КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота

на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

АЛГОРИТМИ РОЗВ'ЯЗАННЯ ЗАДАЧІ ПРО МІНІМАЛЬНИЙ ЧАС ПЕРЕДАЧІ ПОВІДОМЛЕННЯ

Виконала студентка 4-го курсу
Сокол Наталія Віталіївна _____
(підпис)

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Ставровський Андрій Борисович _____
(підпис)

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

Студент _____
(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри теоретичної кібернетики
«____» _____ 202_ р.,
протокол № ___
Завідувач кафедри
Ю. В. Крак _____
(підпис

Київ – 2021

РЕФЕРАТ

Обсяг роботи 36 сторінок, 16 ілюстрацій, 8 джерел посилань.

МІНІМАЛЬНИЙ ЧАС ПЕРЕДАЧІ ПОВІДОМЛЕННЯ, МІНІМАЛЬНИЙ БРОДКАСТ-ГРАФ, АЛГОРИТМИ НА ГРАФАХ, ЕВРИСТИЧНІ АЛГОРИТМИ, NP-ПОВНІ ЗАДАЧІ, ГЕНЕТИЧНИЙ АЛГОРИТМ.

Об’єктом роботи є процес розв’язування задачі про мінімальний час передачі повідомлення, а також розробка тестової системи і дослідження генетичного алгоритму для розв’язання задачі за допомогою створеної тестової системи. Предметом роботи є програмний засіб «Тестова система для дослідження роботи генетичного алгоритму для задачі про мінімальний час передачі повідомлення», що реалізує роботу генетичного алгоритму для розв’язання задачі про мінімальний час передачі повідомлення.

Метою роботи є створення програмного засобу для тестування роботи генетичного алгоритму для розв'язання задачі про мінімальний час передачі повідомлення.

Методи розроблення: комп’ютерне моделювання, евристичні алгоритми розв’язання задачі про мінімальний час передачі повідомлення. Інструменти розроблення: безкоштовне, вільно поширюване інтегроване середовище розробки Visual Studio 2019, мова програмування C++.

Результати роботи: було виконано загальний огляд задачі про мінімальний час передачі повідомлення та існуючих алгоритмів для її розв’язання. В роботі було досліджено роботу генетичного алгоритму для розв’язання цієї задачі, а саме вплив параметрів алгоритму, таких як коефіцієнт мутацій та розмір початкової популяції.

Програмний продукт «Тестова система для дослідження роботи генетичного алгоритму для задачі про мінімальний час передачі

повідомлення» може застосовуватися для розв’язання задач широкомовної передачі даних та дослідження роботи генетичного алгоритму для даної задачі.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

P — клас задач, які мають поліноміальні алгоритми розв'язання;

NP — клас задач, які мають недетерміновані поліноміальні алгоритми розв'язання;

ГА — генетичний алгоритм.

ВСТУП

Оцінка сучасного стану об’єкту дослідження. Завданням роботи є дослідити задачу про мінімальний час передачі повідомлення. Задача трансляції інформації має широке застосування в сучасному світі - від того, яким алгоритмом буде знайдено оптимальний шлях передачі пакетів повідомлення, буде залежати наскільки швидко користувач отримає потрібні йому дані. При однаковій кількості ресурсів вибір тієї чи іншої мережі для передачі інформації може суттєво вплинути на ефективність роботи системи. В загальному випадку задача на сьогоднішній день не має ефективного алгоритму розв’язку, але окремі її випадки активно досліджуються і отримані результати знаходять застосування незважаючи на те, що загальна задача не має ефективного алгоритму. Наприклад, у 2001 році вийшла стаття [1], в якій розглядається випадок графу з 26 вершинами. У статті стверджується і доводиться якою має бути мережа для 26 точок, що передають інформацію, щоб передача відбувалась якнайшвидше.

Актуальність роботи та підстави для її виконання. Задача про мінімальний час передачі повідомлення має ефективні алгоритми розв’язання тільки для деяких окремих випадків з певними значеннями її параметрів. Задача залишається актуальною і потребує подальшого дослідження окремих її випадків або доведення того, що загальна задача має або не має ефективного алгоритму розв’язання.

Мета і завдання роботи. Дослідити існуючі алгоритми розв’язання окремих випадків загальної задачі, в тому числі генетичний алгоритм для задачі про мінімальний час передачі повідомлення; розробити систему тестування для генетичного алгоритму та дослідити як впливають параметри,

такі як коефіцієнт мутації та розмір популяції, на швидкість роботи алгоритму та ефективність отриманих результатів.

Об’єкт, методи і засоби дослідження. Об’єктом дослідження є задача про мінімальний час передачі повідомлення - відома NP-повна задача, що на даний момент не має ефективного поліноміального алгоритму розв’язання в загальному випадку. В процесі дослідження необхідно оглянути існуючі алгоритми, змоделювати їх роботу на простих прикладах. Для генетичного алгоритму методом дослідження є тестова система, яку необхідно розробити.

Для розробки тестової системи було обрано мову C++ та середовище розробки Visual Studio 2019. Вибір мови програмування дозволяє ефективніше використовувати обчислювальні ресурси і подавати на вхід тестовій системі великі об’єми вхідних даних, що збільшить точність проведених тестів.

Можливі сфери застосування. Результати дослідження можуть застосовуватися при розв’язанні практичних задач, пов’язаних з мінімальним часом передачі повідомлення. В статті [2] описано можливі практичні застосування, такі як телекомунікаційні технології, або з більш сучасних проблем: мережі розповсюдження контенту (CDN), одноранговi мережі (Peer-to-Peer) або мережі когнітивного радіо. Результати досліджень допоможуть обрати правильні параметри для генетичного алгоритму і отримати ефективні результати обчислень за мінімально можливий час для даного алгоритму.

