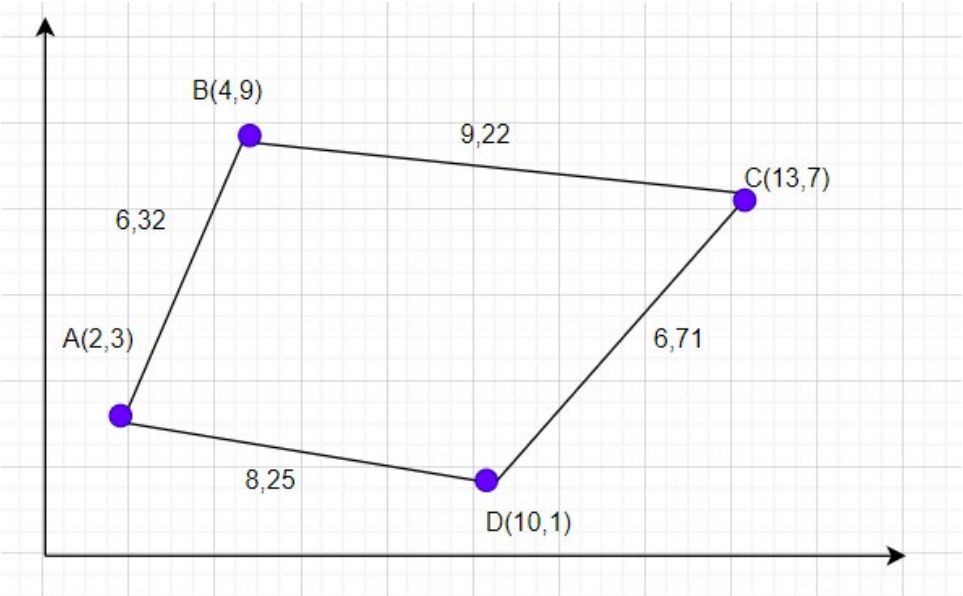

Рис. 2.1 Початкові точкі графа

Кластеризація: Спочатку ми поділимо площу на 2 трикутника (Рис.2.2). Далі знаходимо точки для кожного з цих трикутників за допомогою программи Steiner Point Calculator. Опис програми приведен в третьому розділі цієї роботи.

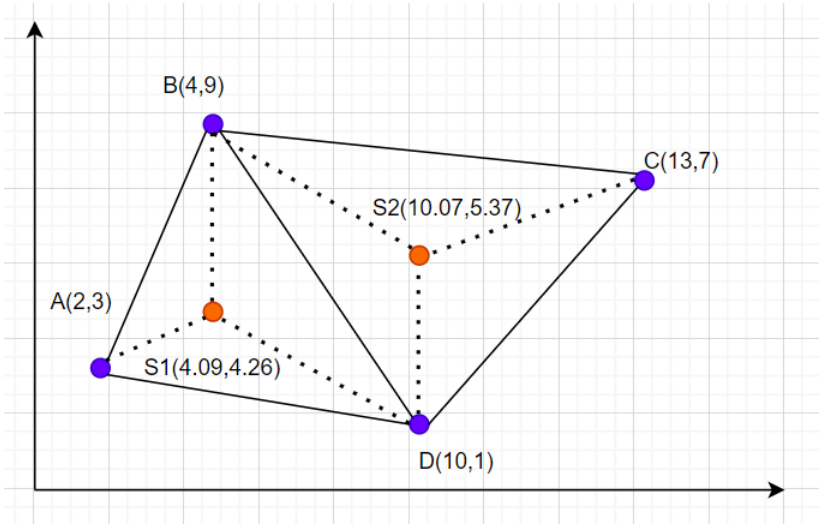

Рис. 2.2 Розділення графа на кластери

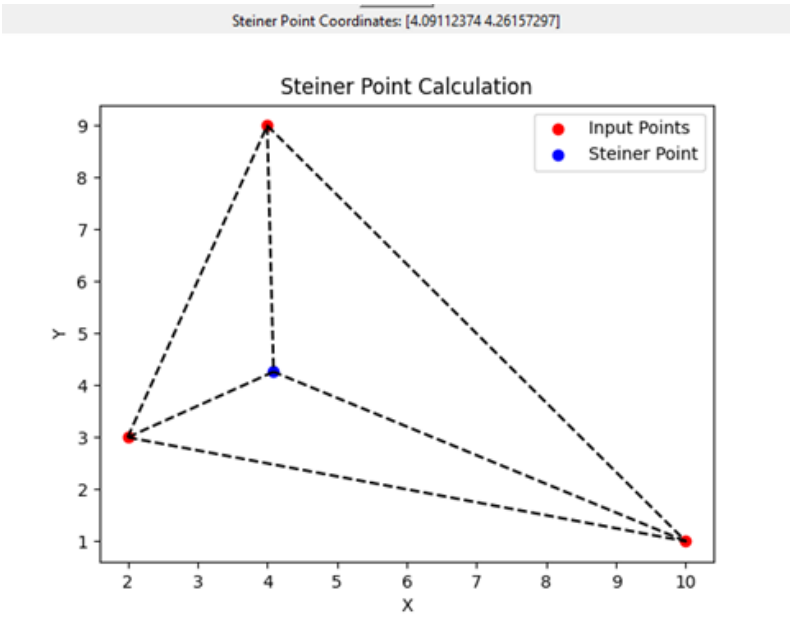

Рис. 2.3 Знаходження ТШ для трикутника

Координати нових точек: S1 (4.09, 4.26); S2(10.07, 5.37); Об’єднаємо точки в дерево Штейнера: по дві найближчи до отриманих та з’єднаємо їх.

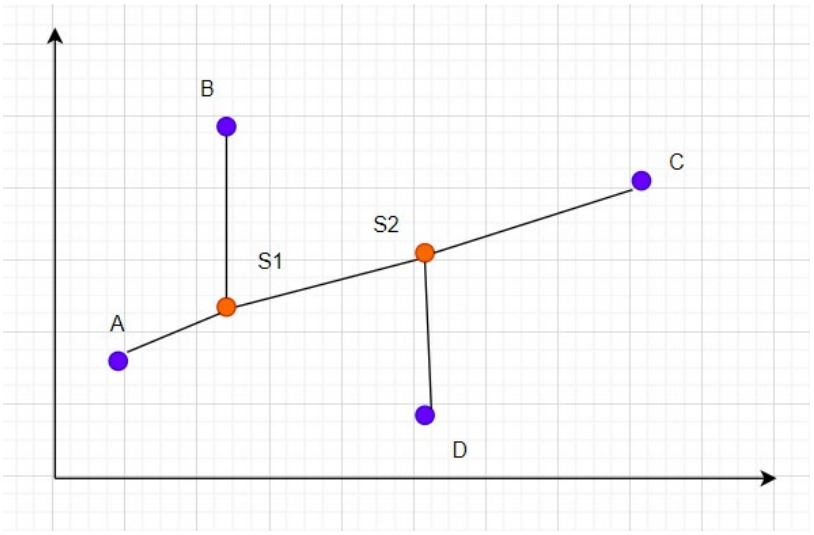

Рис. 2.4 ДШ для 4-х точок прямим з’єднанням

Для порівняння знайдемо сумму всіх ребер в отриманом дереві Штейнера та мінімальним остовним деревом для четирьох початкових точок.

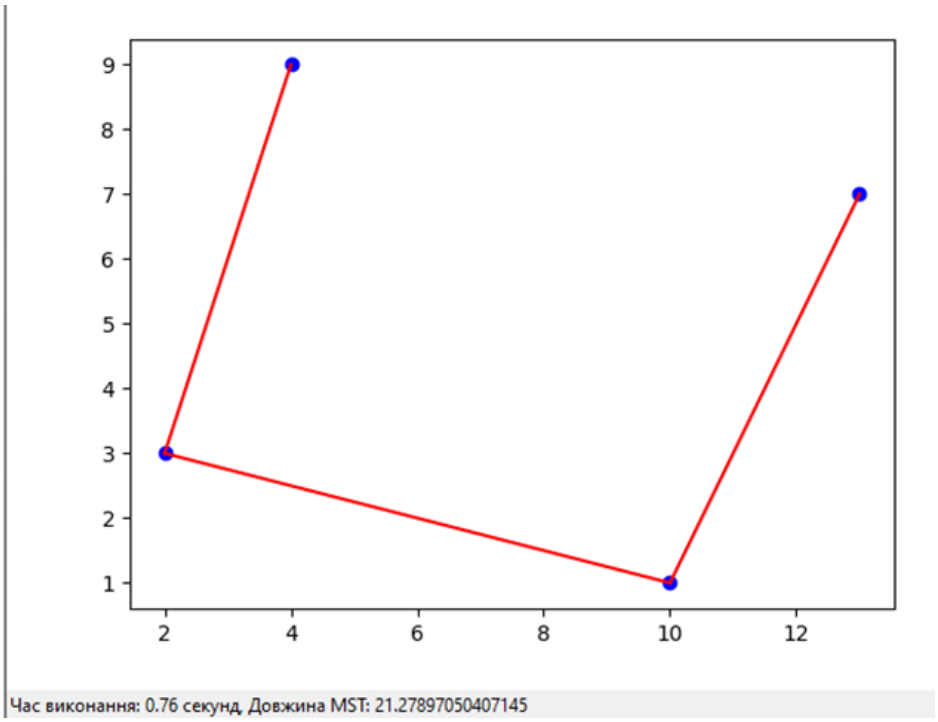

Рис. 2.5 МОД для 4 точок за допомогою EMST

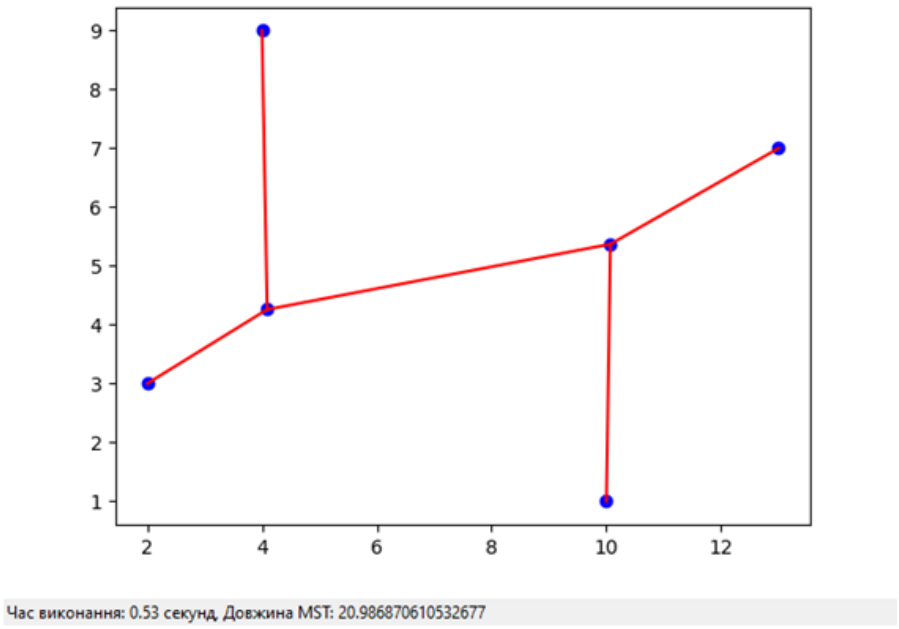

Рис. 2.6 ДШ для 4-х точок за допомогою EMST

Довжина МОД 21.28

Довжина ребер дерева Штейнера $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

A–S1= 2,44; B–S1=4,74; C–S2=3,35; D–S2=4,37; S1–S2=6,08;

Загальна сума довжин ребер після з’єднання на графі: 20,98
Загальна сума довжин ребер ДШ за допомогою EMST: 20,98
Загальна сума довжин ребер МОД за допомогою EMST: 21,28

2.2 Для п’яти точок

Візьмемо п’ять точок, розміщених випадково на площині з такими координатами (Рис. 2.7):

A (1, 3); B(4, 8); C(9, 8); D(10, 3); E(6, 1)

Довжина ребер: AB=5,83; BC=5,00; CD=5,10; DE=4,47; EA=5,39.

Кластерізація: Спочатку ми поділимо площу на три трикутника (Рис. 2.8).

