КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (29), 2025
ISSN 2663 - 4023
DOI 10.28925/2663-4023.2025.29.822
УДК 004.4

**Хотинський Владислав Віталійович**
студент факультету інформаційних технологій та математики
Волинський національний університет імені Лесі Українки, Луцьк, Україна
*khotynskyi.vladyslav2022@vnu.edu.ua*

**Павленко Юлія Степанівна**
старший викладач кафедри комп'ютерних наук та кібербезпеки
Волинський національний університет імені Лесі Українки, Луцьк, Україна
ORCID ID: 0000-0002-4065-045X
*Pavlenko.Yulya@vnu.edu.ua*

# ОДИН ІЗ СПОСОБІВ РЕЛІЗАЦІЇ МУРАШИНОГО АЛГОРИТМУ НА ПРИКЛАДІ РОЗВ'ЯЗКУ ЗАДАЧІ КОМІВОЯЖЕРА

**Анотація.** У сучасному світі оптимізаційні задачі мають ключове значення в логістиці, енергетиці, промисловості, фінансах, медицині, машинному навчанні тощо. Задача оптимізації — це математична задача пошуку найкращого рішення з-поміж усіх можливих, за певним критерієм (мінімізація витрат, часу, максимізація прибутку). Існують точні методи, що дають оптимальний результат (метод повного перебору, динамічне програмування), проте вони неефективні для задач великого масштабу. Евристичні та метаевристичні методи (жадібні, генетичні, мурашині алгоритми) дають наближені, але практично ефективні рішення. Однією із задач оптимізації є задача комівояжера, метою якої є пошук найкоротшого маршруту, що проходить через усі міста один раз. Мурашиний алгоритм є одним із ефективних алгоритмів для знаходження наближених розв'язків задачі комівояжера, а також аналогічних завдань пошуку маршрутів на графах. Під час пошуку їжі мурахи залишають феромони на шляху. Агенти (мурахи) обирають шляхи за ймовірністю, що залежить від інтенсивності феромонів та довжини маршруту. Кожна мураха поступово будує маршрут, феромони оновлюються після ітерацій з урахуванням якості рішень. Параметри алгоритму (α, β, ρ) впливають на вибір маршруту, здатність до адаптації та уникнення застарілих рішень. В статті наведений процес проєктування та розробки програмного продукту, який забезпечує генерацію графа та реалізує на ньому пошук найкоротшого маршруту з допомогою мурашиного алгоритму. Програмна реалізація алгоритму оптимізації мурашиної колонії виконана з використанням HTML, CSS і JavaScript. Однією з важливих особливостей програми є візуалізація графа та роботи алгоритму. Графічне представлення здійснюється за допомогою елемента Canvas, що дозволяє відображати вершини у вигляді кольорових кіл, а ребра — у вигляді ліній із товщиною та прозорістю, які відповідають інтенсивності феромонів.

**Ключові слова:** задачі оптимізації; мурашиний алгоритм; феромони; проєктування програмного засобу; граф; найкоротший маршрут.

## ВСТУП

В сучасному світі оптимізаційні задачі відіграють ключову роль у багатьох сферах: мінімізація витрат на паливо або час доставки, оптимальне завантаження машин та верстатів на заводі, зниження споживання електроенергії в міських системах, планування графіків роботи персоналу, оптимізація маршрутів доставки, інвестиційного портфеля, дозування ліків, алгоритмів машинного навчання тощо. Застосування оптимізаційних алгоритмів є фундаментальною основою для створення ефективних рішень у складних системах, а їх розвиток дозволяє вирішувати задачі, які раніше вважалися обчислювально недосяжними. В загальному задачі оптимізації — це математичні задачі, у яких потрібно знайти найкраще рішення з множини можливих

варіантів, керуючись певними критеріями (наприклад, мінімізація витрат, максимізація прибутку, зменшення часу тощо).

**Постановка проблеми.** Для вирішення оптимізаційних задач застосовуються різні алгоритми, які можна умовно розділити на два типи:

- точні методи: такі, що забезпечують знаходження оптимального рішення (наприклад, метод повного перебору або метод динамічного програмування). Ці методи ефективні для задач невеликого масштабу, але швидко втрачають продуктивність при збільшенні розмірності;
- евристичні та метаевристичні методи: забезпечують знаходження наближеного рішення (наприклад, жадібні алгоритми, генетичні алгоритми, мурашині алгоритми). Ці підходи демонструють високу ефективність у вирішенні великих і складних задач [1] – [3].

Одним із ефективних методів розв'язання оптимізаційних задач є мурашиний алгоритм (Ant Colony Optimization, ACO), який використовує природню поведінку мурах у пошуку найкоротшого шляху до джерела їжі, а саме — механізм підсилення феромонних слідів, які мурахи залишають на маршрутах. З часом цей процес дозволяє знайти оптимальні або близькі до оптимальних рішення, адаптуючись до змінних умов середовища. Завдяки своїм особливостям, мурашиний алгоритм ефективно застосовується у комбінаторних задачах, зокрема для розв'язання задачі комівояжера, оптимізації розкладів, розподілу ресурсів та аналізу транспортних потоків [4].