РОЗДІЛ 1. ФОРМАЛІЗАЦІЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

1.1 Постановка задачі.

Розглянемо граф G = (V, E). Граф представляє собою модель мережі, де кожна з вершин є точкою, яка може передавати та отримувати інформацію, а ребра графу - це канали зв'язку між цими точками. В графі задано підмножину його вершин V0 (так званих інформаторів), яким відомо початкове повідомлення, що потрібно поширити на всю мережу. Кожна вершина може передати лише одне повідомлення за одиницю часу. Потрібно дізнатися, чи можливо за вказаний час K поширити повідомлення на всю мережу. Задачу вперше сформульовано у книжці [3]. Задачі такого типу, з відповіддю "так " або "ні", називаються задачами розпізнавання. Більшість задач можна переписати у вигляді задачі розпізнавання, що спрощує подальше їх вивчення та класифікацію.

На практичних прикладах ця задача має кілька близьких питань, що набагато частіше знаходять застосування. Наприклад, якщо відійти від обмеження на однозначну відповідь "так " або "ні", питання можна поставити таким чином: маємо граф з підмножиною вершин, яким відоме повідомлення. Потрібно знайти найшвидший спосіб передати повідомлення до всіх вершин графу. Також можна відійти від фіксованого заданого графу і поставити запитання таким чином - як потрібно поєднати точки передачі інформації, щоб витратити найменше ресурсів і мати можливість розповсюджувати інформацію з будь-якої початкової точки якнайшвидше. Остання задача розглядається частіше при побудові мереж, коли ми маємо можливість створити граф для передачі, і потрібно дізнатися, яким саме він має бути, щоб максимально швидко поширювати повідомлення, на відміну від задачі, коли

ми працюємо з готовим графом. Всі ці задачі близько пов’язані і в першу чергу їх об’єднує те, що їх дуже складно розв’язати.

Задача передачі повідомлення належить до класу задач, що не мають поліноміального алгоритму розв’язання у загальному випадку.

Розглянемо детальніше класифікацію задач. Існує два великих класи задач - P та NP. До класу P відносяться задачі, для яких існує алгоритм, що знаходить відповідь за час обмежений зверху певним поліномом, що залежить від входу задачі. Якщо ж такий алгоритм невідомий, то ми маємо справу з більш складною задачею, яка потребує інших підходів для її розв’язання. Як варіант, ми можемо припустити деяку відповідь, і перевірити чи справді вона розв’язує поставлену задачу. Для перевірки відповідей знайдемо алгоритм для перевірки вгаданої відповіді за поліноміальний час. Якщо такий алгоритм існує, то ми маємо справу з представником класу NP.

1.2 Формалізація класів задач.

Для формалізації понять звернемось до машини Тюрінга (рис. 1). Машина Тюрінга - це пристрій, що має нескінчену кількість упорядкованих комірок, набір станів та певних правил переходу. Вона працює на певному алфавіті - наборі символів, і вміє розпізнавати певні слова цього алфавіту. В множині станів маємо кілька заключних станів, перехід в які зупиняє роботу програми.

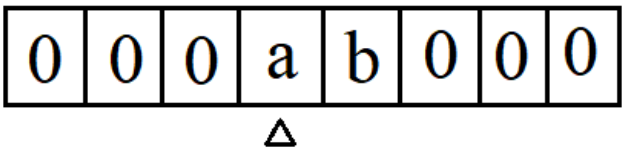

Рисунок 1: Схематичне зображення машини Тюрінга

Звернемо увагу на поняття алфавіту. Алфавіт - це набір символів, який включає спеціальний порожній символ. На початку роботи всі комірки, окрім тих, де записано вхідне слово, заповнені порожнім символом. Будь-яка комбінація непорожніх символів є словом цього алфавіту. А будь-яка підмножина слів алфавіту є мовою. Машина Тюрінга розпізнає мову, якщо для кожного її слова, вона закінчує свою роботу в приймаючому заключному стані.

Кожній задачі розпізнавання за допомогою кодування можна поставити у відповідність еквівалентну їй мову. Машина Тюрінга для задачі розпізнавання має два заключних стани - стан “так” і стан “ні”. Очевидно, що тоді розв’язання задачі зводиться до пошуку програми для машини Тюрінга, яка закінчить свою роботу в приймаючому стані “так” для всіх слів мови, і тільки для них.

Повернемось до класів задач - якщо ця програма завершується для всіх слів мови за кількість кроків, що обмежена зверху певним поліномом, то цю задачу можна віднести до класу P.

В іншому випадку звернемось до недетермінованої машини Тюрінга. Вона відрізняється від детермінованої тим, що перед початком програми в певні комірки записується “здогадка” - випадкове слово з алфавіту, а далі робота машини продовжується звичайним чином. При цьому потрібно відмітити, що при одному вхідному слові програма може виконати безліч різних обчислень з різними результатами. Програма приймає це слово, якщо хоча б одне з обчислень завершується в приймаючому стані. Це є еквівалентом вгадування відповіді на питання задачі і подальша її перевірка.

Час роботи недетермінованої машини Тюрінга - це кількість кроків від початку вгадування до моменту, коли машина прийде в заключний стан “так”.

При цьому береться мінімальна кількість кроків по всім приймаючим обчисленням. Якщо для задачі існує програма для недетермінованої машини Тюрінга, що працює за поліноміальний час, то задача відноситься до класу NP задач.