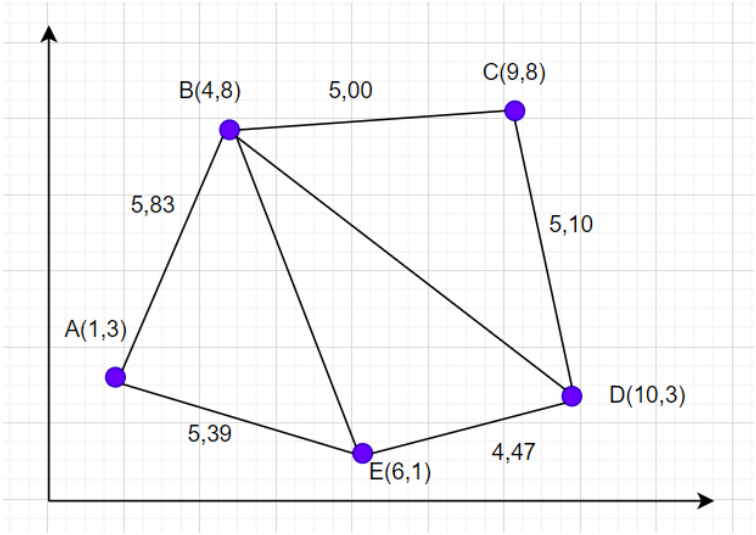

Рис. 2.7 Початкові точкі графа з п’ятьох вершин

Знаходження Точок Штейнера для Кластерів: Для кожного з цих кластерів з трьох точок ми визначимо відповідну точку Штейнера за допомогою программи Steiner Point Calculator.

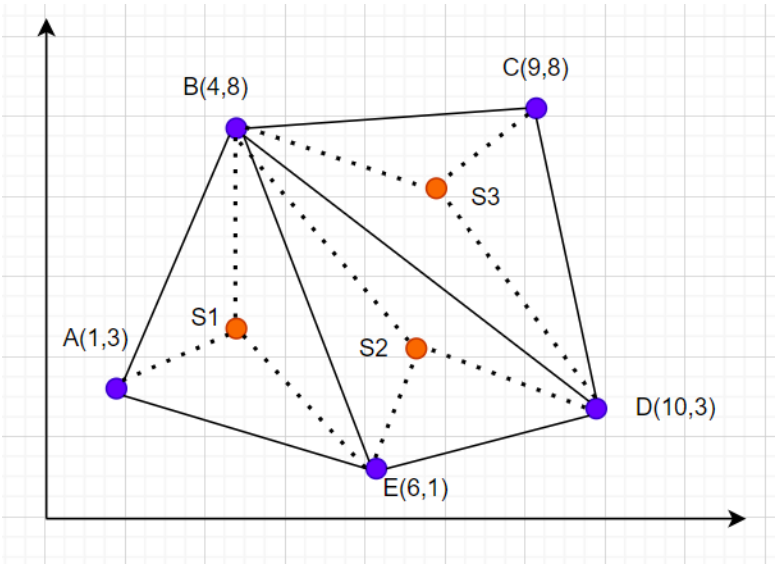

Рис. 2.8 Розділення графа з п’ятью точками на кластери

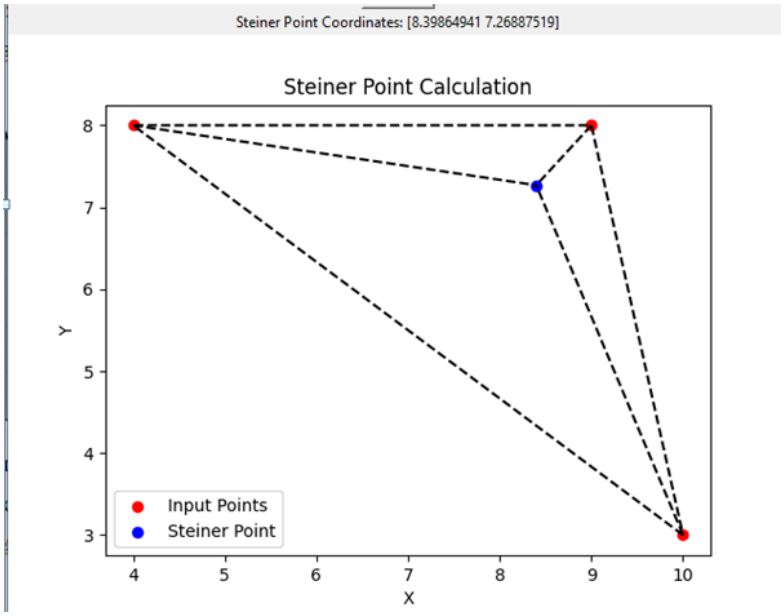

Рис. 2.9 Знаходження ТШ для трикутника

S1 (3.06, 3.66); S2(7.07, 2.93); S3(8.40, 7.27)

Об'єднання отриманих точок S1, S2, S3 в новий трикутник та знаходження точки Штейнера для нового трикутника S4(6.54, 3.78)

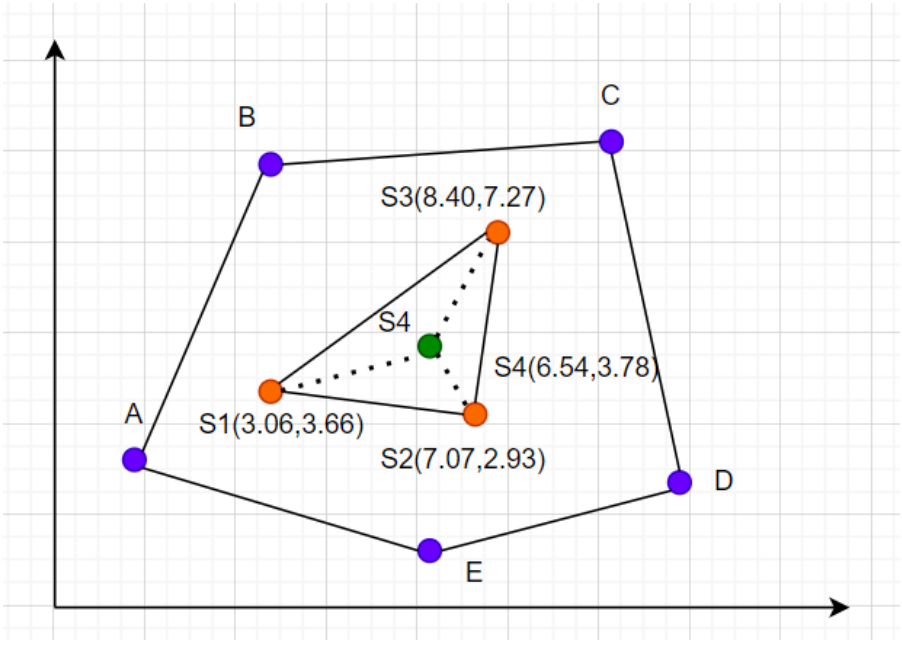

Рис. 2.10 Результуюча ТШ для графа з п’яти точок

Нова точка в отриманому трикутнику і буде розрахунковою точкою Штейнера для початкових п’яти вершин.

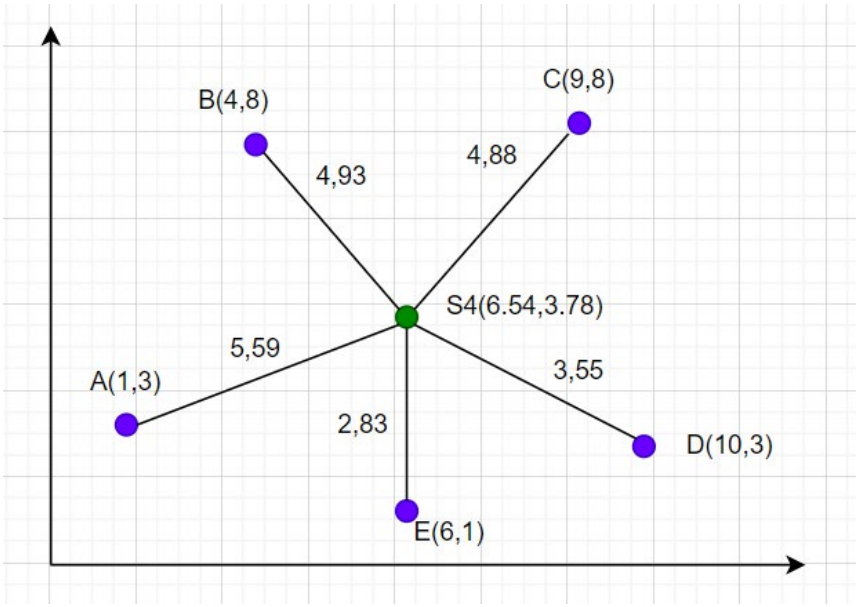

Рис. 2.11 ДШ для 5-и точок прямим з’єднанням

Для порівняння знайдемо сумму всіх ребер в отриманом дереві Штейнера та мінімальним остовним деревом для п’ятьох початкових точок за допомогою програми EMST.

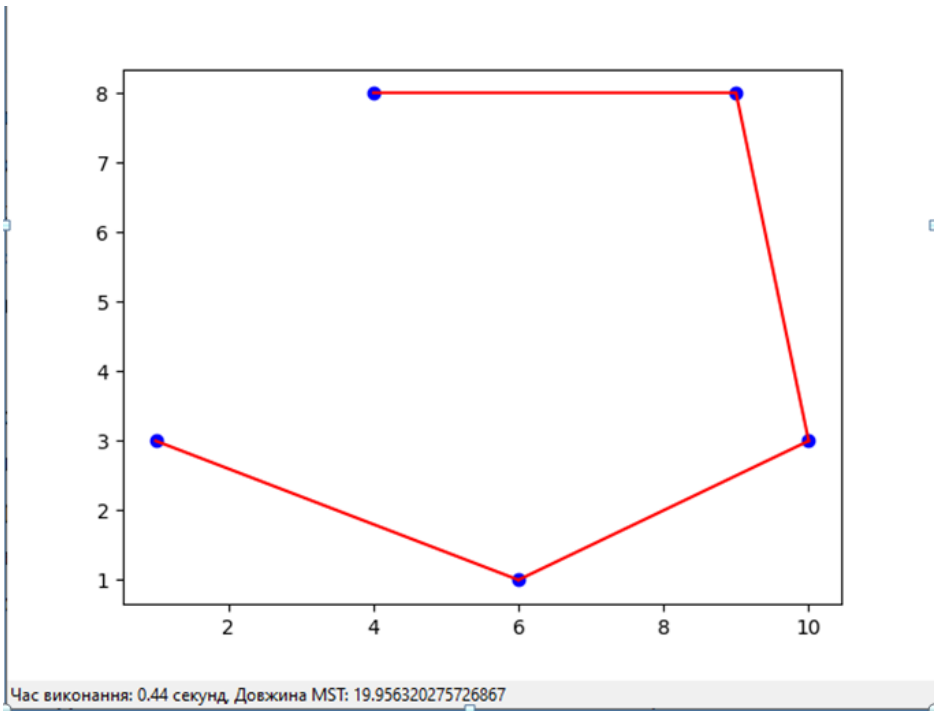

Рис. 2.12 МОД для п’яти точок за допомогою EMST

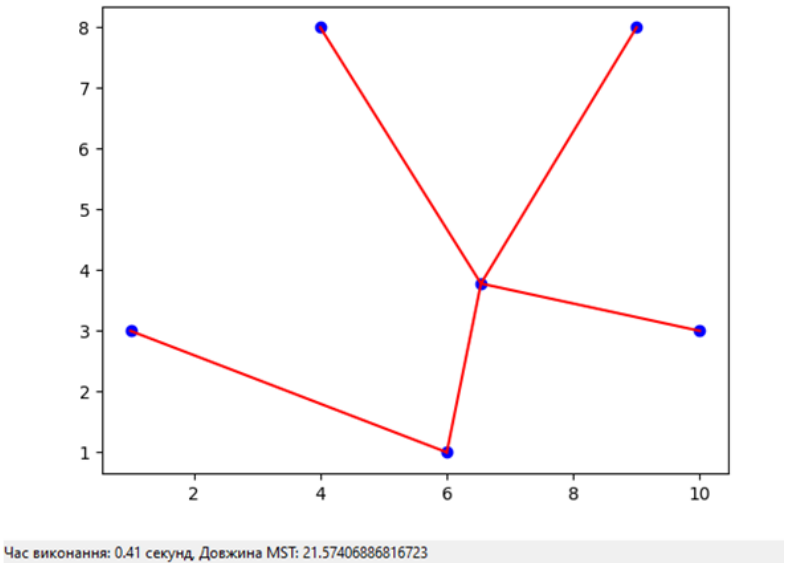

Рис. 2.13 ДШ для п’яти точок за допомогою EMST

Довжина ребер ДШ прямим з’єднанням(Рис. 2.11): 21.78.
(A–S4=5.59; B–S4=4.93; C–S4=4.88; D–S4=3.55; E–S4=2.83).
Довжина МОД за допомогою EMST(Рис 2.12): 19.9.
Довжина ДШ за допомогою програми EMST(Рис. 2.13): 21,5.