**Метою статті** є опис процесу проєктування та розробки програмного засобу, що дозволяє вирішити задачу комівояжера з допомогою мурашиного алгоритму із візуалізацією процесу пошуку рішення.

## ТЕОРЕТИЧНІ ОСНОВИ ДОСЛІДЖЕННЯ

Комбінаторна оптимізація є розділом математичного програмування, що займається пошуком оптимального рішення у дискретних структурах. Вона має справу із задачами, де простір можливих рішень утворює величезну, але скінченну множину варіантів. Такі задачі зазвичай формулюються у вигляді графів, множин або інших дискретних об'єктів. Основна мета комбінаторної оптимізації — знайти найкраще рішення, яке задовольняє визначені критерії, такі як мінімізація витрат чи максимізація ефективності [5].

Задача комівояжера є класичним прикладом комбінаторної оптимізації та формулюється так: «Є набір міст і відстані між ними. Потрібно знайти найкоротший маршрут, який проходить через всі міста рівно один раз і повертається у початкове місто».

Ця задача належить до NP-складних, оскільки кількість можливих рішень зростає експоненційно зі збільшенням кількості міст. Це робить використання точних методів, таких як повний перебір чи динамічне програмування, неефективним для задач великого масштабу [6], [7].

Застосування комбінаторної оптимізації до задачі комівояжера дозволяє не лише моделювати цю проблему за допомогою графів, де вершини відповідають містам, а ребра — відстаням між ними, але й обирати ефективні методи пошуку розв'язків. Евристичні та метаевристичні підходи, зокрема мурашиний алгоритм, демонструють значно вищу ефективність при роботі з великими графами.

Мурашиний алгоритм належить до метаевристичних методів, розроблених на основі спостережень за поведінкою колоній мурах у природі. Головна ідея алгоритму

полягає у використанні колективної взаємодії та механізмів, через які мурахи знаходять найкоротші шляхи між своїм гніздом і джерелами їжі [5].

У природі мурахи під час пошуку їжі залишають феромонний слід на своєму шляху. Інші мурахи, рухаючись навмання, з більшою ймовірністю обирають маршрути з сильнішим феромонним слідом. Якщо цей маршрут дійсно ефективний все більше мурах починають його використовувати, посилюючи феромонний слід. Таким чином, шлях із найсильнішим слідом стає пріоритетним для всієї колонії. З часом феромони випаровуються, що дозволяє колонії уникати старих, неефективних маршрутів і пристосовуватися до змін навколишнього середовища.

Цей принцип став основою мурашиного алгоритму. У математичній моделі кожна мураха символізує агента, який досліджує можливі рішення задачі. Мураха обирає свій наступний крок на основі ймовірності, яка залежить від двох факторів: інтенсивності феромонного сліду та привабливості маршруту.

Мурашиний алгоритм реалізує оптимізацію шляхом ітеративного пошуку рішень, що враховують як феромонні сліди, так і довжину маршруту. Робота алгоритму складається з кількох ключових етапів, кожен з яких спрямований на пошук та уточнення оптимального рішення [6] – [8].

На початковому етапі створюється граф, що представляє задачу. Наприклад, у задачі комівояжера вершини графа відповідають містам, а ребра — відстаням між ними. На кожне ребро накладається початковий феромонний слід, який є однаковим для всіх маршрутів. Крім того, встановлюються параметри алгоритму: кількість мурах, кількість ітерацій, коефіцієнти впливу феромонів (α), і довжин маршрутів (β), а також коефіцієнт випаровування феромонів [4].

На кожній ітерації алгоритму мурахи починають свій шлях із випадково обраних точок графа. Кожна мураха поступово будує рішення, обираючи наступну вершину на основі ймовірності переходу з одного міста в інше. Цю ймовірність обчислюють за формулою [8], [9]:

$$P_{ij} = \frac{(\tau_{ij})^{\alpha} \cdot (\eta_{ij})^{\beta}}{\sum_{k \in N_i} (\tau_{ik})^{\alpha} \cdot (\eta_{ik})^{\beta}}$$

де $\tau_{ij}$ — рівень феромону на ребрі $i \to j$; $\eta_{ij}$ — інверсія довжини шляху; $\alpha$ і $\beta$ — коефіцієнти, що контролюють вплив феромонів і довжини шляху; $N_i$ — набір доступних вершин для мурахи в поточному стані.

Після того, як усі мурахи завершили свої маршрути, феромонні сліди оновлюються. Нижче наведені формули для оновлення феромонів. Спочатку феромони випаровуються на кожному ребрі залежно від коефіцієнта випаровування. Тоді ребра, якими користувалися мурахи, отримують ще додаткову кількість феромонів, що пропорційна якості їхнього маршруту [8], [9]:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \sum_{k} \Delta\tau_{ij}^{k}$$

де $\rho$ — коефіцієнт випаровування феромонів; $\Delta\tau_{ij}^{k}$ — кількість феромону, залишена $k$-ю мурахою.

