МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

КРАВЧЕНКО ПАВЛО ПАВЛОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 20__ р.

ДОСЛІДЖЕННЯ МЕТАЕВРИСТИЧНИХ МЕТОДІВ РОЗВ'ЯЗАННЯ ОПТИМІЗАЦІЙНИХ ЗАВДАНЬ ПОШУКУ ОПТИМАЛЬНОГО МАРШРУТУ

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (магістерська) робота

Науковий керівник:
Федоров Є. Є., професор кафедри
інформаційних технологій
д-р техн. наук, професор

_____
(підпис)

Оцінка: _____/_____/_____
(бали за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____
(підпис)

Вінниця – 2024

АНОТАЦІЯ

Кравченко П.П. Дослідження метаевристичних методів розв'язання оптимізаційних завдань пошуку оптимального маршруту. Спеціальність 122 «Комп’ютерні науки», Освітня програма «Data science». Донецький національний університет імені Василя Стуса, Вінниця, 2023.

У магістерській роботі досліджені метаевристичні методи розв'язання оптимізаційних завдань пошуку оптимального маршруту. Показані алгоритми для оптимізації числових функцій, приклади алгоритмів на мові Python та отримані результати

112 с., 26 рис., 20 джерел.

Ключові слова: метаевристика, алгоритм, роєві біологічні мета евристики, мурашині алгоритми, оптимізація рою кішок, оптимізація рою частинок, ненатуральні метаевристики, натуральні метаевристики, ітеративний локальний пошук, пошук зі змінною околицею, жадібний рандомізований адаптивний пошук, мавповий пошук, імітація відпалу

ABSTRACT

Kravchenko P.P. Research of metaheuristic methods for solving optimization problems of finding the optimal route. Specialty 122 "Computer Science", Educational program "Data science". Vasyl' Stus Donetsk National University, Vinnytsia, 2023.

The qualification (master's) thesis investigates metaheuristic methods for solving optimization problems of finding the optimal route. The algorithms for optimizing numerical functions, examples of algorithms in Python and the results obtained are shown.

112p., 26 figures., 20 ref.

Keywords: metaheuristic, algorithm, swarm biological metaheuristics, ant algorithms, cat swarm optimisation, particle swarm optimisation, non-natural metaheuristics, natural metaheuristics, iterative local search, variable neighbourhood search, greedy randomised adaptive search, monkey search, annealing simulation

ЗМІСТ

ВСТУП

Актуальність теми полягає у тому, що на сьогоднішній день метаевристика актуальна для різних галузей науки та техніки, таких як оптимізація, штучний інтелект, машинне навчання та дослідження операцій. Метаевристика також корисна для вирішення складних і динамічних задач в умовах невизначеності і стохастичності.

Метаевристика постійно розвивається і включає нові ідеї та методи з інших дисциплін, таких як моделювання, навчання, паралельні обчислення та точні методи. Таким чином, метаевристика є не тільки потужним інструментом для вирішення складних задач оптимізації, але ще і багатою і активною областю досліджень, яка відкриває безліч можливостей.

Об’єктом дослідження в рамках магістерської роботи є процес пошуку оптимального маршруту

Предметом дослідження є метаевристичні алгоритми для пошуку оптимального маршруту

Метою роботи є дослідження існуючих метаевристичних алгоритмів для пошуку оптимального маршруту

Завдання дослідження:

- Дослідження та аналіз метаевристичних алгоритмів
- Створення та реалізація програми на основі метаевристичних алгоритмів

ПЕРЕДМОВА

В даний час швидко розвивається новий напрямок в теорії та практиці штучного інтелекту – метаевристики (або сучасні евристики). Евристика є алгоритмом, який знаходить "досить хороші" рішення складної проблеми за прийнятний час, теоретично не обґрунтовуючи їх правильність або оптимальність, тобто емпіричним шляхом. Термін "метаевристика" введений Гловером у 1986 році Метаевристика розширює можливості евристик, комбінуючи евристичні методи (процедури) на основі високорівневої стратегії (приставка «мета»).

Блюм і Ролі виділили дев'ять властивостей метаевристик:

1. Метаевристика є стратегією, яка управляє процесом пошуку.

2. Метою метаевристики є ефективне дослідження простору пошуку для знаходження (суб)оптимального рішення.

3. Методи, що використовуються метаевристичним алгоритмом, знаходиться в діапазоні від простого локального пошуку до складного процесу навчання.

4. Метаевристичний алгоритм є наближеним і зазвичай не детермінованим.

5. Метаевристика може використовувати механізм, запобігає потраплянню в пастку в обмеженій області простору пошуку.

6. Основні положення метаевристики допускають абстрактне опис.

7. Метаевристика не є проблемно-орієнтованою.

8. Метаевристика може використовувати проблемно орієнтовані знання у формі керованих евристик високорівневою стратегією.

9. Передові метаевристики використовують досвід, накопичений в процесі пошуку і представлений у вигляді пам'яті, для управління пошук.

Метаевристики за кількістю використовуваних рішень поділяються на:

- непопуляційні (використовують одне потенційне рішення);
- популяційні (використовують безліч потенційних рішення).

Непопуляційні метаевристики поділяються на:

- натуральні (біологічні та фізичні);

– ненатуральний.

Під ненатуральними метаевристиками авторами розуміються метаевристики, які не засновані на моделюванні процесів і механізмів біологічних або фізичних систем. Напроти натуральні евристики побудовані на базі моделей природних система.