2.3 Для шести точок

Вибір Точок: Візьмемо шість точок, розміщених випадково на площині з такими координатами:

A (1, 2); B(2, 7); C(6, 5); D(8, 3.5);E(5, 1);F(3, 3).

потім поділимо площу в середині графа на трикутники (Рис 2.14).

Вага ребер графа дорівнює відстані між його точками:

A-B=5.10; B-C=4.47; C-D=2.50; D-E=3.91; E-A=4.12;

A-F=2.24; B-F=4.12; C-F=3.61; D-F=5.03; E-F=2.83.

Тепер за допомогою програми Steiner Point Calculator знайдемо точки штейнера для кожного трикутника (Рис. 2.15):

S1=(A; B; F) = (2.53; 3.19)

S2=(F; B; C) = (3.73; 4.72)

S3=(F; C; D) = (5.99; 4.69);

S4=(F; D; E) = (5.01; 1.81);

S5=(F; E; A) = (2.95; 2.71);

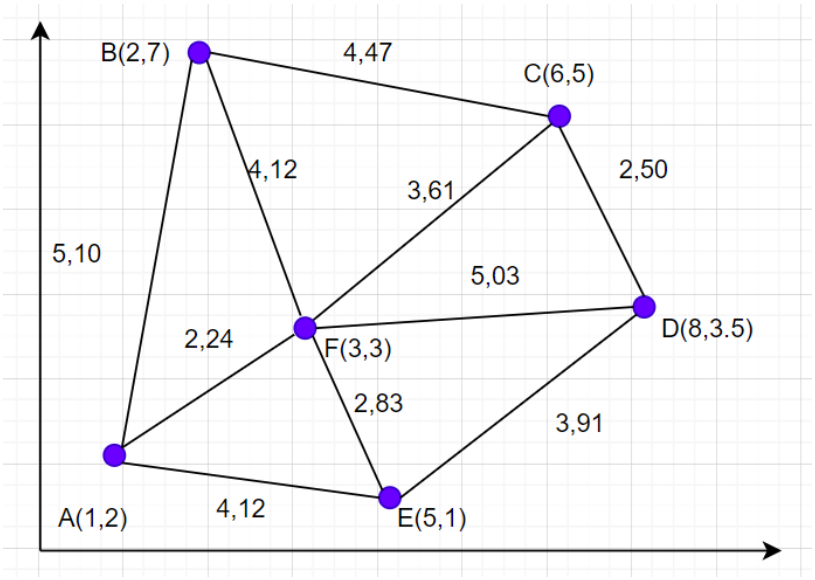

Рис. 2.14 Початкові точки графа з шести вершин

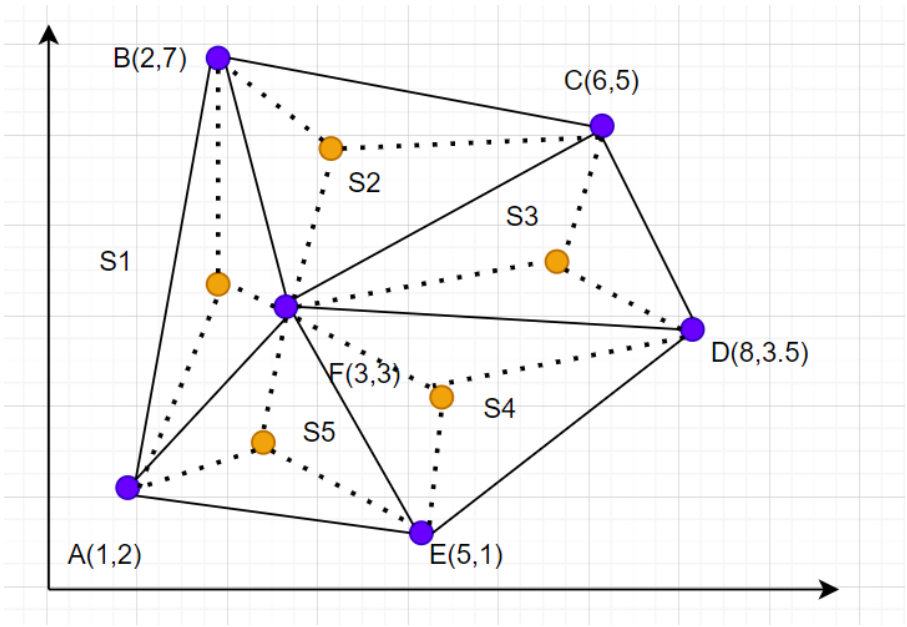

Рис. 2.15 Розділення графа з шести точками на кластери

Далі розділяємо площу графа з п’яти точок на три трикутника та знаходимо для кожного точку Штейнера H1, H2, H3 (Рис. 2.16).

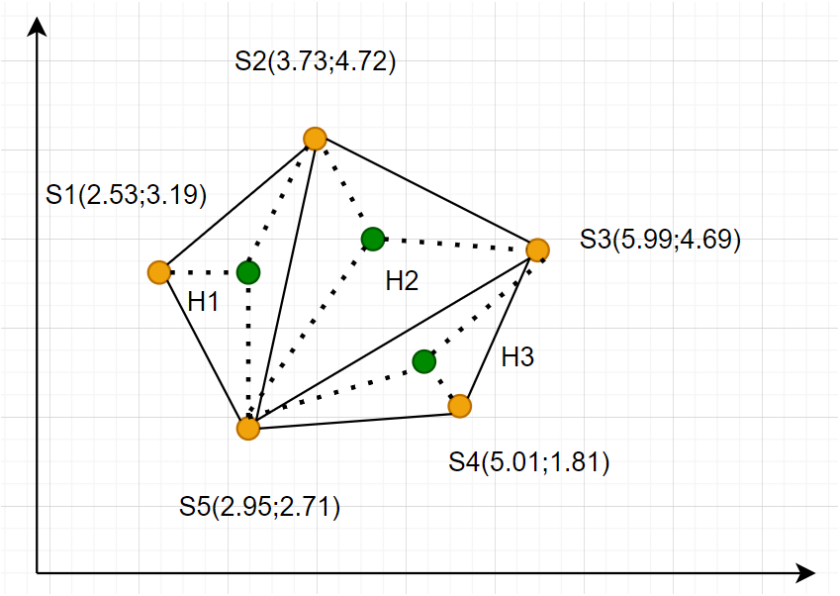

Рис. 2.16 ТШ для для кластерів другої ітерації

H1(S1; S2; S5)=(2.71;3.18)

H2(S2; S3; S5)=(3.85; 4 .54)

H3(S3; S4; S5)=(4.61; 2.61)

Точка Штейнера (Рис. 2.17, 2.18) для трикутника H1-H2-H3 буде передбачуваною точкою Штейнера (T) для шести початкових вершин графа T(3.63; 3.48).

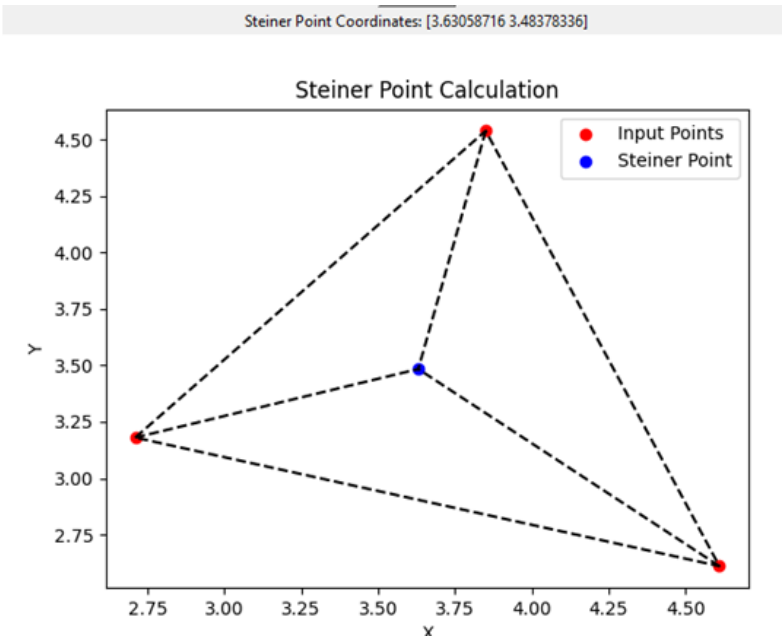

Рис. 2.17 Результуюча ТШ для графа з шести точок

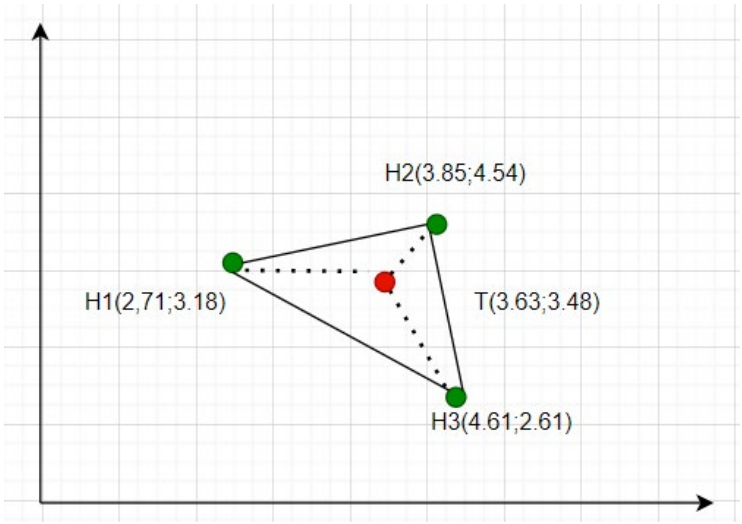

Рис. 2.18 Результуюча ТШ для графа з шести вершин

Для порівняння з МОД знайдемо суму всіх ребер отриманого дерева Штейнера за фор:

A-T = 3.02; B-T = 3.88; C-T = 2.82; D-T = 4.37; E-T = 2.83; F-T = 0.79.

Сумма дорівнює 17,71

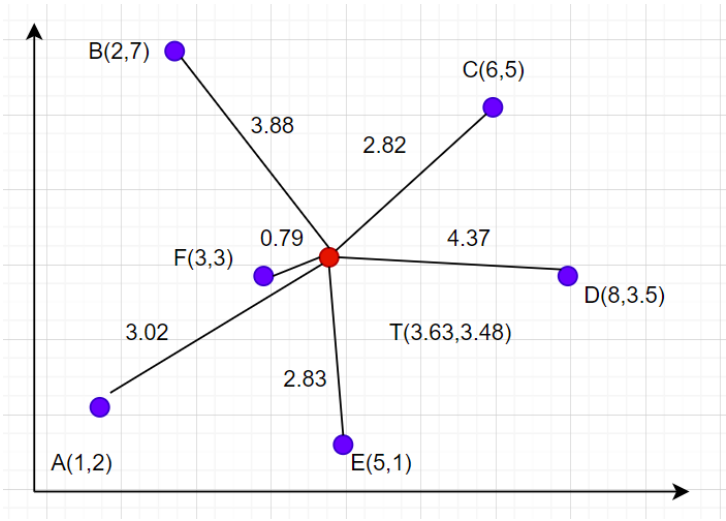

Рис. 2.19 ДШ для графа з шести точок прямим з’єднанням

Знайдемо МОД початкових шисти точок за допомогою програми EMST

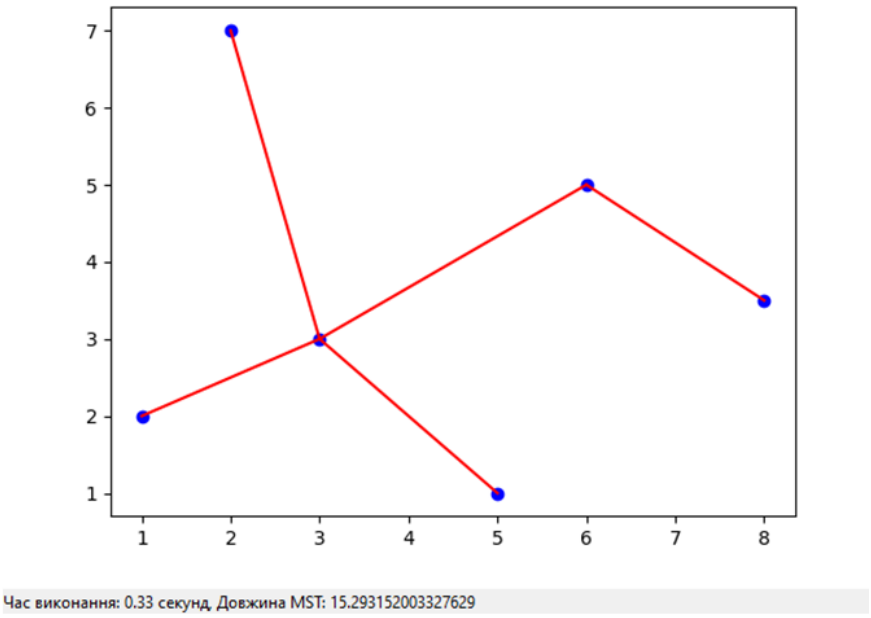

Рис. 2.20 МОД для графа з шести точок за допомогою EMS

Довжина МОД дорівнює 20,23.