Кожною ітерацією алгоритм зберігає інформацію про найкращий маршрут, знайдений мурахами. Якщо новий маршрут має коротшу довжину, ніж попередній найкращий, він зберігається як поточне оптимальне рішення. Процес повторюється протягом заданої кількості повторень або до досягнення критерію зупинки (наприклад,

коли найкращий маршрут не змінюється протягом декількох ітерацій). У підсумку повертається найкоротший маршрут, знайдений алгоритмом.

Мурашиний алгоритм залежить від набору параметрів, які визначають його ефективність та здатність знаходити оптимальні або наближені до оптимальних рішення. Найважливішими серед них є коефіцієнти α, β та швидкість випаровування феромонів ρ. Кожен з цих параметрів впливає на поведінку мурах і на те, як алгоритм досліджує простір можливих рішень.

Коефіцієнт α визначає, наскільки сильно мураха враховує феромонний слід при виборі наступного маршруту. Чим більший коефіцієнт α, тим більший вплив має феромон, і тим сильніше мурахи «дотримуються» вже прокладених шляхів.

Коефіцієнт β визначає важливість локальної евристичної інформації, наприклад, оберненої величини довжини маршруту під час вибору наступного шляху. Чим більший коефіцієнт β, тим більшу вагу мураха надає коротшим шляхам.

Коефіцієнт випаровування феромонів ρ визначає швидкість, з якою феромонний слід слабшає на маршрутах, що не використовуються. Він задається як множник, що зменшує інтенсивність феромону на кожній ітерації. Високе значення ρ (наприклад, ρ = 0,8) означає швидке випаровування феромонів. Це дозволяє алгоритму швидше забувати застарілі маршрути й зосереджуватися на нових, але може призвести до втрати корисної інформації. Низьке значення ρ (наприклад, ρ = 0,1) уповільнює випаровування, що сприяє більшому збереженню інформації про попередні маршрути. Проте це може призвести до перенасичення феромонів і застрягання алгоритму на неефективних та застарілих рішеннях [9], [10].

## РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Розроблюване програмне забезпечення повинно забезпечити інтерактивну візуалізацію роботи мурашиного алгоритму для вирішення задачі комівояжера, включаючи відображення графа, маршрутів та феромонних слідів, а також підтримувати генерацію випадкових графів. Його реалізація повинна гарантувати зручність у використанні, оптимальну швидкість виконання для задач середнього масштабу (10–50 вершин), сумісність із сучасними веббравзерами, кросплатформність.

Програмний продукт повинен виконувати наступні функції:

- генерація графа задачі: створення випадкових наборів вершин (міст) із заданими координатами та побудова графа з відображенням ребер (шляхів);
- реалізація мурашиного алгоритму: забезпечення пошуку найкоротшого маршруту із застосуванням феромонних слідів, враховуючи параметри α, β і швидкість випаровування ρ;
- інтерактивне налаштування параметрів алгоритму: кількість ітерацій, коефіцієнти впливу α, β;
- візуалізація процесу: динамічне відображення руху мурах, зміни інтенсивності феромонів та побудови маршруту;
- виведення результатів: відображення найкращого маршруту, його довжини та порівняння з іншими знайденими рішеннями;
- зручний інтерфейс: забезпечення інтуїтивної роботи користувача з програмою.

Програмна реалізація алгоритму оптимізації мурашиної колонії виконана з використанням HTML, CSS і JavaScript [11] та включає у себе кілька ключових компонентів, що забезпечують виконання основних етапів алгоритму, а також динамічну візуалізацію процесу пошуку оптимального рішення. Розглянемо їх детальніше.

ISSN 2663 - 4023

Однією з таких функцій є генерація графа задачі. Граф задачі реалізовано у вигляді матриці суміжності, де кожен елемент відповідає відстані між містами, яка обчислюється за допомогою формули Евклідової відстані (рис. 1).

```
main.js:289
▼ (10) [Array(10), Array(10), Array(10), Array(10), Array(10), Array(10), Array(10), Array(10), Array(10), Array(10)]
  ▶ 0: (10) [0, 444, 305, 320, 635, 466, 376, 498, 211, 169]
  ▶ 1: (10) [444, 0, 505, 259, 224, 168, 299, 377, 257, 596]
  ▶ 2: (10) [305, 505, 0, 252, 603, 417, 603, 308, 409, 285]
  ▶ 3: (10) [320, 259, 252, 0, 361, 173, 435, 196, 267, 419]
  ▶ 4: (10) [635, 224, 603, 361, 0, 188, 516, 357, 473, 767]
  ▶ 5: (10) [466, 168, 417, 173, 188, 0, 446, 215, 339, 586]
  ▶ 6: (10) [376, 299, 603, 435, 516, 446, 0, 616, 194, 543]
  ▶ 7: (10) [498, 377, 308, 196, 357, 215, 616, 0, 461, 561]
  ▶ 8: (10) [211, 257, 409, 267, 473, 339, 194, 461, 0, 379]
  ▶ 9: (10) [169, 596, 285, 419, 767, 586, 543, 561, 379, 0]
    length: 10
  ▶ [[Prototype]]: Array(0)
```