РОЗДІЛ 2. ОГЛЯД ІСНУЮЧИХ АЛГОРИТМІВ

2.1 NP-повнота та досліджені випадки

Задача про мінімальний час передачі повідомлення є відомою NP-повною задачею, що була описана в книжці М. Гері та Д. Джонсона “Computers and Intractability” [3]. Тому як поліноміального алгоритму розв’язання досі не існує, було розроблено ряд евристичних алгоритмів для цієї задачі, які хоча і не завжди дають оптимальну відповідь, на практиці є дуже корисними і мають широке застосування в різних сферах, такі як комп’ютерні мережі або телекомунікаційні технології. В роботі [4] доводиться NP-повнота для деяких окремих сімейств графів з певними параметрами задачі (час, за який потрібно передати повідомлення та кількість інформованих вершин в момент часу 0). Для доведення автори зводять інші відомі NP-повні задачі до кожного окремого розглянутого випадку задачі про мінімальний час. Так, наприклад, в статті розглядаються такі випадки:

- двочастковий планарний граф з максимальним степенем вершин 3 та параметром задачі K = 2
- розщеплювані графи для K = 2
- планарні графи з максимальним степенем вершин 3 та для кількості інформаторів 1
- графи решітки з максимальним степенем 4 та для кількості інформаторів 1

Для окремих випадків задач також було розроблено алгоритми, що дають точну відповідь. Наприклад, якщо значення часу K дорівнює 1, то задача вирішувана за поліноміальний час. Очевидно, щоб розв’язок такої задачі існував, достатньо щоб граф G мав двочастковий підграф, одна частка якого

збігається з множиною інформаторів V_0. Також в роботі [5] описано алгоритм, що розв'язує задачу для дерев з довільною кількістю інформаторів, за поліноміальний час.

Очевидно, що для повного графу час передачі повідомлення буде становити $\lceil \log_2 n \rceil$, але при цьому не всі ребра будуть використані під час широкомовної передачі даних. Постає питання, яку максимальну кількість ребер ми можемо видалити так, щоб не програти в часі початковому графу. Цю задачу називають "minimum broadcast graph" і вона розглядається у [6]. В статті мінімальну можливу кількість ребер в графі з n вершинами позначають як B(n) і називають функцією бродкасту. Автори вказують значення B(n) для всіх $n \leq 15$ та наводять приклад побудови відповідного графу. Також в статті досліджуються графи з $n = 2^k$ та доводиться, що функція бродкасту для таких графів дорівнює: $B(2^k) = k \cdot 2^{k-1}$.

2.2 Ітеративний алгоритм О. Арутюняна та К. Моросяна

У 2004 році вийшла стаття двох канадських вчених, в якій було представлено новий евристичний алгоритм для знаходження мінімального часу передачі повідомлення [7]. Алгоритм знаходить приблизний мінімальний час для кожної з вершин, і з кожною ітерацією намагається збільшити точність відповіді. Час роботи алгоритму - $O(knm)$, де k - кількість ітерацій.

Суть алгоритму полягає в тому, що на початку для кожної вершини ми припускаємо час передачі, який дорівнює кількості вершин мінус один (очевидно, що найгірший можливий випадок - це передача з вершини 0 в графі на рис. 2, і час такої передачі дорівнює $n-1$). Далі ми починаємо першу ітерацію, під час якої проходимо по списку всіх інформованих вершин і обираємо для них непроінформованого сусіда зі списку суміжних вершин.

Обирається завжди вершина з найменшим часом передачі. Очевидно, що на початку роботи алгоритму оцінка часу для всіх вершин є однаковою, тому в такому випадку наступну вершину можна обирати випадковим чином, до того моменту, коли оцінка часу для якоїсь із них не зменшиться. Коли проінформовані вершини закінчились, додаємо до їх списку вершин, що були проінформовані тільки що, і вилучаємо ті, які більше не мають непроінформованих сусідів. Коли проінформовані всі вершини, перевіряємо скільки часу це зайняло і запам’ятовуємо цей час, щоб використати його при виборі сусідів у наступних ітераціях.

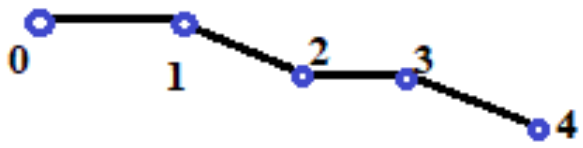

Рисунок 2: Передача з 0 в 4 відбувається за 3 одиниці часу

Кількість ітерацій в цьому алгоритмі - це незалежна змінна, яка відповідає за точність відповіді. Шляхом експерименту було виявлено, що для більшості графів під час перших $log\ n$ ітерацій оптимальність отриманої відповіді стрімко збільшується, а подальші обрахунки приносять незначні покращення. Також для деяких графів, таких як наприклад повні бінарні дерева, алгоритму достатньо лише однієї ітерації, щоб знайти оптимальну відповідь.

Розглянемо приклад реалізації алгоритму мовою C++. Зовнішній цикл визначається заданною кількістю ітерацій — MAX_ITER. Далі для кожної вершини графу проводимо ініціалізацію (встановлюємо нульовий час та множину інформованих вершин, якій належить тільки поточна вершина, а також копіюємо списки суміжності, щоб модифікувати їх під час роботи алгоритму).

```
28  void Graph::algorithm(int MAX_ITER)
29  {
30      for (int i = 0; i < MAX_ITER; ++i)
31      {
32          for (int v = 0; v < adj_lists.size(); v++)
33          {
34              std::vector<std::set<int>> adj_lists_for_v = adj_lists;
35              std::set<int> S = { v };
36              int informed_vertices = 1;
37              int b_time = 0;
```

Рисунок 3: Ініціалізація ітеративного алгоритму

Після цього запускаємо внутрішній цикл, умова зупинки якого — всі вершини проінформовані (рис. 4). Всередині ми працюємо з множиною S – це множина всіх вершин, які будуть корисними на наступному кроці.