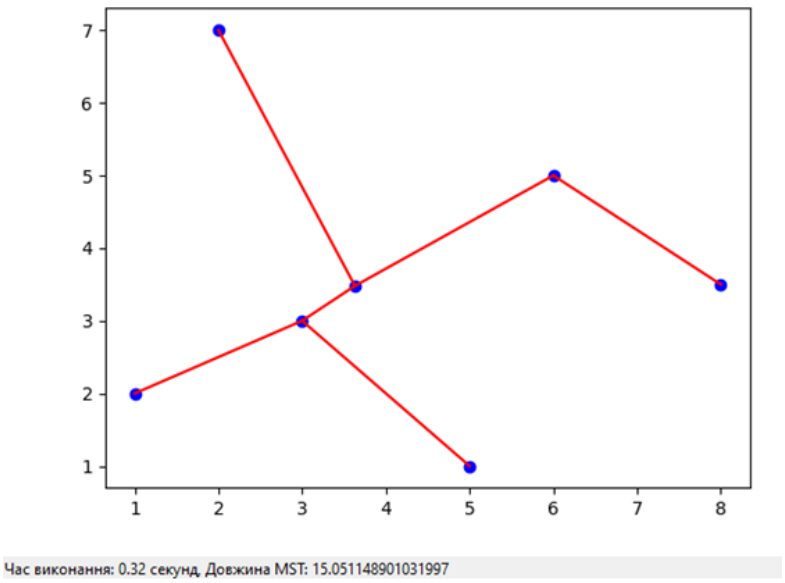

Рис. 2.21 ДШ для графа з шести точок за допомогою EMST

Довжина ребер ДШ прямим з’єднанням(Рис. 2.19): 17,71.

Довжина МОД за допомогою EMST(Рис 2.20): 15,29.

Довжина ДШ за допомогою програми EMST(Рис. 2.13): 15,05.

Висновки до другого розділа

При розробці вдосконаленого алгоритму знаходження точок Штейнера були проведені дослідження мінімальних остовних дерев да дерев Штейнера побудованих різними методами на графах з різною кількістю початкових вершин.

Чотири точки:

За методом прямого з'єднання та EMST отримано однакову довжину ребер (20,98), свідчачи про однакову ефективність обох методів у цьому конкретному випадку.

Метод МОД за допомогою EMST показав дещо вищу довжину ребер (21,28), що може вказувати на його меншу ефективність для невеликої кількості точок.

П'ять точок:

Пряме з'єднання дало найбільшу довжину ребер (21.78), що свідчить про його відносно меншу ефективність у цьому сценарії.

Метод EMST з деревом Штейнера показав кращі результати (21,5) порівняно з прямим з'єднанням.

Найкращі результати показав метод МОД за допомогою EMST (19.9), вказуючи на його вищу ефективність для цієї кількості точок.

Шість точок:

Пряме з'єднання знову показало найбільшу довжину ребер (17,71).

Метод EMST з деревом Штейнера виявився найефективнішим, демонструючи найменшу довжину ребер (15,05).

Метод МОД за допомогою EMST показав трохи вищу довжину ребер (15,29) порівняно з методом EMST з деревом Штейнера, але все ще кращі результати, ніж пряме з'єднання.

Загальні висновки:

1. Результати вказують на необхідність адаптації методів залежно від конкретних сценаріїв та кількості точок для досягнення оптимальних результатів.
2. У випадку з чотирма точками, методи прямого з'єднання ДШ та ДШ за допомогою програми EMST показали ідентичні результати, тоді як МОД за допомогою EMST був менш ефективним.
3. Для п'яти та шести точок метод МОД за допомогою EMST виявився більш ефективним, ніж інші методи, особливо порівняно з прямим з'єднанням.

3 РОЗРОБКА ЕКСПЕРИМЕНТАЛЬНОГО ЗРАЗКА ПРОГРАМИ РЕАЛІЗАЦІЇ АЛГОРИТМУ

Експериментальна программа для реалізації алгоритму знаходження дерева Штейнера складається з двох модулів: Steiner Point Calculation і EMST.

3.1 Steiner Point Calculation

Перший модуль створен для обчислення та візуалізації точки Штейнера в трикутнику. Програма використовує бібліотеки Python, включаючи Tkinter для створення графічного інтерфейсу користувача (GUI), Matplotlib для малювання графіків та NumPy для математичних обчислень. Основна частина програми, функція обчислення точки Штейнера, здійснюється за допомогою методу оптимізації Nelder-Mead, який є частиною бібліотеки scipy.optimize. Цей метод дозволяє знайти точку всередині трикутника, яка мінімізує загальну відстань до трьох вершин.

Інтерфейс користувача програми простий і інтуїтивно зрозумілий: користувач вводить координати трьох точок, що формують трикутник, а програма обчислює та відображає координати точки Штейнера. Результат представлений як текстове повідомлення з координатами точки та візуалізацією на графіку, де зображено вхідні точки, їхнє з'єднання та розташування точки Штейнера. На рис. 3.1 представлен інтерфейс модуля Steiner Point Calculation. Код модуля приведен в додатку А.

Виконання коду відбувається наступним чином:

Імпорт необхідних бібліотек:

- tkinter для створення графічного інтерфейсу користувача (GUI).
- messagebox з tkinter для відображення повідомлень про помилки.
- matplotlib.pyplot і FigureCanvasTkAgg для малювання графіків.

- numpy для математичних операцій.
- minimize з scipy.optimize для використання методів оптимізації.

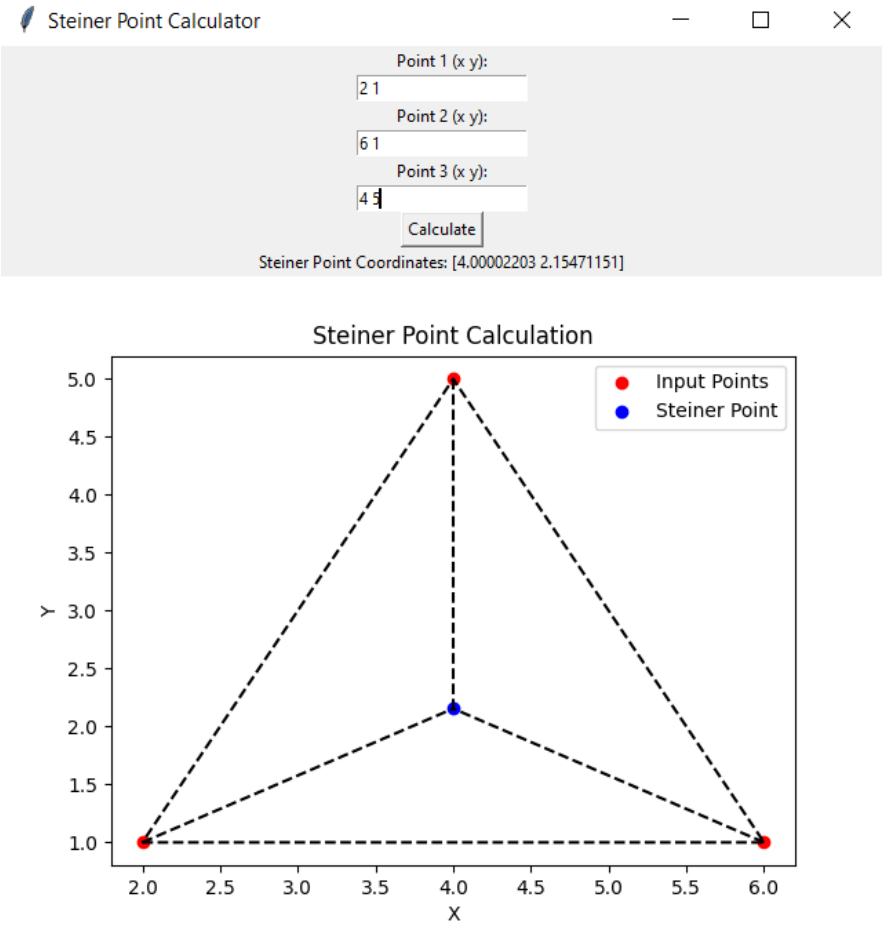

Рис. 3.1 Інтерфейс модуля Steiner Point Calculation

Опис функцій:

- calculate_steiner_point(p1, p2, p3): Розраховує точку Штейнера для заданих точок p1, p2, p3. Використовує метод оптимізації Nelder-Mead з scipy.optimize для мінімізації суми відстаней від точки до вершин трикутника.
- distance_sum(point, p1, p2, p3): Допоміжна функція всередині calculate_steiner_point, яка обчислює суму відстаней від заданої точки до трьох точок трикутника.

- on_calculate(): Зчитує координати з текстових полів, обчислює точку Штейнера, викликає функцію plot_graph для візуалізації результатів та виводить координати точки Штейнера.
- plot_graph(p1, p2, p3, steiner_point): Малює графік з вхідними точками та точкою Штейнера.

Створення графічного інтерфейсу користувача (GUI):

- Текстові поля для введення координат трьох точок.
- Кнопка "Calculate", яка викликає функцію on_calculate.
- Місце для відображення результатів розрахунків.
- Область для візуалізації графіка.

Робота програми:

- Вводиться координати трьох точок у текстові поля.
- Натискає кнопку "Calculate".
- Програма виконує функцію on_calculate, яка перетворює введені дані в числовий формат, перевіряє їх валідність та викликає calculate_steiner_point.
- calculate_steiner_point використовує метод оптимізації для знаходження точки Штейнера.
- Після знаходження точки Штейнера, plot_graph візуалізує вхідні точки та точку Штейнера на графіку.
- Координати точки Штейнера відображаються на інтерфейсі.

Обробка помилок:

- Якщо введені дані не відповідають формату або виникають інші помилки при обчисленнях, програма виводить відповідне повідомлення про помилку через messagebox.

Запуск головного циклу інтерфейсу:

- Виклик root.mainloop() запускає головний цикл інтерфейсу, що дозволяє користувачеві взаємодіяти з програмою.

3.2 EMST (Мінімальне остовне дерево)

Другий модуль програми зосереджений на знаходженні мінімального остовного дерева (МОД) в графі, використовуючи генетичний алгоритм. Код написан на мові програмування Python. Повний текст наведено в додатку Б. Інтерфейс приведено на рисунке 3.2.