*Рис. 1. Приклад матриці відстаней між містами*

На початковому етапі всі ребра графа отримують однакову початкову кількість феромонів, яка зберігається у спеціальній матриці (рис. 2).

```
main.js:290
▼ (10) [Array(10), Array(10), Array(10), Array(10), Array(10), Array(10), Array(10), Array(10), Array(10), Array(10)]
  ▶ 0: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 1: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 2: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 3: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 4: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 5: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 6: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 7: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 8: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
  ▶ 9: (10) [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
    length: 10
  ▶ [[Prototype]]: Array(0)
```

*Рис. 2. Приклад матриці феромонів між містами*

Вершини графа (міста) генеруються випадковим чином із дотриманням обмежень на мінімальну відстань між ними, щоб уникнути накладання елементів на візуальній схемі (рис. 3).

```
75  const generateRandomCities = () => {
76      ctx.clearRect(0, 0, canvas.width, canvas.height);
77      let valid = false;
78      let cityDiametr = 2 * cityRadius;
79      let minCityDistance = 8 * cityRadius;
80      let x, y;
81      cities = []
82          for (let i = 0; i < numOfCities; i++) {
83              while (!valid) {
84                  x = Math.ceil(Math.random() * (canvas.width - 2 * cityDiametr) + cityDiametr);
85                  y = Math.ceil(Math.random() * (canvas.height - 2 * cityDiametr) + cityDiametr);
86
87                  valid = cities.every(city => findDistance({x, y}, {x: city.x, y: city.y}) >= minCityDistance);
88              }
89
90              cities.push({x, y});
91              valid = false;
92          }
93          drawSities();
94  };
```

*Рис. 3. Фрагмент коду, що реалізує генерацію вершин графа (міст)*

ISSN 2663 - 4023

Приклад одного із варіантів згенероваого графа наведено на рис. 4.


*Рис. 4. Знімок екрану з прикладом згенерованого графу*

Важливим етапом є вибір наступного міста для кожної мурахи. Цей процес базується на ймовірнісному підході, що враховує два ключові чинники: інтенсивність феромонів на ребрах графа та відстані між містами. Розрахунок ймовірностей дозволяє збалансувати вплив цих факторів через параметри, які регулюють важливість феромонів і відстаней. Такий підхід дає змогу формувати перспективні маршрути, які водночас враховують історію попередніх маршрутів та фактор нового маршрута. Не менш важливим є те, що для кожної мурахи обчислюється загальна довжина маршруту після завершення всіх переходів між містами. Це дає змогу відслідковувати пройдену відстань мурахою та в подальшому порівняти її з довжиною шляху інших мурах.

В проєкті цю логіку реалізовано через клас Ant (рис. 5), який моделює поведінку окремої мурахи, відповідаючи за побудову маршруту, вибір наступного міста та обчислення загальної довжини шляху. Конструктор класу ініціалізує початкові параметри: початкове місто, поточне місто, список відвіданих міст та загальну довжину маршруту. Це дозволяє відстежувати прогрес мурахи під час виконання алгоритму.

Метод chooseNextCity() відповідає за вибір наступного міста для мурахи на основі ймовірностей, які розраховуються з використанням феромонів і відстаней між містами. Для кожного міста обчислюється значення «бажання» досягти його, що враховує інтенсивність феромону та відстань до нього. Потім ці «бажання» нормалізуються для визначення ймовірностей переходу до кожного з доступних міст.

На основі випадкового числа мураха вибирає наступне місто, додає його до списку відвіданих, оновлює загальну довжину маршруту і переміщується до нового міста. Таким чином, клас реалізує логіку прийняття рішень для кожної мурахи на основі локальних даних.

Не менш важливим етапом є оновлення феромонів. Оновлення феромонів у програмі відбувається у два етапи. Спочатку виконується випаровування феромонів, що дозволяє поступово знижувати їхню інтенсивність на менш перспективних маршрутах. Потім додається нова кількість феромона на маршрутах, якими скористалися мурахи, причому кількість доданого феромона залежить від якості маршруту — коротші шляхи отримують більше підкріплення у вигляді більшої кількості феромона.

ISSN 2663 - 4023

```
class Ant{
    constructor(initialCity){
        this.initialCity = initialCity; // Задається початкове місто
        this.currentCity = initialCity;  // Поточне місто, на початку збігається з початковим
        this.visitedCities = [initialCity]; // Місто, яке мураха відвідала, ініціалізується початковим
        this.totalPathLength = 0; // Початкова довжина маршруту
    }

    chooseNextCity(pheromones, distances) {
        let probabilities = [];
        let desiresToTransition = [];
        let totalDesires = 0;

        // Якщо всі міста відвідані, повертаємось у початкове
        if (this.visitedCities.length === cities.length) {···
        }

        // Обчислюємо бажання (ймовірності) переходу для невідвіданих міст
        for (let i = 0; i < cities.length; i++) {···
        }

        // Обчислення ймовірностей переходу
        for (let i = 0; i < desiresToTransition.length; i++) {···
        }

        // Вибір наступного міста на основі ймовірності
        let rand = Math.random();
        for (let prob of probabilities) {···
        }
        console.log(this.visitedCities) //Виводими відвідані міста в консоль
    }
}
```