В кінці внутрішнього циклу перевіряємо чи зможе щойно проінформована вершина передати комусь повідомлення, і якщо так, додаєм її до множини S. Також перевіряємо, чи має тільки що оброблена вершина ще не проінформованих сусідів, і якщо ні, то видаляємо її з множини S.

```
// знаходимо суміжну вершину з найменшим часом
int best_neighbour = *adj_lists_for_v[u].begin();
for (int neighbour : adj_lists_for_v[u])
    if (min_br_times[neighbour] < min_br_times[best_neighbour])
        best_neighbour = neighbour;

// видаляємо ребро зі списків суміжності
adj_lists_for_v[best_neighbour].erase(u);
adj_lists_for_v[u].erase(best_neighbour);

if (!adj_lists_for_v[best_neighbour].empty())
    NI.insert(best_neighbour);

if (adj_lists_for_v[u].empty())
    S.erase(u);
```

Рисунок 4: Обробка вершин в ітеративному алгоритмі

Розглянемо принцип роботи алгоритму на простому прикладі. Візьмемо граф G з 5 вершинами і звичайним перебором обчислимо значення мінімального часу передачі повідомлення для кожної вершини графу *b(v)* (рис. 5). Використаємо їх як контрольні значення, щоб порівняти результати роботи алгоритму з ними.

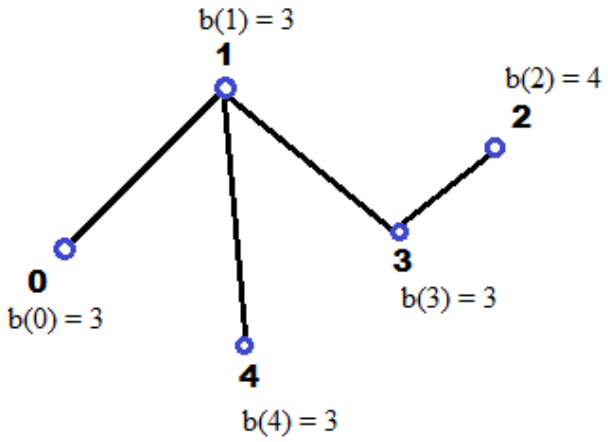

Рисунок 5: Приклад графу для демонстрації роботи ітеративного алгоритму

Для цього графу побудуємо списки суміжності вершин (рис. 6).

Також запишемо для кожної вершини початкову оцінку часу *b(v)*. Кожна вершина графу має почергово виступати початковим інформатором. Почнемо з вершини 0.

0: {1}	**b(0) = 4**
1: {3, 4, 0}	**b(1) = 4**
2: {3}	**b(2) = 4**
3: {1, 2}	**b(3) = 4**
4: {1}	**b(4) = 4**

Рисунок 6: Списки суміжності

S = {0} - список проінформованих вершин. Час витрачено 0 одиниць. Вершина 0 має єдиного сусіда 1, тому обираємо її. Більше суміжних вершин немає, тому ця внутрішня ітерація завершується і тепер потрібно модифікувати множину S. Перевіряємо, чи залишились у вершини 0 інші сусіди. Їх немає, тому вилучаємо її з множини S. Далі перевіряємо, котрі з нових проінформованих вершин ми можемо додати. Додаємо вершину 1, тому що вона має інших непроінформованих сусідів. Таким чином, сенс модифікації множини S полягає в тому, що там повинні знаходитись лише “корисні” вершини - якщо вершина вже передала повідомлення всім, кому могла - ми її вилучаємо і не витрачаємо на неї час на наступному проході. При цьому з нових вершин ми додаємо лише ті, які зможуть принести користь під час наступного проходу по S. Таким чином, маємо тепер **S = {1}**. Відмітимо також, що було витрачено одиницю часу для цих дій. Вершина 1 має двох сусідів - 3 та 4 (вершину 0 ми

не враховуємо, тому що вона вже була проінформована). Оцінка часу на цьому етапі для них обох однакова, тому оберемо за порядком запису вершину 3. Як і в попередньому проході, модифікуємо множину S: **S = {1, 3}**. На даний момент витрачено дві одиниці часу. Легко побачити, що в наступному проході вершина 1 проінформує 4, а 3 проінформує 2. На цьому підрахунок часу для першої вершини графу закінчується і ми записуємо нову оцінку часу: b(0) = 3. Завдяки цьому для наступної вершини вибір сусіда буде відбуватися обгрунтовано.

Коротко опишемо процес для інформатора 1. **S = {1}**, вершина 1 має сусідів 0, 3 та 4 і серед них ми обираємо вершину 0, яка має найменше значення b(0). Далі **S = {1}** (0 не додається, тому що не має більше сусідів), інформуємо 3. S = {1, 3}, інформуємо 4 та 2. Часу витрачено: 3, записуємо нову оцінку.

Для інформатора 2 вибір вершини буде відбуватися лише в самому кінці між 0 та 4, і на час це ніяк не вплине. Залишається b(2) = 4.

Тепер звернемо увагу на наступний прохід для інформатора 3. На самому початку маємо два сусіда 1 та 2, обидва з яких вже було пройдено першою ітерацією і отримано перші результати. b(1) < b(2), тому обираємо першу вершину. Нескладно помітити, що завдяки цьому вибору передача відбудеться швидше, ніж якби ми обрали вершину 2. Тепер 3 інформує 2 (3 -> 2), і 1 -> 0, і в кінці 1 -> 4. Таким чином, вже на першій ітерації алгоритм обрав кращу вершину і за рахунок цього уточнив оцінку b(3) = 3.

Для останньої вершини 4 маємо: 4 -> 1; 1 -> 3; 1 -> 0, 3 -> 2 (витрачено три одиниці часу). Якщо порівняти результати з попередньо обрахованими, побачимо, що однієї ітерації вистачило для знаходження оптимальної відповіді. Але необхідно відмітити, що велику роль в цьому випадку відіграв випадково вдало підібраний порядок вершин. Наприклад, якби для

інформатора 4 було обрано порядок 4 -> 1; 1 ->0; 1 -> 3; 3 -> 2, то в такому випадку ми б отримали час b(4) = 4, а це не є оптимальною відповіддю. Для вершин 0 та 3 оцінка отримана вірно, і вона однакова, тому подальші ітерації не допомогли б на цьому кроці зробити правильний вибір.