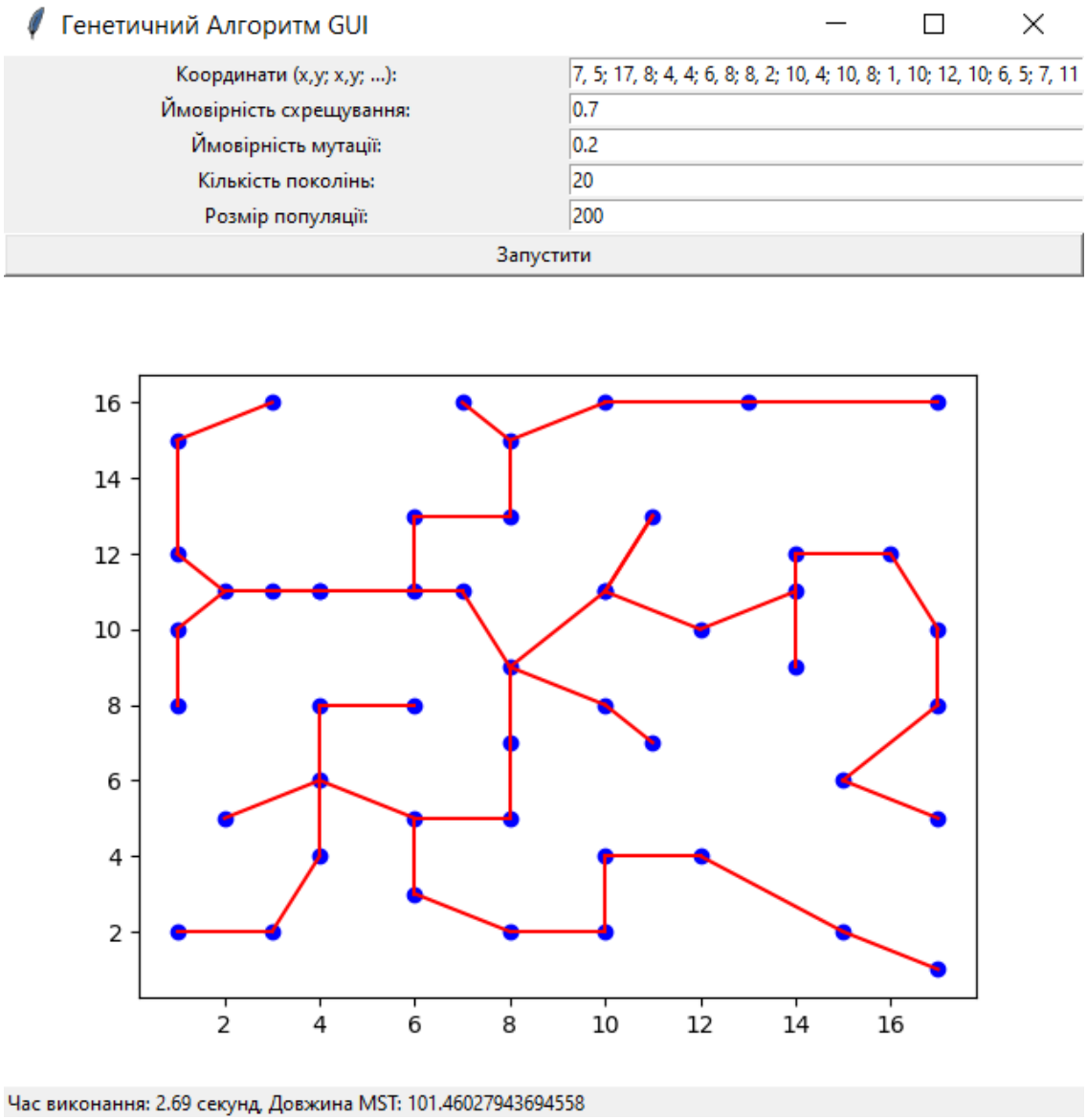

Рис. 3.2 Інтерфейс модуля EMST

Програма дозволяє вводити ключові параметри генетичного алгоритму, такі як ймовірність схрещування, ймовірність мутації, кількість поколінь, та розмір популяції. Ці параметри визначають поведінку та ефективність алгоритму в процесі пошуку оптимального рішення. Така можливість введення параметрів надає користувачам більше контролю над процесом оптимізації та дозволяє експериментувати з різними налаштуваннями для досягнення кращих результатів. У текстовому полі інтерфейсу користувача вводяться координати точок, які формують граф. Координати вводяться у вигляді пар значень, розділених комами та крапками з комою між різними парами. В результаті роботи програми на основі введених початкових даних видаються такі результати:

- Час виконання
- Візуальне представлення МОД.
- Сума всіх ребер отриманого МОД.

Виконання коду відбувається наступним чином:

Популяція та індивіди:

Початкові дані вводяться через поля вводу (Entry).

Після вводу даних та запуску відбувається ініціалізація: Спочатку програма отримує координати точок, що репрезентують вершини графа. На основі цих координат створюється матриця відстаней, яка використовується для представлення зваженого графа. Координати точок отримуються через графічний інтерфейс користувача (GUI). Це реалізовано в функції run():

```
coord_input = coord_entry.get()
coordinates = np.array([list(map(float, point.split(','))) for point in
coord_input.split(';')])
```

координати вводяться в поле вводу coord_entry, і вони перетворюються на масив numpy. Матриця відстаней створюється в функції main():

dist_matrix = distance_matrix(coordinates, coordinates)

Вона обчислює відстані між усіма парами точок, використовуючи координати.

Представлення індивідів: В генетичному алгоритмі кожен індивід представляє потенційне рішення задачі. В нашому випадку, індивід — це певна перестановка вершин графа, яка вказує порядок, у якому вершини будуть з'єднані. У контексті генетичного алгоритму, кожен індивід представляє можливе рішення задачі. В цій програмі, задача полягає в знаходженні мінімального остовного дерева для заданого набору точок. Кожен індивід в генетичному алгоритмі представлений як перестановка індексів точок, що вказує порядок, в якому точки (вершини графа) будуть з'єднані[8].

Це представлення реалізовано наступним чином у вашому коді:

toolbox.register("indices", random.sample, range(len(coordinates)), len(coordinates))
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.indices)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

toolbox.register("indices", random.sample, range(len(coordinates)), len(coordinates)):

Цей рядок створює функцію indices у toolbox, яка генерує випадкову перестановку індексів точок. Кожна перестановка є випадковим унікальним порядком індексів з набору точок.

toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.indices):

Тут individual стає функцією, яка створює новий індивід, використовуючи клас Individual, визначений за допомогою creator. Цей індивід ініціалізується викликом функції indices, тобто він представляє собою перестановку індексів.

toolbox.register("population", tools.initRepeat, list, toolbox.individual):

Цей рядок створює функцію для генерації початкової популяції індивідів. Популяція складається з декількох індивідів (рішень), які будуть еволюціонувати в процесі виконання генетичного алгоритму.

Кожен індивід у популяції представляє потенційне рішення задачі МОД. Генетичний алгоритм працює з цією популяцією, використовуючи процеси селекції, схрещування та мутації для знаходження найкращого можливого рішення.

Оцінка пристосованості через фітнес-функцію (fitness function) є критичною складовою генетичного алгоритму, оскільки вона визначає, наскільки добре кожен індивід вирішує задану задачу. Фітнес-функція оцінює індивідів та присвоює їм фітнес-оцінку, яка зазвичай відображає вартість (cost), здатність до виживання або якість рішення, що індивід пропонує. У задачах оптимізації, фітнес-функція часто розраховується як зворотня величина вартості рішення, де нижча вартість відповідає вищому фітнесу. У класифікаційних або предиктивних задачах фітнес може бути визначений як точність або інша метрика якості передбачення[8].

Фітнес-функція повинна бути добре визначена та відповідати цілям оптимізації. Вона також повинна бути обчислювально ефективною, оскільки буде використовуватися багатократно протягом усього процесу еволюції.

В наведеній програмі фітнес-функцію виконує функція evalMST:

```
def evalMST(individual):
   mst_matrix = minimum_spanning_tree(dist_matrix[individual][:,
individual])
   total_length = mst_matrix.sum()
   return (total_length,)
```

- Ця функція обчислює мінімальне остовне дерево (MST) для підмножини точок, представлених індивідом, і використовує загальну довжину дерева як оцінку пристосованості. Менша загальна довжина остовного дерева свідчить про кращу пристосованість індивіда.

mst_matrix = minimum_spanning_tree(dist_matrix[individual][:, individual]):

Функція **minimum_spanning_tree** з бібліотеки scipy використовується для побудови мінімального остовного дерева.
dist_matrix[individual][:, individual] формує матрицю відстаней для підмножини точок, яка визначається поточною перестановкою індексів **individual**. Це означає, що ми розглядаємо тільки відстані між точками, які входять у цей індивід, згідно з порядком їх перестановки.

- Обчислення загальної довжини МОД: total_length = mst_matrix.sum():

Цей рядок обчислює загальну довжину мінімального остовного дерева, сумуючи ваги ребер, які входять в дерево.

- Повернення фітнес-значення: **return (total_length,):**

Функція повертає загальну довжину як фітнес-значення індивіда. У генетичному алгоритмі це значення використовується для порівняння індивідів: чим менша загальна довжина, тим краще індивід.

Ця фітнес-функція використовується в генетичному алгоритмі для визначення того, наскільки добре кожен індивід вирішує задачу побудови мінімального остовного дерева. Вона визначає "пристосованість" кожного індивіда, яка пізніше використовується при відборі для схрещування та формування наступних поколінь.

Селекція: Вибір найкращих індивідів для схрещування. Використовуються механізми відбору, такі як турнірний відбір, для визначення індивідів, які будуть брати участь у наступному поколінні.

В програмі для відбору індивідів, які братимуть участь у наступному поколінні генетичного алгоритму, використовується турнірний відбір.

toolbox.register("select", tools.selTournament, tournsize=3)

toolbox.register використовується для додавання функції відбору до інструментарію DEAP.
tools.selTournament є функцією турнірного відбору, яка входить в стандартний набір інструментів DEAP.
tournsize=3 означає, що кожен турнір буде включати трьох індивідів.

Турнірний відбір - це метод відбору, при якому випадковим чином обираються декілька індивідів з популяції, і серед них обирається найкращий (з найвищим фітнес-значенням) для участі у схрещуванні та формуванні

нового покоління. У нашому випадку розмір турніру встановлено в 3, що означає, що у кожному турнірі беруть участь три індивіди, і найкращий з них вибирається для подальшої еволюції. Цей метод дозволяє збалансувати дослідження (різноманітність генетичного матеріалу) та використання (збереження та поширення сильних генів), що є важливим для ефективності генетичних алгоритмів. Він також допомагає уникнути передчасної конвергенції, коли алгоритм застрягає в локальному максимумі, не досягаючи глобального оптимального рішення[8].

Схрещування (Кросовер) виконується в рамках функції algorithms.eaSimple, яка є частиною бібліотеки DEAP. Ця функція використовується для виконання основного циклу генетичного алгоритму, що включає селекцію, кросовер, мутацію та оцінку індивідів.

Коли викликається algorithms.eaSimple, він автоматично організовує процес генетичного алгоритму, виконуючи схрещування на кожному кроці (поколінні) [8].

Селекція Батьківських Індивідів: Спочатку з популяції вибираються індивіди для участі у схрещуванні. Вибір базується на їх фітнес-значеннях, і цей процес здійснюється за допомогою раніше зареєстрованої функції селекції (toolbox.select). Вибрані індивіди потім підлягають процесу кросоверу, під час якого пари індивідів комбінуються для створення потомства, яке наслідує генетичні характеристики обох батьків. Оператор кросоверу (toolbox.mate).

toolbox.register("mate", tools.cxOrdered)

tools.cxOrdered є методом схрещування, який використовується для перестановок. Він забезпечує, що кожен елемент з'являється лише один раз у кожному дитинчаті. Цей метод особливо корисний для задач, де індивіди представляють перестановки, як у випадку з задачею МОД.

Мутація вносить випадкові зміни в індивідів, що допомагає підтримувати генетичне різноманіття в популяції та запобігає ранній конвергенції алгоритму до локального максимуму. Це дозволяє алгоритму досліджувати різні області простору рішень. У програмі ці два механізми (мутація та кросовкер) співпрацюють для поступового вдосконалення популяції індивідів, направляючи пошук до оптимального або близького до оптимального рішення задачі побудови мінімального остовного дерева.

toolbox.register("mutate", tools.mutShuffleIndexes, indpb=mutation_prob)

Цей рядок коду визначає оператор мутації:
tools.mutShuffleIndexes є методом мутації для перестановок, який випадково змінює позиції елементів у перестановці.
indpb (імовірність зміни кожного індексу) - це параметр, який контролює, наскільки часто будуть відбуватися зміни. Він встановлюється через графічний інтерфейс користувача. Схрещування забезпечує обмін генетичною інформацією між різними індивідами, що дозволяє створювати нові рішення з існуючих хороших характеристик.

Повторення процесу: Ці етапи повторюються багато разів (поколінь). З кожним поколінням ГА намагається поліпшити рішення, поки не буде знайдено оптимальне або поки не буде досягнуто задану кількість поколінь.
У програмі повторення процесу генетичного алгоритму через декілька поколінь реалізовано за допомогою функції algorithms.eaSimple з бібліотеки DEAP[8].

pop, log = algorithms.eaSimple(pop, toolbox, crossover_prob, mutation_prob, generations, stats=stats, halloffame=hof, verbose=True)

Цей рядок коду викликає функцію eaSimple, яка представляє собою простий генетичний алгоритм. Основні компоненти цього виклику:

pop: Початкова популяція індивідів.

toolbox: Інструментарій, який містить зареєстровані функції для генетичного алгоритму, включаючи селекцію, схрещування, мутацію та оцінку.

crossover_prob: Імовірність схрещування.

mutation_prob: Імовірність мутації.

generations: Кількість поколінь, які будуть пройдені генетичним алгоритмом.

stats: Об'єкт статистики для збору даних про процес еволюції.

halloffame: Об'єкт, що зберігає найкращих індивідів.

verbose: Якщо True, виводить додаткову інформацію під час виконання. Процесс працює так:

- Ініціалізація: Починається з початкової популяції індивідів, що були згенеровані на етапі ініціалізації.