*Рис. 5. Фрагмент коду: опис класу Ant*

У фрагменті коду (рис. 6) функція updatePheromones() реалізує цей функціонал — оновлення феромонів на ребрах графа. Вона моделює два основні процеси: випаровування старих слідів і підсилення феромонів на основі якості знайдених маршрутів. На етапі випаровування всі значення феромонів у матриці pheromones зменшуються шляхом множення на коефіцієнт evaporationInfluence, що регулює швидкість цього процесу. Це запобігає накопиченню феромонів на застарілих маршрутах і стимулює дослідження нових рішень.

```
const updatePheromones = (pheromones, ants, evaporationInfluence) => {
    // Випаровування феромонів на всіх ребрах графа
    for (let i = 0; i < pheromones.length; i++) {
        for (let j = 0; j < pheromones[i].length; j++) {
            pheromones[i][j] *= evaporationInfluence;
        }
    }
    // Додавання нових феромонів за маршрутами, пройденими мурахами
    ants.forEach(ant => {
        // Проходимо по кожному ребру маршруту мурахи
        for (let i = 0; i < ant.visitedCities.length - 1; i++) {
            let currentCityNumber = ant.visitedCities[i];
            let nextCityNumber = ant.visitedCities[i + 1];

            // Додаємо феромони до ребра в двох напрямках
            pheromones[currentCityNumber][nextCityNumber] += Q / ant.totalPathLength;
            pheromones[nextCityNumber][currentCityNumber] += Q / ant.totalPathLength;
        }
    })
}
```

*Рис. 6. Фрагмент коду: функція для оновлення феромонів на ребрах*

Додавання нових феромонів здійснюється для кожної мурахи за її маршрутом.

Значення, що додається до ребра, пропорційне якості маршруту: чим коротший маршрут, тим більший внесок у феромонний слід. Константа $Q$ використовується для масштабування цього ефекту, а оновлення виконується симетрично для обох напрямків кожного ребра, оскільки граф є неорієнтованим. Ця функція забезпечує адаптивність алгоритму, дозволяючи йому зосереджуватися на перспективних маршрутах і поступово покращувати якість знайдених рішень з кожною ітерацією.

Програма надає користувачу можливість налаштовувати параметри алгоритму через інтерактивний інтерфейс. Зокрема, можна змінювати кількість ітерацій, значення коефіцієнтів, що визначають вплив феромонів і відстані, а також можна показати найкращий шлях знайдений мурахами. Ці параметри дозволяють адаптувати алгоритм до різних задач і дослідницьких потреб (рис. 7).

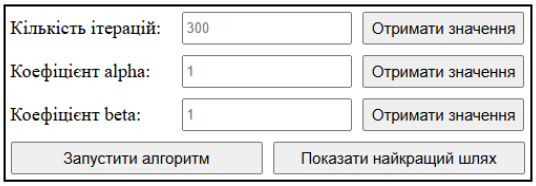


*Рис. 7. Знімок екрану з інтерфейсом для налаштування параметрів алгоритму*

Однією з важливих особливостей програми є візуалізація графа та роботи алгоритму. Графічне представлення здійснюється за допомогою елемента Canvas, що дозволяє відображати вершини у вигляді кольорових кіл, а ребра — у вигляді ліній із товщиною та прозорістю, які відповідають інтенсивності феромонів.

Після генерації графа (рис. 4) розпочинається пошук найкращого маршруту, що візуально забезпечується появою ребер між вершинами. Чим більше разів мурахи «пройшли» по шляху від вершини до вершини, тим ширшою і насиченішою буде лінія, що позначає ребро. Для кожної ітерації яскравим кольором позначається найкращий маршрут (рис. 8).