Тестування алгоритму показало, що він є найбільш ефективним для нерегулярних графів, у яких значення b(v) мають великий розкид поміж собою. Для регулярних графів немає сенсу застосовувати цей алгоритм, тому як на деякому кроці значення b(v) стануть майже однаковими і вибір сусідів буде залежати лише від порядку їх запису. Таким чином, в цьому випадку обчислення не буде відрізнятися від алгоритму з випадковим вибором вершин.

2.3 Генетичний алгоритм

Для розв’язання задачі також було запропоновано генетичний алгоритм, що використовує глобальний вектор пріоритетів. У своїй статті [8] автори визначають функцію оцінки для довільного елементу популяції, та спосіб схрещування двох елементів.

Генетичні алгоритми - це евристичні алгоритми, що моделюють природне явище еволюції і таким чином, знаходять оптимальну відповідь до поставленої задачі. Суть обчислень полягає в тому, що ми моделюємо процес еволюції популяції і наприкінці обираємо найкращий існуючий елемент популяції. В нашому випадку головними операторами алгоритму є оператор схрещування та функція оцінки елементу популяції. Від цих операторів залежить робота всього алгоритму, а саме “в який бік” буде еволюціонувати популяція. Розглянемо детальніше суть алгоритму.

Елемент популяції - це перестановка вершин даної мережі, відповідно до якої відбувається передача повідомлень. З цієї перестановки утворюємо дві множини: V0 — інформовані вершини, та V\V0 - решта вершин. Важливо

зберігати порядок V0 таким, як в початковій перестановці, а в V\V0 — порядок, зворотній даному в перестановці. Сенс заключається в тому, що передача повідомлення буде відбуватися ефективніше, якщо починати передавати з вершин, що мають найменше сусідів, у вершини, що мають найбільше сусідів, з ціллю подальшого розповсюдження повідомлення. В потенційного “гарного” елемента популяції на початку будуть йти вершини з найменшим степенем, і поступово збільшуватися. Саме з цієї причини ми підтримуємо зворотній порядок у множині V\V0. Далі, щоб визначити схему передачі по даній перестановці, починаємо з першої вершини із V0 і шукаємо у впорядкованій множині V\V0 першу суміжну їй вершину. Так для кожного елементу V0 по можливості інформуються інші вершини і після цього оцінка часу для цієї схеми збільшується на один. Повторюємо дії поки множина V\V0 не залишиться порожньою. Таким чином ми оцінюємо осіб популяції для знаходження найкращих.

Для операції схрещування нам знадобиться глобальний вектор пріоритетів. Для його створення потрібно обчислити степені всіх вершин графу і відсортувати список вершин за зростанням степеня. Операція схрещування відбувається таким чином: беремо дві перестановки, які будуть виступати батьками. Для знаходження елементу дочірньої особини, ми беремо два відповідних елементи у батьків і віддаємо перевагу тому, що йде першим у векторі пріоритетів. Після цього необхідно локально модифікувати батьківську особину з програвшим геном, а саме поміняти місцями два елементи перестановки так, щоб на місці програвшого гену стояла вершина, що перемогла.

В своїй статті [6] автори детально описують операцію схрещування двох хромосом та функцію оцінки хромосоми, що однозначно обраховує час

передачі повідомлення для кожної унікальної хромосоми. Всі інші параметри для генетичного алгоритму потрібно дослідити експериментальним шляхом на власній тестовій системі, розробка якої описана в наступному розділі.

РОЗДІЛ 3. РОЗРОБКА ТЕСТОВОЇ СИСТЕМИ

3.1 Методи розробки системи

Для розробки тестової системи генетичного алгоритму було обрано мову програмування C++ та середовище розробки Visual Studio 2019. Вибір мови зумовлений тим, що генетичні алгоритми потребують значних обчислювальних ресурсів і можливість максимально ефективно використовувати обчислювальну потужність власного комп'ютера стає у нагоді.

При розробці програми активно використовуються можливості стандартної бібліотеки C++ для таких задач, як сортування масивів або генерації перестановок. Реалізація алгоритмів в стандарній бібліотеці дозволяє використовувати їх для розв'язання подібних задач з великими об'ємами вхідних даних.

```
5   class GenAlg
6   {
7   public:
8       void InitializeGraph(const char* filename);
9       void Populate(int size);
10      void StartEvolution(int& generations);
11      void SortByFitnessFunction();
12      void FillPrecedenceVector();
13      void Mutate(std::vector<int>& chromosome);
14      void SetOriginators(std::vector<int> _originators) { originators = _originators; }
15      void SetRandomOriginator();
16      int  FitnessFunction(const std::vector<int>& chromosome, bool print = false);
17      std::vector<int> Crossover(std::vector<int> a, std::vector<int> b);
18
19  private:
20      std::vector<std::vector<int>> population;
21      //graph
22      std::vector<std::vector<int>> graph;
23      std::vector<int> originators;
24      std::vector<int> precedenceVector;
25      std::list<int> last_fitness;
```

Рисунок 7: Оголошення класу

3.2 Реалізація

Розглянемо оголошення класу. Публічні функції на рис. 7 відповідають за процес ініціалізації та роботи генетичного алгоритму.

Інтерфейс класу, що виконує основні обчислення, є досить простим та інтуїтивно зрозумілим. Маємо набір функцій для ініціалізації алгоритму, такі як:

- InitializeGraph(const char* filename) - ініціалізує граф, для якого буде розв'язуватися задача. У функцію передаємо ім'я файлу, в якому записані списки суміжності для кожної з вершин.
- Populate(int size) - заповнює початкову популяцію випадковими елементами. Розмір популяції - параметр size. Для генерації випадкових перестановок використовується функція std::random_shuffle, яка перетасовує елементи вхідної послідовності випадковим чином за допомогою функції std::swap, і головне - працює за лінійний час. Це важливо, тому що для алгоритму з великим параметром розміру популяції, початкове наповнення її елементами може займати немало часу.