- Виконання Поколінь: Для кожного покоління відбуваються процеси селекції, схрещування, мутації та оцінки. Це включає вибір індивідів для розмноження, застосування операторів схрещування та мутації для створення нових індивідів, а потім оцінку цих нових індивідів.

- Оновлення популяції: Нове покоління індивідів замінює старе, і процес повторюється.

- Збір статистики та відбір кращих: Протягом процесу збирається статистика про покоління, і найкращі індивіди зберігаються в halloffame.

- Завершення: Процес повторюється задану кількість поколінь (generations). Після досягнення цієї кількості, алгоритм завершується, і краще знайдене рішення може бути представлено користувачу.

Цей повторюваний процес дозволяє генетичному алгоритму поступово поліпшувати рішення задачі, використовуючи принципи еволюції та природного відбору.

Візуалізація та виведення результатів: Після завершення обчислень програма візуалізує найкраще знайдене МОД і виводить час виконання алгоритму. Графічний інтерфейс, створений за допомогою Tkinter, дозволяє користувачам вводити дані та відображати результати в зручному форматі.
Після знаходження найкращого рішення (МОД) воно візуалізується за допомогою бібліотеки matplotlib.

Час початку та завершення вимірюється, а різниця між ними визначає загальний час виконання. Результати візуалізації, час виконання та загальна довжина МОД відображаються у вікні програми.

4 ДОСЛІДЖЕННЯ ХАРАКТЕРИСТИК РОЗРОБЛЕННОГО ПІДХОДУ ДО РОЗВ’ЯЗАННЯ ЗАВДАННЯ

Знаходження мінімального остовного дерева та дерева Штейнера є важливими, але різними задачами в теорії графів, які тісно пов'язані між собою. Обидві ці задачі центруються навколо ідеї оптимізації мережевих структур, але з різними підходами та цілями. Задача знаходження МОД полягає у визначенні підграфа, який з'єднує всі вершини графа з мінімальною сумарною вагою. У випадку дерева Штейнера, задача стає більш складною через можливість включення додаткових точок для подальшої оптимізації загальної ваги.

Ці задачі мають велике значення у мережевій оптимізації, наприклад, при плануванні телекомунікаційних мереж або електромереж. Дерево Штейнера можна розглядати як узагальнення концепції МОД, де додавання додаткових вузлів дозволяє досягнути більш ефективного розв'язку. Розв'язання обох задач вимагає застосування складних алгоритмів. Для МОД існують ефективні алгоритми, такі як алгоритми Прима або Краскала, але задача дерева Штейнера є NP-складною і вимагає застосування більш складних методів, як-от генетичні алгоритми або інші евристичні методи оптимізації.

В ході дослідження буде проведено серію експериментів із змінними параметрами для оцінки того, як вони впливають на час виконання алгоритму та якість мінімального остовного дерева. Це дозволить оптимізувати роботу генетичного алгоритму для конкретних умов та глибше зрозуміти принципи його роботи та області застосування.

Основними параметрами, які будуть розглядатися, є ймовірність схрещування, ймовірність мутації, кількість поколінь, кількість точок та розмір популяції. Основник критерієм оцінки буде час виконання.

Ймовірність схрещування (crossover probability) - це параметр генетичного алгоритму, який визначає, з якою ймовірністю відбудеться схрещування між двома особинами (індивідами) у популяції. Схрещування - це процес, під час якого відбувається обмін генетичною інформацією між батьківськими хромосомами для створення нового покоління особин, які містять характеристики обох батьків. Цей параметр зазвичай варіюється від 0 до 1 (або від 0% до 100%), де значення близьке до 0 означає, що схрещування відбудеться дуже рідко, тоді як значення близьке до 1 означає, що схрещування відбуватиметься майже завжди при зустрічі двох особин.

Вибір оптимальної ймовірності схрещування може залежати від конкретної задачі та властивостей популяції. Висока ймовірність схрещування сприяє різноманітності генетичного матеріалу в популяції, але також може призвести до втрати високоякісних генетичних комбінацій. З іншого боку, низька ймовірність схрещування може призвести до більшої стабільності у популяції, але також може сповільнити процес еволюції та покращення рішень[8].

Ймовірність мутації (mutation probability) в генетичних алгоритмах — це параметр, який визначає частоту з якою відбуваються випадкові зміни в генах індивіда популяції. Мутація — це процес, при якому один або кілька генів особини (індивіда) зазнають випадкових змін, що може призвести до нових варіантів генотипів в популяції.

Так само як і ймовірність схрещування, ймовірність мутації зазвичай виражається числом від 0 до 1. Висока ймовірність мутації може збільшити генетичну різноманітність та допомогти уникнути локальних мінімумів оптимізації, але також може призвести до втрати добре пристосованих особин. З іншого боку, низька ймовірність мутації зменшує ризик руйнування високоякісних рішень, але може сприяти передчасній конвергенції, коли алгоритм зациклюється на певному рішенні і не може знайти краще[8].

Оптимальний рівень ймовірності мутації залежить від багатьох факторів, включно з розміром популяції, структурою проблеми та іншими параметрами алгоритму. Цей параметр потребує тонкого налаштування для досягнення балансу між дослідженням нових областей пошукового простору та експлуатацією вже знайдених хороших рішень.

Кількість поколінь у генетичному алгоритмі визначає загальну кількість ітерацій еволюційного циклу, який включає вибір особин, схрещування, мутацію та селекцію. Це числовий параметр, який встановлює, скільки разів популяція буде оновлюватися у спробі покращити рішення проблеми. Мале значення кількості поколінь може не бути достатнім для того, щоб генетичний алгоритм досягнув оптимального рішення, особливо при складних задачах оптимізації, де потрібно більше часу для експлорації простору рішень. Велике значення може забезпечити кращу конвергенцію та більшу імовірність знаходження оптимального чи близького до оптимального рішення, але також може призвести до зайвих обчислень і збільшення часу виконання алгоритму, особливо якщо він швидко знаходить задовільне рішення[8].

Вибір правильної кількості поколінь залежить від багатьох факторів, включаючи складність задачі, розмір популяції, ймовірності схрещування та мутації, а також часу та обчислювальних ресурсів, які є доступними для вирішення проблеми. Іноді кількість поколінь встановлюється емпірично через серію експериментів.

Розмір популяції у генетичному алгоритмі визначає кількість індивідів (особин), які будуть існувати в кожному поколінні. Цей параметр має істотний вплив на пошукові можливості алгоритму та його здатність уникати локальних оптимумів.

Малий розмір популяції може призвести до більш швидкого виконання алгоритму, але існує ризик, що алгоритм може не знайти оптимальне рішення

через недостатню генетичну різноманітність, що може призвести до передчасної конвергенції.

Великий розмір популяції збільшує генетичну різноманітність та може покращити здатність алгоритму досягати глобального оптимуму, але це також збільшує обчислювальні витрати, оскільки кожен індивід потрібно оцінити, а кількість генетичних операцій зростає.

Оптимальний розмір популяції залежить від багатьох факторів, таких як складність задачі, структура простору рішень, і час, який доступний для виконання алгоритму. Вибір правильного розміру популяції часто є компромісом між різноманітністю та обчислювальною ефективністю. У деяких випадках розмір популяції може бути динамічно змінюваним протягом еволюції, адаптуючись до прогресу пошуку оптимального рішення[8].

4.1 Дослідження впливу параметрів моделі на час виконання алгоритму

Встановлюємо початкові параметри з такими величинами:

Кількість точок: 50

Координати точок: 1, 2; 2, 5; 1, 8; 4, 6; 4, 8; 3, 11; 6, 11; 6, 3; 8, 9; 8, 7; 8, 5; 10, 11; 11, 7; 12, 4; 10, 2; 3, 2; 1, 12; 2, 11; 1, 15; 3, 16; 4, 11; 6, 13; 8, 15; 10, 11; 13, 16; 17, 16; 14, 11; 16, 12; 17, 10; 14, 9; 15, 6; 15, 2; 17, 1; 8, 13; 14, 12; 11, 13; 7, 16; 4, 11; 10, 16; 17, 5; 17, 8; 4, 4; 6, 8; 8, 2; 10, 4; 10, 8; 1, 10; 12, 10; 6, 5; 7, 11

Ймовірність схрещування: 0,7

Ймовірність мутації: 0,2

Кількість поколінь: 20

Розмір популяції: 200

Час виконання алгоритму: 1,75 сек. Результати представлені на рис. 4.1

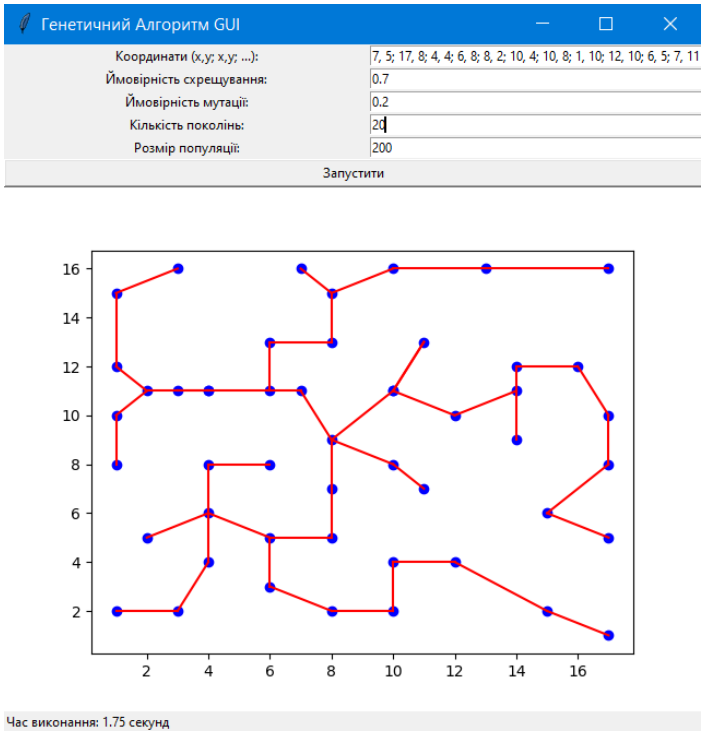

Рис. 4.1 Результати виконання програми

Проаналізуємо час виконання алгоритму при зміні кожного параметра.

Залежність часу виконання від кількості точок представлена в таблиці 4.1 та рис. 4.2.

Таблиця 4.1 Час виконання від кількості точок

Кількість точок	10	20	30	40	50
Час виконання, сек	1.05	1.15	1.31	1.5	1.72

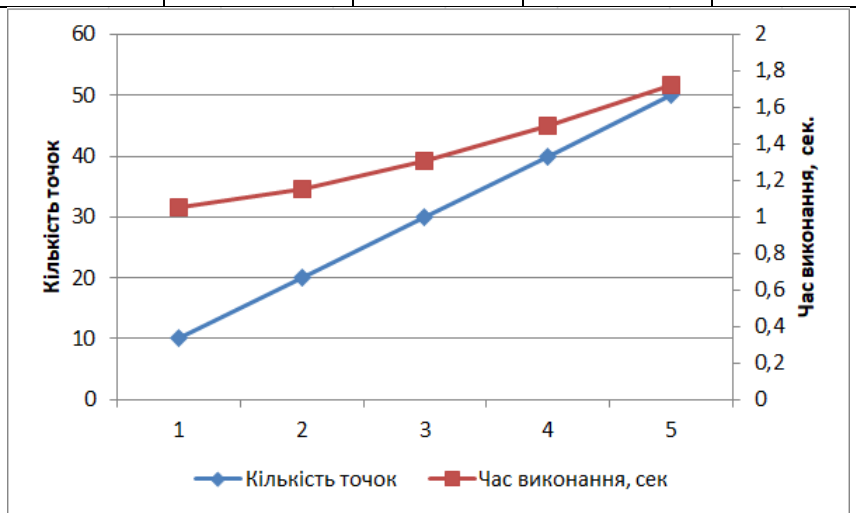

Рис. 4.2 Час виконання від кількості точок

Залежність часу виконання від ймовірності схрещування представлено в таблиці 4.2 та рис. 4.3.