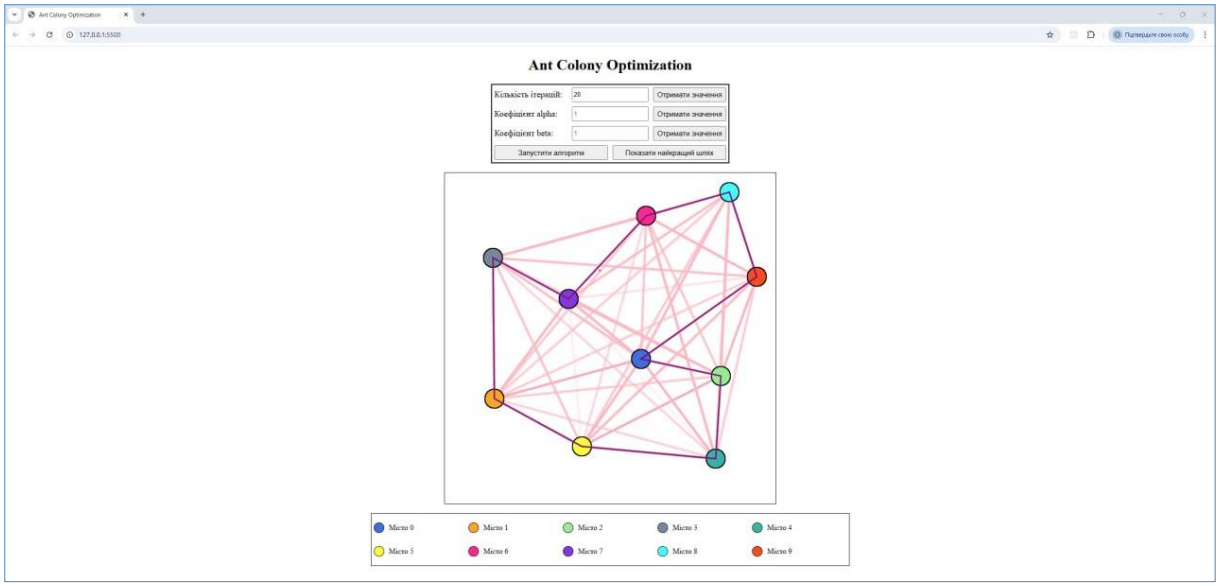


*Рис. 8. Знімок екрану роботи програми: рух мурах спричиняє появу ребер*

ISSN 2663 - 4023

Результатами роботи програми є побудований граф, в якому контрастним кольором позначений найкращий знайдений маршрут (рис. 9), та виведений в консоль результат обчислень алгоритму (рис. 10).

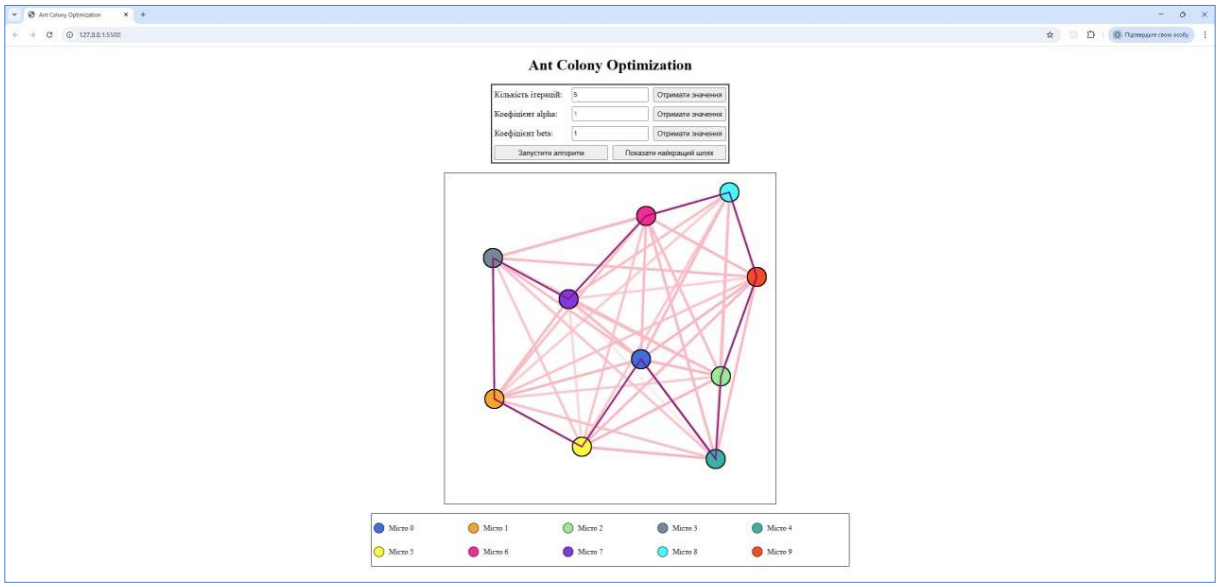


*Рис. 9. Знімок екрану з результатом роботи програми*

Для наочності на рис. 4, 8–10 наведені знімки екрану роботи програмного засобу для одного і того ж графу.