Далі розглянемо реалізацію двох важливих функцій, що будуть використовуватися під час процесу обчислення наступних поколінь. Це функції кросоверу та оцінки часу.

Для функції оцінки часу маємо чіткий алгоритм, описаний в статті. Маємо дві впорядковані множини - інформовані та неінформовані вершини. Проходимо по множині неінформованих і знаходимо першу вершину, що має суміжну вершину в множині інформованих.

```
47  void GenAlg::Populate(int size)
48  {
49      std::vector<int> chromosome(graph.size());
50      for (int i = 0; i < graph.size(); ++i)
51          chromosome[i] = i;
52      population.push_back(chromosome);
53
54      for (int i = 0; i < size - 1; ++i)
55      {
56          std::random_shuffle(population[i].begin(), population[i].end());
57          population.push_back(population[i]);
58      }
59  }
60
```

Рисунок 8: Наповнення початкової популяції випадковими елементами

В якості інформатора, якщо є декілька варіантів, то обираємо, той, що йде першим в списку інформованих вершин.

Звернемо увагу на функції FirstAdjacentInInformed та GetIndexInInformed. Перша функція для кожної вершини з множини uninformed шукає першу суміжну їх вершину у списку informed та повертає -1, якщо такої вершини немає. Друга функція GetIndexInInformed визначає, на яку позицію потрібно вставити тільки що проінформовану вершину так, щоб зберегти впорядкованість вершин таку ж, як в початковій хромосомі, для якої ми і обраховуємо час.

З кожною ітерацією змінна time збільшується і таким чином ми отримаємо час, який було витрачено на схему розповсюдження повідомлення мережею для цієї конкретної хромосоми.

```
198     while (!uninformed_vertices.empty())
199     {
200         std::vector<int> unused_informators = informed_vertices;
201         for (int i = 0; i < uninformed_vertices.size(); ++i)
202         {
203             int informator_index = 0;
204             int informator = FirstAdjacentInInformed(uninformed_vertices[i], unused_informators,
205                                                                                 informator_index);
206             if (informator == -1)
207                 continue;
208
209             int index = GetIndexInInformed(uninformed_vertices[i], chromosome);
210             if (index == informed_vertices.size())
211                 informed_vertices.push_back(uninformed_vertices[i]);
212             else
213                 informed_vertices.insert(informed_vertices.begin() + index, uninformed_vertices[i]);
214             unused_informators.erase(unused_informators.begin() + informator_index);
215             uninformed_vertices.erase(uninformed_vertices.begin() + i);
216             i--;
217         }
218         time++;
219
220     }
```

Рисунок 9: Передача повідомлення в генетичному алгоритмі

Тепер розглянемо функцію схрещування двох хромосом.

Для цього нам знадобиться вектор пріоритетів - це послідовність всіх вершин графа у порядку зростання їх степеня. Щоб перетнути дві хромосоми, будемо послідовно брати відповідні їх елементи та порівнювати між собою. Для цього ми використовуємо допоміжну лямбда-функцію bool a_is_first(int a, int b) (рис. 10).

```
260     auto a_is_first = [this](int a, int b){
261         for (int i = 0; i < precedenceVector.size(); ++i)
262         {
263             if (precedenceVector[i] == a)
264                 return true;
265             if (precedenceVector[i] == b)
266                 return false;
267         }
268         return false;
269     };
```

Рисунок 10: Визначення пріоритетнішого гену

Ця функція приймає дві вершини (відповідні елементи обох батьків), та перевіряє, який з них зустрічається першим у векторі пріоритетів. Якщо елемент батька А переміг, то записуємо його у відповідний елемент дочірньої хромосоми, а далі необхідно модифікувати батька, що програв на цьому кроці таким чином, щоб на місці програвшої вершини тепер була та сама вершина, що і в іншого батька. Для цього просто знайдемо цю вершину та поміняємо її місцями з поточною. Розглянемо приклад.

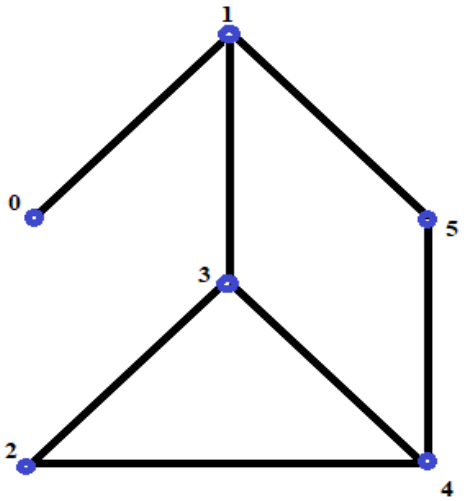

Рисунок 11. Приклад графу для демонстрації роботи генетичного алгоритму

Вектор пріоритетів для даного графу (рис. 11) може бути, наприклад:

(0, 5, 1, 4, 3, 2)

В якості батьків візьмемо такі дві хромосоми: [3 2 0 5 4 1], [0 1 2 5 3 4]

Порівнюємо 3 та 0: функція a_is_first повертає false, тобто переможець 0. Записуємо 0 на першу позицію дочірньої хромосоми і в другому батьку міняємо місцями 0 та 3. Отримуємо:

parents: A[0 2 3 5 4 1], B[0 1 2 5 3 4]; child: [0 * * * * *]

Аналогічно, наступні кроки:

parents: A[0 1 3 5 4 2], B[0 1 2 5 3 4]; 1 << 2 child: [0 1 * * * *]
parents: A[0 1 3 5 4 2], B[0 1 3 5 2 4]; 3 << 2 child: [0 1 3 * * *]
parents: A[0 1 3 5 4 2], B[0 1 3 5 2 4]; 5 == 5 child: [0 1 3 5 * *]
parents: A[0 1 3 5 4 2], B[0 1 3 5 4 2]; 4 << 2 child: [0 1 3 5 4 *]
parents: A[0 1 3 5 4 2], B[0 1 3 5 4 2]; 2 == 2 child: [0 1 3 5 4 2]

Тепер маємо можливість розпочинати обчислення.