Таблиця 4.2 Час виконання від ймовірності схрещування

Ймовірність схрещування	0,1	0,3	0,5	0,7	0,9
Час виконання, сек	0.75	1.09	1.39	1.71	2.05

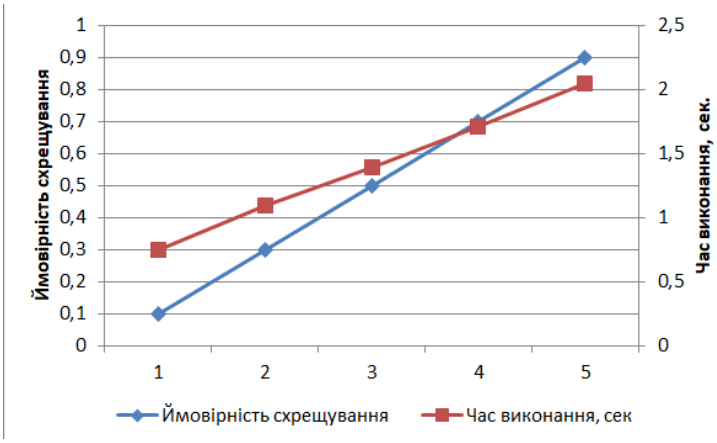

Рис. 4.3 Час виконання від ймовірності схрещування

Залежність часу виконання від ймовірності мутації представлено в таблиці 4.3 та рис. 4.4.

Таблиця 4.3 Час виконання від ймовірності мутації

Ймовірність мутації	0,1	0,3	0,5	0,7	0,9
Час виконання, сек	1.69	1.79	1.92	2.08	2.23

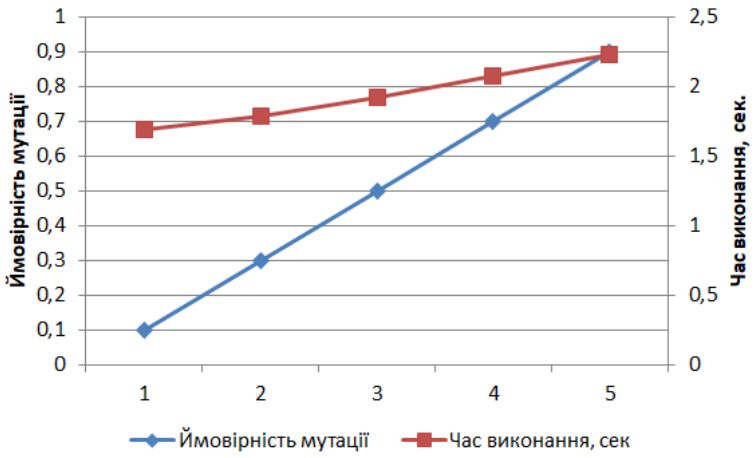

Рис. 4.4 Час виконання від ймовірності мутації

Залежність часу виконання від кількості поколінь представлено в таблиці 4.4 та рис. 4.5.

Таблиця 4.4 Час виконання від кількості поколінь

Кількість поколінь	10	20	30	40	50
Час виконання, сек	0,9	1,73	2,52	3,34	4,15

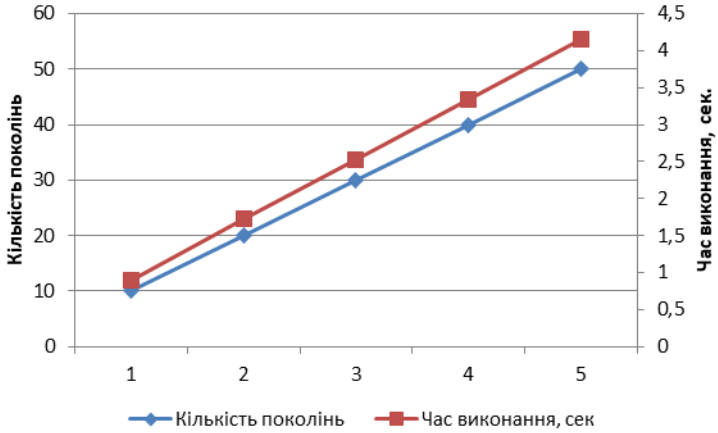

Рис. 4.5 Час виконання від кількості поколінь

Залежність часу виконання від розміру популяції представлено в таблиці 4.5 та рис. 4.6.

Таблиця 4.5 Час виконання від розміру популяції

Розмір популяції	100	130	150	170	200
Час виконання, сек	0,86	1,12	1,32	1,45	1,72

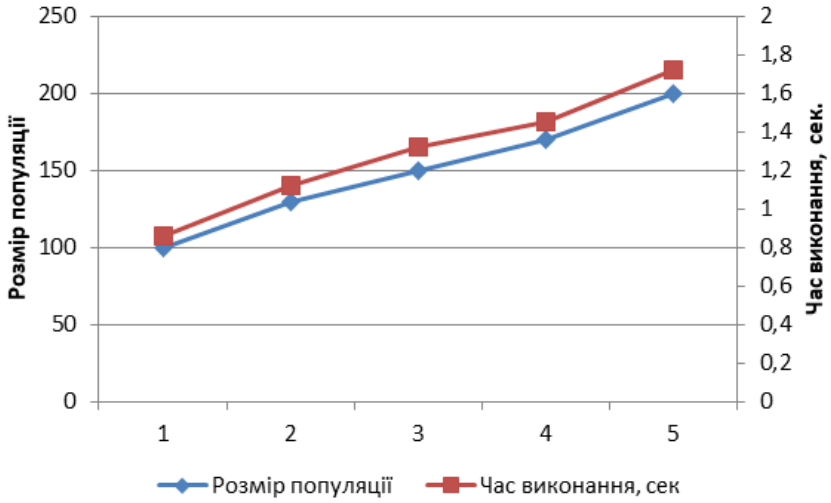

Рис. 4.6 Час виконання від розміру популяції

Висновки до четвертого розділу

Залежність часу виконання від кількості точок: Час виконання алгоритму збільшується із зростанням кількості точок. Це пов'язано з тим, що більша кількість точок вимагає більше обчислень для оцінки фітнес-функцій та знаходження оптимальних рішень.

Залежність часу виконання від ймовірності схрещування: Із збільшенням ймовірності схрещування час виконання також зростає. Схрещування є ключовим етапом у генетичних алгоритмах, і вища ймовірність схрещування означає більше обчислень для генерації нових рішень.

Залежність часу виконання від ймовірності мутації: Зі збільшенням ймовірності мутації спостерігається зростання часу виконання. Мутації додають варіативність до популяції, але також вимагають додаткових обчислень для аналізу нових генетичних варіантів.

Залежність часу виконання від кількості поколінь: Чим більше кількість поколінь, тим більше часу потрібно для завершення роботи алгоритму. Кожне покоління вимагає обчислення фітнес-функцій, схрещування, мутації та відбору, що збільшує загальний час обчислень.

Залежність часу виконання від розміру популяції: Чим більша розмір популяції, тим довше триває виконання алгоритму. Більша популяція означає більше кандидатів для оцінки та відбору, що логічно призводить до збільшення обчислювальних витрат.

Узагальнюючи результати підтверджується той факт, що кожен параметр генетичного алгоритму має вплив на час його виконання, і це важливо враховувати при налаштуванні алгоритму для конкретної задачі. Налаштування параметрів таким чином, щоб забезпечити оптимальний баланс між часом виконання і якістю результатів, є важливим завданням при використанні генетичних алгоритмів.

ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи здійснено розробку вдосконаленого алгоритму знаходження точок Штейнера методом кластеризації, розроблено експериментальний зразок програми для реалізації алгоритму цього алгоритму та проведени дослідження характеристик розробленого підходу до розв'язання завдання.

Дослідження ефективності кластеризації в визначенні точок Штейнера показало, що використання цього методу може бути ефективним у зниженні обчислювальної складності, особливо у сценаріях з великою кількістю точок. Цей метод демонструє адекватну точність у побудові дерева Штейнера, що робить його корисним для оптимізації процесу розв'язання задач.

Детальне дослідження, виконане на графах з різною кількістю точок, вказує на те, що різні методи знаходження оптимальних рішень виявляються більш або менш ефективними залежно від конкретної кількості точок. Цей аналіз підкреслює необхідність адаптації алгоритму до специфіки задачі та підтверджує його універсальність і гнучкість.

Розроблений експериментальний зразок програми реалізації алгоритму дозволяє знаходити точку Штейнера для графа з трьох точок, а також знаходити мінімальне остовне дерево для графа з більшою кількістю вершин за допомогою генетичного алгоритму з можливістю зміни параметрів роботи алгоритму.

Порівняльний аналіз різних методів (пряме з'єднання, ДШ за допомогою програми EMST, МОД за допомогою EMST) показав, що ефективність кожного методу значно відрізняється залежно від кількості точок. Наприклад, виявилося, що для п’яти точок метод МОД за допомогою EMST є найбільш ефективним, тоді як для чотирьох точок методи прямого з'єднання та EMST показали ідентичні результати.

Базуючись на отриманих результатах, важливо провести подальші дослідження з використанням більшої кількості точок і різних конфігурацій графів. Це допоможе більш детально зрозуміти масштаби застосовності розробленого алгоритму та його ефективність у широкому спектрі задач.

Отримані результати можуть бути корисними в практичних застосуваннях, де необхідна оптимізація мережевих структур, наприклад, у телекомунікаціях або при плануванні транспортних мереж. Адаптація алгоритму до специфічних умов цих застосувань може значно підвищити їх ефективність.

ПЕРЕЛІК ПОСИЛАНЬ

1. Задача Ферма-Торричелли-Штейнера. URL: https://studbooks.net/2301045/matematika_himiya_fizika/zadacha_ferma_torrichelli_shteynera (Дата звернення: 15.10.2023).
2. Melzak, Z. A. Companion to Concrete Mathematics. John Wiley & Sons, Inc., 1973.
3. Winter, Pawel. Steiner Problem in Networks: A Survey. Networks, 17(2), 1987, 129–167.
4. The Shortesr-Network Problem URL: https://mathweb.ucsd.edu/~ronspubs/89_01_shortest_network.pdf
5. Gilbert, E. N., & Pollak, H. O. Steiner Minimal Trees. SIAM Journal on Applied Mathematics, 16(1), 1968, 1–29.
6. Ольшевський А.І., Починський М.Я. Вирішення завдання Штейнера за допомогою генетичного алгоритму. Біоніка інтелекту, №2(69), 2008, 145-151. (Дата звернення: 24.10.2023).
7. Ольшевський А.І. Алгоритми побудови маршруту при груповому розсиланні мережевих пакетів даних дистанційного навчання на базі ДонДДІІІ. Штучний інтелект, № 4, 2007, 483-490.
8. Wirsansky, E. Hands-On Genetic Algorithms with Python. Packt Publishing, 2020, 346 p.
9. Литвиненко В.А., Ховансков С.А., Максюта Д.Ю. Адаптивний алгоритм побудови дерева Штейнера, КПУ, 2016, 9 с.
10. Євтушенко, Ольга. Застосування тріангуляції Делоне для розв’язання евклідової задачі Штейнера. Наукові записки НаУКМА. Т. 177: Комп'ютерні науки, 2015, 62-68.
11. Євтушенко О.Я. Меметичний алгоритм для евклідової задачі Штейнера. НаУКМА. Т. 151: Комп'ютерні науки, 2015, 55-60.
12. Gnuplot library. URL: http://www.gnuplot.info/.