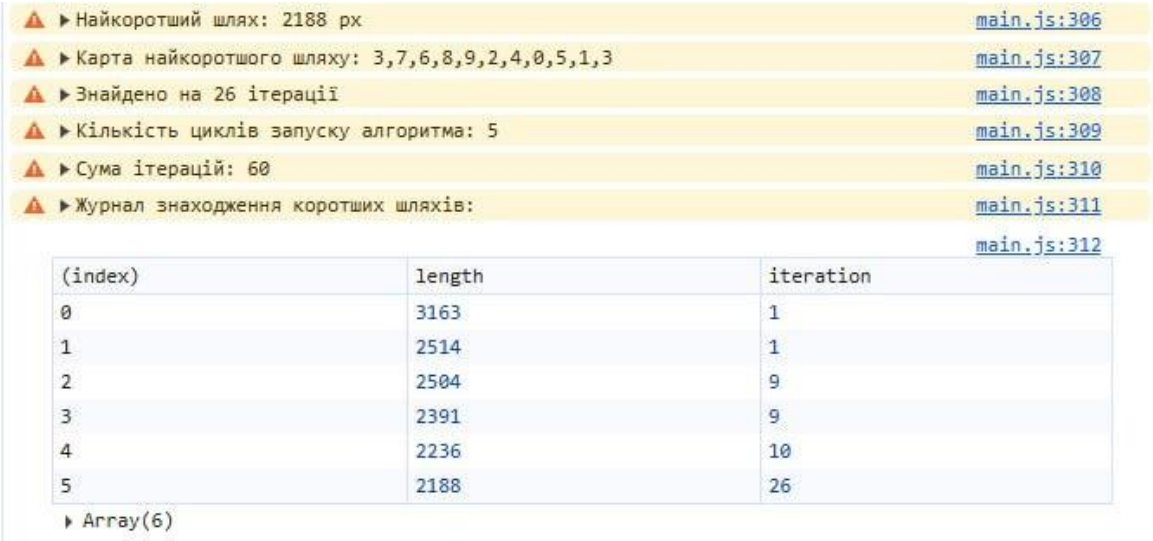


*Рис. 10. Приклад одного результату обчислень алгоритму, виведений в консоль*

Програмний засіб був перевірений на коректність обчислення відстаней між містами та правильність взаємодії користувача з алгоритмом через форму. Для перевірки коректності обчислення відстаней між містами використовувалися заздалегідь визначені координати точок. На їх основі було розраховано матрицю відстаней за формулою евклідової метрики. Обчислені значення порівнювалися з теоретично очікуваними результатами і перевірялись на достовірність. Тестування взаємодії з користувачем через

форму було спрямоване на перевірку коректності введення та обробки параметрів алгоритму. Були протестовані поля для введення ключових налаштувань: кількість ітерацій, коефіцієнти α та β. Перевірка включала як коректні значення, так і випадки введення некоректних даних, наприклад, від'ємних чисел або символів. Програма коректно обробляє параметри та повертає очікувані результати.

## ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

Розроблений програмний засіб у вигляді вебдодатку реалізує мурашиний алгоритм для вирішення задачі комівояжера. Його завданням та результатом є пошук найкоротшого маршруту між усіма заданими точками графа із можливістю візуалізації процесу пошуку рішення та налаштування параметрів алгоритму, таких як кількість ітерацій, значення α, β. Загальна архітектура програми побудована модульно. Окремі модулі відповідають за логіку алгоритму, графічне представлення та взаємодію з користувачем через інтерфейс. Така структура забезпечує гнучкість у розробці, легкість у внесенні змін і розширенні функціональності при використанні функцій повторно.

Реалізація анімації руху мурах додає динамічності програмі. У реальному часі користувач може спостерігати, як мурахи обирають маршрути, як змінюються феромонні сліди, і як поступово формується найкращий маршрут. Це сприяє кращому розумінню роботи алгоритму. Також реалізовано механізм збереження результатів. Після завершення роботи алгоритму відображається інформація про найкращий знайдений маршрут і його довжину.

Розроблений додаток дозволяє не лише вивчати принципи алгоритму, а й застосовувати його для вирішення задач оптимізації, таких як планування маршрутів і розподіл ресурсів. Ця задача має велике практичне значення, оскільки вона виникає в багатьох реальних сценаріях, які потребують ефективного планування та оптимізації ресурсів.

У перспективіх передбачається розширення функціональності додатку шляхом інтеграції інших метаевристичних алгоритмів, а також адаптація системи до задач з реальними обмеженнями та динамічними параметрами, що дозволить застосовувати її в більш складних і практично орієнтованих середовищах.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. An Overview of Ant Colony Optimization Algorithms for Dynamic Optimization Problems. *IntechOpen - Open Science Open Minds | IntechOpen*. https://www.intechopen.com/chapters/87285. (2023)
2. Носенко В.А. (2018). Евристичні алгоритми. Харків : ХНУ. 223 с.
3. Elitist Ant System | Algorithm Afternoon. *Home | Algorithm Afternoon*. https://algorithmafternoon.com/ants/elitist_ant_system/
4. Левченко О.О. (2020). Оптимізаційні методи та алгоритми. Дніпро: ДНУ. 340 с.
5. Ines Alaya, Christine Solnon, Khaled Ghedira. (2020). Ant algorithm for the multidimensional knapsack problem. International conference on Bioinspired Methods and their Applications (BIOMA), Ljubljana, Slovenia. pp.63-72. https://hal.science/hal-01541529/file/bioma04.pdf.
6. Гаврилюк В.В. (2018). Алгоритми та методи обчислювального інтелекту. Львів: ЛНУ ім. Івана Франка. 312 с.
7. Corman T. H. (2001). Introduction to Algorithms. Cambridge, Massachusetts : The MIT Press. 1157 p.
8. Штовба С. Д., Рудий О. М. (2012). Мурашині алгоритми оптимізації // Інформаційні технології та комп'ютерна техніка. 2012. № 78. С. 123-130.
9. Dorigo M., Stützle T. (2026). The ant colony optimization metaheuristic: Algorithms, applications, and advances. Handbook of metaheuristics. https://www.researchgate.net/publication/226007381_The_Ant_Colony_Optimization_Metaheuristic_Algorithms_Applications_and_Advances.