Функція StartEvolution відповідає за генерацію нових елементів популяції та відбір вдалих екземплярів.

На початку заповнюємо вектор пріоритетів, який буде далі використовуватися при створенні нащадків (рис. 12). Починаємо головний цикл, в тілі якого генерується кожне наступне покоління. Випадковим чином обираємо батьків для нащадка і викликаємо описану вище функцію кросоверу.

```
77  void GenAlg::StartEvolution()
78  {
79      FillPrecedenceVector();
80      for (int count = 0; count < GenerationNumber; ++count)
81      {
82          int children_count = children_percent * population.size();
83          int current_population_size = population.size();
84          for (int i = 0; i < children_count; ++i)
85          {
86              int parentA = rand() % current_population_size;
87              int parentB = rand() % current_population_size;
88              auto child = Crossover(population[parentA], population[parentB]);
89
90              if (ShouldMutate())
91                  Mutate(child);
92
93              population.push_back(child);
94          }
95
96          SortByFitnessFunction();
97
98          int kill_count = kill_percent * population.size();
99          for (int i = 0; i < kill_count; ++i)
100             population.pop_back();
101     }
102 }
```

Рисунок 12: Функція, що керує процесом еволюції

Далі згідно зі вказаним процентом мутацій, можливо модифікуємо створену хромосому, міняючи місцями дві випадкові позиції в ній.

Після цього маємо нове покоління, з якого потрібно видалити найслабші елементи. Для цього посортуємо всю популяцію за зростанням функції оцінки часу. Використовуємо стандартну функцію std::sort з кастомізованим компаратором (рис. 13).

```
113  void GenAlg::SortByFitnessFunction()
114  {
115      std::sort(population.begin(), population.end(),
116          [this](const std::vector<int>& a, const std::vector<int>& b)
117          {
118              return FitnessFunction(a) < FitnessFunction(b);
119          });
120  }
```

Рисунок 13: Сортування популяції для відокремлення найслабших осіб

Таким чином, алгоритм пройшов одну повну ітерацію - утворив нащадків та видалив найслабші елементи популяції.

Тепер, за допомогою розробленої тестової системи, маємо змогу дослідити роботу даного генетичного алгоритму, а саме: дослідити, як залежить оптимальність отриманого рішення та швидкість його обчислення від таких параметрів алгоритму, як: коефіцієнт мутацій, розмір початкової популяції.

РОЗДІЛ 4. РЕЗУЛЬТАТИ ДОСЛІДЖЕНЬ НА ТЕСТОВІЙ СИСТЕМІ

4.1 Вхідні дані

Для створення тестових даних було розроблено застосунок, що дозволяє генерувати псевдовипадкові зв'язні графи з заданою кількістю вершин, що представлені списками суміжності.

Алгоритм генерації працює таким чином — спочатку випадковим чином ініціалізуємо змінну VertexToConnect. Далі в циклі підбираємо несуміжну їй випадкову вершину і додаєм ребро до графа. Функція IsConnected(int&) перевіряє граф на зв'язність за допомою пошуку в глибину, і, якщо граф не є зв'язним, записує в змінну VertexToConnect випадково обрану вершину з тих, що не були помічені під час пошуку в глибину.

```
int VertexToConnect = rand() % N;
do
{
	int k;
	do
	{
		k = rand() % N;
	} while (AreAdjacent(k, VertexToConnect));
	adj_lists[k].push_back(VertexToConnect);
	adj_lists[VertexToConnect].push_back(k);
} while (!IsConnected(VertexToConnect));
```

За допомогою цього застосунку було створено датасети з графами на 10, 20, 50, 100 та 500 вершин. В якості правильної відповіді запустимо алгоритм зі значним розміром початкової популяції, таким чином ми отримаємо масив рішень для кожного графа, що з великою ймовірністю будуть оптимальними.

4.2 Статистичні дані

Оглянемо процес еволюції на прикладі невеликого графу. Після створення кожного нового покоління будемо виводити оцінки кожного елемента в порядку зростання. Можемо спостерігати, як з кожним новим поколінням, перестановки поступово покращуються. У випадку з меншим початковим розміром популяції, алгоритму знадобилось згенерувати більше поколінь, порівняно з другим запуском, де об’єм початкової популяції було задано більшим.

```
P = 6
Generation 1: 5 6 6 6 6 7
Generation 2: 5 5 6 6 6 6
Generation 3: 5 5 6 6 6 6
Generation 4: 5 5 5 6 6 6
Generation 5: 5 5 5 6 6 6

P = 18
Generation 1: 5 6 6 6 7 7 7 7 7 7 7 7 7 8 8 8 8 8
Generation 2: 5 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 8 8
Generation 3: 5 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7
Generation 4: 5 5 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 8
Generation 5: 5 5 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
```

Рисунок 14: Еволюція поколінь на прикладі одного графу з різним початковим об'ємом популяції