13. Библиотека алгоритмов вычислительной геометрии. URL: https://www.cgal.org/.
14. do Nascimento, Marcelo Zanchetta. An interactive programme for weighted Steiner trees. URL: https://www.facom.ufu.br/~nascimento/preprint_stree.pdf.
15. Introduction to Genetic Algorithms — Including Example Code. URL: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.
16. Генетичні алгоритми. Ключові поняття і методи реалізації. URL: http://www.znannya.org/?view=ga_general.
17. Эвристика построения решения задачи о емкостном дереве Штейнера. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0270147.
18. Что такое Дерево Штейнера? URL: https://www.geeksforgeeks.org/steiner-tree/.
19. GeeksforGeeks. Steiner Tree Problem. URL: https://www.geeksforgeeks.org/steiner-tree-problem/.
20. Papers With Code. Steiner Tree Problem. URL: https://paperswithcode.com/task/steiner-tree-problem.
21. Prömel, H. J., & Steger, A. The Steiner Tree Problem: A Tour Through Graphs, Algorithms, and Complexity. 1st ed. Braunschweig: Vieweg, 2002.
22. Євристический метод. URL: https://worldcomp-proceedings.com/proc/p2012/FCS2552.pdf.
23. Kruskal-based approximation algorithm for the multi-level Steiner tree problem. URL: https://arxiv.org/pdf/2002.06421v2.pdf.
24. Жадібні алгоритми. URL: https://devzone.org.ua/post/zhadibni-algoritmi.
25. Коэволюционные алгоритмы для решения иерархических задач дерева Штейнера в телекоммуникационных сетях. URL: https://www.sciencedirect.com/science/article/abs/pii/S1568494619304995

ДОДАТОК А

Код модуля Steiner Point Calculation

```
import tkinter as tk
from tkinter import messagebox
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import numpy as np
from scipy.optimize import minimize

def calculate_steiner_point(p1, p2, p3):
    # Функція для обчислення суми відстаней від точки до вершин трикутника
    def distance_sum(point, p1, p2, p3):
        return (
            np.linalg.norm(np.array(point) - np.array(p1)) +
            np.linalg.norm(np.array(point) - np.array(p2)) +
            np.linalg.norm(np.array(point) - np.array(p3))
        )
    # Вибір початкової точки для оптимізації (можна використовувати центроїд)
    initial_point = [(p1[0] + p2[0] + p3[0]) / 3, (p1[1] + p2[1] + p3[1]) / 3]
    # Використання методу оптимізації для знаходження точки Штейнера
    result = minimize(distance_sum, initial_point, args=(p1, p2, p3), method='Nelder-Mead')

    if result.success:
        return result.x
    else:
        raise ValueError("Optimization failed to find the Steiner point")

def on_calculate():
    try:
        p1 = tuple(map(float, entry_p1.get().split()))
        p2 = tuple(map(float, entry_p2.get().split()))
        p3 = tuple(map(float, entry_p3.get().split()))
        if len(p1) != 2 or len(p2) != 2 or len(p3) != 2:
            raise ValueError("Each point must contain two coordinates (x and y).")
        steiner_point = calculate_steiner_point(p1, p2, p3)
        plot_graph(p1, p2, p3, steiner_point)
        label_result.config(text=f"Steiner Point Coordinates: {steiner_point}")
    except ValueError as e:
        messagebox.showerror("Error", str(e))
def plot_graph(p1, p2, p3, steiner_point):
```

```
    fig.clear()
    ax = fig.add_subplot(111)
    ax.scatter(*zip(p1, p2, p3), c='red', label='Input Points')
    if steiner_point is not None:
        ax.scatter(*steiner_point, c='blue', label='Steiner Point')
        ax.plot([p1[0], steiner_point[0]], [p1[1], steiner_point[1]], 'k--')
        ax.plot([p2[0], steiner_point[0]], [p2[1], steiner_point[1]], 'k--')
        ax.plot([p3[0], steiner_point[0]], [p3[1], steiner_point[1]], 'k--')
    ax.plot([p1[0], p2[0], p3[0], p1[0]], [p1[1], p2[1], p3[1], p1[1]], 'k--')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_title('Steiner Point Calculation')
    ax.legend()
    canvas.draw()

root = tk.Tk()
root.title("Steiner Point Calculator")

# Input fields for coordinates
tk.Label(root, text="Point 1 (x y):").pack()
entry_p1 = tk.Entry(root)
entry_p1.pack()
tk.Label(root, text="Point 2 (x y):").pack()
entry_p2 = tk.Entry(root)
entry_p2.pack()
tk.Label(root, text="Point 3 (x y):").pack()
entry_p3 = tk.Entry(root)
entry_p3.pack()
# Calculate button
button_calculate = tk.Button(root, text="Calculate", command=on_calculate)
button_calculate.pack()
# Result label
label_result = tk.Label(root, text="")
label_result.pack()
# Graph
fig, ax = plt.subplots()
canvas = FigureCanvasTkAgg(fig, master=root)
canvas.get_tk_widget().pack()

root.mainloop()
```

ДОДАТОК Б

Код модуля EMST

```
import tkinter as tk
from tkinter import ttk
import numpy as np
import time
from deap import base, creator, tools, algorithms
from scipy.spatial import distance_matrix
from scipy.sparse.csgraph import minimum_spanning_tree
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import random

# Ініціалізація DEAP
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

# Основний алгоритм
def main(coordinates, crossover_prob, mutation_prob, generations,
population_size, ax):
    dist_matrix = distance_matrix(coordinates, coordinates)

    toolbox = base.Toolbox()
    toolbox.register("indices", random.sample, range(len(coordinates)),
len(coordinates))
    toolbox.register("individual", tools.initIterate, creator.Individual,
toolbox.indices)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)

    def evalMST(individual):
        mst_matrix = minimum_spanning_tree(dist_matrix[individual][:, individual])
        total_length = mst_matrix.sum()
        return (total_length,)

    toolbox.register("evaluate", evalMST)
```

```
    toolbox.register("mate", tools.cxOrdered)
    toolbox.register("mutate", tools.mutShuffleIndexes, indpb=mutation_prob)
    toolbox.register("select", tools.selTournament, tournsize=3)

    pop = toolbox.population(n=population_size)
    hof = tools.HallOfFame(1, similar=np.array_equal)

    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean)
    stats.register("min", np.min)
    stats.register("max", np.max)

    start_time = time.time()
    pop, log = algorithms.eaSimple(pop, toolbox, crossover_prob, mutation_prob,
generations, stats=stats, halloffame=hof, verbose=True)
    end_time = time.time()
    execution_time = end_time - start_time

    best_ind = hof[0]
    best_mst_length = evalMST(best_ind)[0]
    print(f"Best individual is {best_ind} with fitness {best_ind.fitness.values} and
MST length {best_mst_length}")

    # Візуалізація найкращого рішення
    ax.clear()
    ax.scatter(coordinates[:, 0], coordinates[:, 1], c='b')  # Всі точки
    mst_matrix = minimum_spanning_tree(dist_matrix).toarray()
    for i in range(len(coordinates)):
        for j in range(len(coordinates)):
            if mst_matrix[i][j] != 0:
                ax.plot([coordinates[i][0], coordinates[j][0]], [coordinates[i][1],
coordinates[j][1]], 'r-')

    return execution_time, best_mst_length

# Функція запуску з GUI
def run():
```

```
    coord_input = coord_entry.get()
    coordinates = np.array([list(map(float, point.split(','))) for point in
coord_input.split(';')])
    crossover_prob = float(crossover_prob_entry.get())
    mutation_prob = float(mutation_prob_entry.get())
    generations = int(generations_entry.get())
    population_size = int(population_size_entry.get())

    # Виконання генетичного алгоритму і візуалізація результатів
    execution_time, best_mst_length = main(coordinates, crossover_prob,
mutation_prob, generations, population_size, ax)
    execution_time_label.config(text=f"Час виконання: {execution_time:.2f}
секунд, Довжина MST: {best_mst_length}")
    canvas.draw()

# Налаштування головного вікна Tkinter
root = tk.Tk()
root.title("Генетичний Алгоритм GUI")

# Дозволяємо масштабування вікна
root.grid_columnconfigure(0, weight=1)
root.grid_columnconfigure(1, weight=1)
root.grid_rowconfigure(0, weight=1)
root.grid_rowconfigure(1, weight=1)
root.grid_rowconfigure(2, weight=1)
root.grid_rowconfigure(3, weight=1)
root.grid_rowconfigure(4, weight=1)
root.grid_rowconfigure(5, weight=1)
root.grid_rowconfigure(6, weight=1)

# Поля вводу для користувача
coord_label = tk.Label(root, text="Координати (x,y; x,y; ...):")
coord_label.grid(row=0, column=0, sticky='ew')
coord_entry = tk.Entry(root)
coord_entry.grid(row=0, column=1, sticky='ew')

crossover_prob_label = tk.Label(root, text="Ймовірність схрещування:")
```

```
crossover_prob_label.grid(row=1, column=0, sticky='ew')
crossover_prob_entry = tk.Entry(root)
crossover_prob_entry.grid(row=1, column=1, sticky='ew')

mutation_prob_label = tk.Label(root, text="Ймовірність мутації:")
mutation_prob_label.grid(row=2, column=0, sticky='ew')
mutation_prob_entry = tk.Entry(root)
mutation_prob_entry.grid(row=2, column=1, sticky='ew')

generations_label = tk.Label(root, text="Кількість поколінь:")
generations_label.grid(row=3, column=0, sticky='ew')
generations_entry = tk.Entry(root)
generations_entry.grid(row=3, column=1, sticky='ew')

population_size_label = tk.Label(root, text="Розмір популяції:")
population_size_label.grid(row=4, column=0, sticky='ew')
population_size_entry = tk.Entry(root)
population_size_entry.grid(row=4, column=1, sticky='ew')

run_button = tk.Button(root, text="Запустити", command=run)
run_button.grid(row=5, column=0, columnspan=2, sticky='ew')

# Підготовка місця для графіка та часу виконання
fig, ax = plt.subplots()
canvas = FigureCanvasTkAgg(fig, master=root)
canvas.get_tk_widget().grid(row=6, column=0, columnspan=2, sticky='ew')
execution_time_label = ttk.Label(root, text="")
execution_time_label.grid(row=7, column=0, columnspan=2, sticky='ew')

root.mainloop()
```

Міністерство освіти і науки України

Міністерство інфраструктури України

Український державний університет науки та технологій

Східний науковий центр транспортної академії наук

TEMPUS: CITISET & SEREIN & CRENG

ТЕЗИ
XVII Міжнародної науково-практичної конференції
«СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ ТА ОСВІТІ»

ABSTRACTS
of the XVII International Conference
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»

13.12.2023 – 14.12.2023

Дніпро
2023

Використання генетичного алгоритму для пошуку точки Штейнера на площині за допомогою кластеризації області пошуку

Глушков О.В., Український державний університет науки і технологій, Україна

Метод Штейнера має безліч застосувань у різних галузях, включаючи телекомунікації (для з'єднання мереж з мінімальною довжиною кабелів), маршрутизацію мереж, виробництво печатних плат, дизайн логістичних систем і багато інших областей. Основна мета задачі Штейнера - знайти таке дерево Штейнера, яке з'єднує всі термінали і стейнерові вершини, мінімізуючи сумарну довжину ребер. Ця задача є комбінаторною оптимізаційною задачею і може бути дуже складною для розв'язання, особливо на великих графах. Тому існують різні алгоритми, які намагаються наблизити оптимальне рішення або знайти розв'язок в прийнятний час.

Одним з методів вирішення задачі Штейнера є генетичний алгоритм. Він є алгоритмом спрямованого пошуку і на відміну від алгоритму повного перебору, що розглядає всі можливі варіанти рішення, алгоритм спрямованого пошуку прагне знайти околицю оптимального рішення. Отже, рішення займе менше часу, ніж у разі перебору варіантів потенційних рішень. Однак, практика використання генетичних алгоритмів для вирішення NP-повних завдань має безліч труднощів, що виникають внаслідок прагнення алгоритму спрямованого пошуку до локально оптимального рішення. Ця проблема випливає з відсутності інформації у алгоритму локальності або глобальності виявленого оптимуму, через що алгоритми спрямованого пошуку не гарантують перебування глобально оптимального рішення.

Для спрощення вирішення задачі Штейнера пропонується розділити площу з початковими точками на частини з приблизно однаковою кількістю вершин и застосувати алгоритми для кожної групи окремо. В результаті отримаємо по одній точці в т.н. кластері. Потім обробити всі нові точки і отримати більш менш точний кінцевий результат.

Розбивка великої кількості точок на кластери має декілька переваг, особливо коли маємо справу зі складними задачами:

- Зменшення складності задачі: Коли велика кількість точок розбивається на кластери, завдання різноманітних обчислень стає менш складним. Кластери групують подібні точки, дозволяючи вам працювати з меншими наборами даних.

- Підвищення швидкодії алгоритму: Працюючи з кластерами, алгоритми можуть бути ефективнішими за рахунок розподілення обчислень на окремі частини даних. Це може покращити час виконання завдання.

- Зменшення вимог до ресурсів: Обробка кластерів дозволяє економити ресурси (такі як пам'ять і обчислювальну потужність), оскільки працюємо з меншим обсягом даних, що може бути важливо для обробки великих обсягів інформації.

- Легше виявлення шаблонів та особливостей: Кластеризація дозволяє виявляти загальні тенденції та особливості в групах точок, спрощуючи аналіз та виявлення шаблонів в даних.

- Підвищення робастності: Використання кластерів може зробити алгоритм менш чутливим до шуму або випадкових аномалій, оскільки обробка відбувається на рівні кластерів, а не окремих точок.

У загальному, розбивка точок на кластери допомагає спростити складні задачі, зменшити обсяг даних для обробки, покращити швидкодію алгоритму та полегшити аналіз даних.

Головним результатом дослідження є збільшення ефективності пошуку точок Штейнера на площині за допомогою використання генетичного алгоритму та методу кластеризації. Отримані результати демонструють покращення швидкості пошуку та точності визначення оптимальних точок Штейнера.