10. Yimeng Yue, Xin Wang. (2015). An Improved Ant Colony Optimization Algorithm for Solving TSP // International Journal of Multimedia and Ubiquitous Engineering Vol.10, No.12, pp.153-164. https://gvpress.com/journals/IJMUE/vol10_no12/16.pdf.
11. Сучасний підручник з JavaScript. *Сучасний підручник з JavaScript*. https://uk.javascript.info/.

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (29), 2025
ISSN 2663 - 4023

**VladyslavKhotynskyi**
student of the Faculty of Information Technologies and Mathematics
Lesya Ukrainka Volyn National University, Lutsk, Ukraine
*khotynskyi.vladyslav2022@vnu.edu.ua*

**Yuliia Pavlenko**
Senior Lecturerof the Department of Computer Science and Cybersecurity
Lesya Ukrainka Volyn National University, Lutsk, Ukraine
ORCID ID: 0000-0002-4065-045X
*Pavlenko.Yulya@vnu.edu.ua*

# ONE OF THE METHODS FOR IMPLEMENTING THE ANT ALGORITHM USING THE TRAVELING SALESMAN PROBLEM AS AN EXAMPLE

**Abstract.** In today's world, optimization problems play a crucial role in logistics, energy systems, industry, finance, healthcare, machine learning, and more. An optimization problem is a mathematical task aimed at finding the best possible solution from a set of feasible options, based on a defined criterion such as minimizing cost or time, or maximizing profit. There are exact methods, such as brute-force search and dynamic programming, which guarantee an optimal solution. However, they become computationally infeasible as the problem size grows. Heuristic and metaheuristic methods — including greedy algorithms, genetic algorithms, and ant colony optimization (ACO) — provide approximate but practically effective solutions for large-scale problems. One classical example of a combinatorial optimization problem is the Traveling Salesman Problem, where the goal is to find the shortest possible route that visits each city exactly once. The ant colony algorithm is one of the most efficient approaches for solving TSP and similar pathfinding problems on graphs. Inspired by the foraging behavior of real ants, the algorithm simulates how ants lay down pheromones along their paths and choose their routes based on the intensity of these pheromone trails and the distance to the next node. Each artificial ant incrementally constructs a solution, and the pheromone levels are updated after each iteration based on the quality of the solutions found. Key algorithm parameters (α, β, ρ) influence route selection, adaptability, and the ability to avoid outdated solutions. This paper presents the design and development of a software tool that generates a graph and implements the ACO algorithm to find the shortest path. The optimization algorithm is implemented using HTML, CSS, and JavaScript. One of the system's key features is real-time visualization: the graph is displayed using the Canvas element, where nodes are rendered as colored circles, and edges as lines whose thickness and transparency reflect the current pheromone intensity.

**Keywords:** optimization problems; ant colony algorithm; pheromones; software design; graph; shortest path.

## REFERENCES (TRANSLATED AND TRANSLITERATED)

1. An Overview of Ant Colony Optimization Algorithms for Dynamic Optimization Problems. *IntechOpen - Open Science Open Minds | IntechOpen*. https://www.intechopen.com/chapters/87285. (2023)
2. Nosenko V.A. (2018). Heuristic algorithms. Kharkiv: KhNU. 223 p.
3. Elitist Ant System | Algorithm Afternoon. *Home | Algorithm Afternoon*. https://algorithmafternoon.com/ants/elitist_ant_system/
4. Levchenko O.O. (2020). Optimization methods and algorithms. Dnipro: DNU. 340 p.
5. Ines Alaya, Christine Solnon, Khaled Ghedira. (2020). Ant algorithm for the multidimensional knapsack problem. International conference on Bioinspired Methods and their Applications (BIOMA), Ljubljana, Slovenia. pp.63-72. https://hal.science/hal-01541529/file/bioma04.pdf.
6. Gavrilyuk V.V. (2018). Algorithms and methods of computational intelligence. Lviv: Ivan Franko National University of Lviv. 312 p.
7. Corman T. H. (2001). Introduction to Algorithms. Cambridge, Massachusetts : The MIT Press. 1157 p.
8. Shtovba S. D., Rudy O. M. (2012). Ant optimization algorithms // Information technologies and computer technology. 2012. No. 78. P. 123-130.

9. Dorigo M., Stützle T. (2026). The ant colony optimization metaheuristic: Algorithms, applications, and advances. Handbook of metaheuristics. https://www.researchgate.net/publication/226007381_The_Ant_Colony_Optimization_Metaheuristic_Algorithms_Applications_and_Advances.
10. Yimeng Yue, Xin Wang. (2015). An Improved Ant Colony Optimization Algorithm for Solving TSP // International Journal of Multimedia and Ubiquitous Engineering Vol.10, No.12, pp.153-164. https://gvpress.com/journals/IJMUE/vol10_no12/16.pdf.
11. Сучасний підручник з JavaScript. *Сучасний підручник з JavaScript*. https://uk.javascript.info/.