Мінімальна кількість поколінь доівнює п’яти, тому як для умови зупинення алгоритму, необхідно щоб 5 поколінь підряд не покращувався результат. Статистика для графа з 10 вершинами:

		n=10			n=20		
		Evolution	Generations	Error	Evolution	Generations	Error
1%	P = 5	16.62	5.28	0.573333	61.65	5.57	1.06494
	P = 15	81.34	5.12	0.529412	284.28	5.27	0.8
	P = 50	190.9	5	0.445946	758.63	5.29	0.920635
3%	P = 5	17.24	5.27	0.5	62.88	5.47	1.27848
	P = 15	82.2	5.06	0.507042	334.22	5.31	1.2
	P = 50	189.97	5	0.478873	806.87	5.11	0.777778
8%	P = 5	18.25	5.2	0.493506	67.17	5.37	1.2875
	P = 15	80.83	5.05	0.550725	344.47	5.43	1.02778
	P = 50	194.67	5	0.452055	798.43	5.16	1.04839

Рисунок 15. Результати запуску алгоритмів на графах з 10 та 20 вершин

Так як ми не маємо точної відповіді і порівнюємо результати з запуском алгоритму на великій кількості популяції, потрібно врахувати, що справжнє значення помилки є трохи більшим. В деяких окремих випадках при невдалій початковій популяції, було отримано перевірочні значення, що виявились гіршими за обчислення для збору статистики.

Для графів з більшою кількістю вершин обчислення займає значну кількість часу. Виведемо статистику по невеликому набору випадкових графів на 100 вершин (об’єм вибірки 10):

		n=100		
		Evolution	Generations	Error
1%	P = 5	603.8	5.5	0.777778
	P = 15	4440.4	6.3	0.777778
	P = 50	13808.3	5.8	0.857143
3%	P = 5	740.5	5.6	1.44444
	P = 15	4068.9	6.1	0.857143
	P = 50	10798.9	5.3	1.28571
8%	P = 5	634.6	5.7	1.33333
	P = 15	4002.1	6	0.9
	P = 50	9237.3	5.6	1

Рисунок 16: Запуск алгоритму на графах на 100 вершин

Тестування на графах, для яких відомий оптимальний розв’язок, показало, що генетичний алгоритм легко розв’язує задачу на таких графах. Наприклад, випадок, коли оптимальний розв’язок є єдиним можливим (кільцеві графи), або коли існує кілька можливих розв’язків, які мають однакову оцінку (повні бінарні дерева). Також алгоритм було перевірено на дворядкових графах решітки — в усіх випадках оптимальний розв’язок було знайдено максимально швидко.

ВИСНОВКИ

В даній роботі було оглянуто існуючі алгоритми розв’язання задачі про мінімальний час передачі повідомлення. Дана задача є NP-повною, тому не відомо ефективних алгоритмів, що розв’язують її в загальному випадку за поліноміальний час. Евристичні алгоритми, які не завжди надають оптимальне рішення, на практиці є дуже корисними і широко застосовуються у сфері комп’ютерних мереж.

Було детально розглянуто два евристичні алгоритми: ітеративний та генетичний. Для ітеративного алгоритму було створено програмну реалізацію, за допомогою якої можна спостерігати процес роботи алгоритму. Також для генетичного алгоритму було створено тестову систему, яка дозволяє задавати різні параметри алгоритму, такі як: розмір початкової популяції та процент мутацій. Таким чином, за допомогою розробленої тестової системи було досліджено роботу алгоритму в залежності від різних значень параметрів на різних наборах даних — випадкових зв’язних графах, а також двохрядкових графах решітки. Як можна побачити по зібраній статистиці, не завжди розмір популяції гарантує оптимальність відповіді.

Генетичний алгоритм для великих об’ємів даних працює досить повільно, тому потрібно уважно підбирати параметри, знаходячи компроміс між швидкістю роботи та оптимальністю відповіді. Коефіцієнт мутацій є важливим параметром, що може позитивно вплинути на роботу алгоритму при невдалій початковій популяції. У запусках с великим об’ємом початкової популяції більший коефіцієнт мутацій означав меншу ймовірність помилки, тому що він вплинув на одиничні випадки, в яких запуск з більшою популяцією приносив менш оптимальний розв’язок порівняно з найоптимальнішим розв’язком серед всіх запусків по цьому графу.

Тестування на графах з очевидним оптимальним розв’язком показало, що генетичний алгоритм знаходить оптимальний розв’язок на першому ж кроці для графів, в яких оптимальний розв’язок є єдиним можливим (будь-яка перестановка вершин лише встановлює пріоритет вершин для передачі, таким чином завжди буде знайдена єдина вершина, в яку потрібно передати повідомлення на наступному кроці). Також було проведено тестування на графах, в яких оптимальний розв’язок не є єдиним можливим, але є достатньо очевидним (дворядковий граф решітки), і результати запусків показали, що такі випадки є легко розв’язуваними для даного генетичного алгоритму.

Перелік джерел посилання

1. Zhou J. A minimum broadcast graph on 26 vertices [Електронний ресурс] / J. Zhou, K. Zhang – Режим доступу до ресурсу: https://doi.org/10.1016/S0893-9659(01)00082-9.
2. Heuristics for the Minimum Broadcast Time / [A. Sousa, G. Gallo, S. Gutierrez та ін.]., 2018. – 2 с.
3. Garey M. R. Computers and Intractability: A Guide to the Theory of NP-Completeness / M. R. Garey, D. S. Johnson. – San Francisco, 1979. – 219 с.
4. Jansen K. The minimum broadcast time problem for several processor networks / K. Jansen, H. Müller., 1995.
5. Slater P. Information Dissemination in Trees / P. Slater, E. Cockayne, S. Hedetniemi., 1981.
6. Minimum broadcast graphs / A.Farley, S. Hedetniemi, S. Mitchell, A. Proskurowski., 1979.
7. Harutyunyan H. A. An Iterative Algorithm for the Minimum Broadcast Time Problem / H. A. Harutyunyan, C. D. Morosan., 2004.
8. Hoelting C. J. A genetic algorithm for the minimum broadcast time problem using a global precedence vector / C. J. Hoelting, D. A. Schoenefeld, R. L. Wainwright., 1996.