Популяційні метаевристики поділяються на:

- еволюційні (детерміновані та ймовірнісні);

- роєві (біологічні та фізичні);

– імунний;

– ненатуральний.

Перші метаевристики з'являються в 1952 р., коли Робінсон і Монро розробили стохастичні оптимізаційні методи. В наприкінці 1950-х р. Растригін створив перші алгоритми випадкового пошуку для адаптивних систем управління. Перші ненатуральні метаевристики з'являються в 1977 р., коли Гловер розробив пошук з розкидом. У 1986 р. Гловер запропонував пошук із заборонами (табу пошук). У 1994 р. Батіті і Текіоллі створили реактивний пошук із заборонами. У 1995 р. Фео і Ресенде розробили жадібний рандомізований адаптивний пошук. У 1997 р Рубінштейн запропонував метод перехресної ентропії. В 1999 р. Тейлерд і Восс створили метаевристику часткової оптимізації при спеціальних умовах посилення.

Перші природні непопуляційні метаевристики з'являються в 1970 р., коли Гастінгс розробив алгоритм Метрополіса-Гастінгса. На основі цього алгоритму в 1983 р. Кірпатрік, Джелетт і Веччі запропонували імітацію відпалу. В 1999 р. Ботчер і Перкус створили екстремальну оптимізацію. В 2001 Джим, Кім та Лонгенатан запропонували гармонічний пошук. У 2007 р.Мучеріно і Сереф розробили мавпячий пошук. У 2011 р. Шах-Хоссейні створив галактичний пошуковий алгоритм.

Перші еволюційні метаевристики з'являються в 1954 р., коли Баррічеллі розробив першу модель процесу еволюції і використовував її для вирішення

загальних оптимізаційних проблем. В 1965 р. Рехенберг запропонував еволюційні стратегії. У 1966 р. Фогель, Оуенс і Уолш створили еволюційне програмування. В 1975 р. Голланд розробив перший генетичний алгоритм. У 1988 р. Коза зареєстрував перший патент з генетичного програмування. У 1989 р. Голдберг випустив фундаментальну книгу з генетичних алгоритмів. У 1989 р. компанією «Axcelis» створюється перше програмне забезпечення "Evolver", яке використовує генетичний алгоритм. У 1989 р Москато запропоновано міметичний алгоритм. У 1997 р. Сторн і Прайс розробили алгоритм диференціальної еволюції. У 1996 р. Мюхленбейном і Паабом створені перші ймовірнісні генетичні алгоритми (звані у літературі алгоритмами оцінювання розподілу або імовірнісними моделями побудови генетичних алгоритмів).

Перші ройові метаевристики з'являються в 1988, коли Мейсон і Мандерік опублікував роботу про колективну поведінку мураха. У 1992 р Дорого створив алгоритм мурашиної колонії. В 1995 р. Кеннеді та Еберхарт запропонували оптимізацію рою частинок. В 2002 р. Ліу і Пассіно розглянули моделі бактеріальних алгоритм. У 2004 р. Накрані і Тові розробили методи оптимізації на основі моделювання колонії бджіл. У 2005 р. Карабога створив алгоритм штучної колонії бджіл, а ФЕМ запропонував бджолиний алгоритм. У 2005 р. Крішнанад і Госе розробили оптимізацію на базі рою жуків-світлячків. У 2007 р. Шах-Хоссейні створив "інтелектуальні краплі води". У 2008 р. Янг запропонував світлячковий алгоритм, а в 2009 році-пошук кукушків.

Перші імунні метаевристики з'являються в 1988, коли Фармер, Пакард і Перельсон видали роботу по моделям штучної імунної системи. У 1996 р. Кастро і фон Зубен запропонували алгоритм клонального відбору, а Фармер та ін. створили алгоритм негативного відбору. У 2000 р. Тімміс, Ніл і Хант розробили штучну імунну мережу.

Метаевристики знаходять широке застосування для вирішення різних оптимізаційних проблем, машинного навчання, розпізнавання образів і ін. коли завдання не може бути вирішена іншими, більш простими методами, метаевристики часто можуть знайти оптимальні або близькі до них рішення. При

цьому обсяг обчислень може виявитися великим, але швидкість, з якою він зростає при збільшенні розмірності завдання зазвичай менше, ніж у інших відомих методів. Після того, як комп'ютерні системи стали досить швидкодіючими і недорогими, метаевристики перетворилися на один з основних інструментів пошуку (суб)оптимальних рішень задач, які до цього вважалися нерозв'язний.

В даний час в Україні дослідження з метаевристиків активно проводяться в Інституті кібернетики, Національному технічному університеті України "КПІ", Київському Національному університеті ім. Тараса Шевченка, Національному авіаційному університеті, Донецькому національному технічному університеті та ін.

Великою перешкодою до практичної реалізації більшості метаевристик є або їх Абстрактний опис (простий перерахування найменування етапів алгоритму) або опис, орієнтоване на вирішення лише однієї певної проблеми (наприклад, оптимізації або числових функцій або комбінаторної оптимізації). Крім того, програмна реалізація метаевристик часто наводиться на мовах програмування, не використовуваних більшістю вчених (наприклад, Ruby в монографії Браунлі), що ускладнює їх розуміння.

Складність і різноманітність досліджуваного матеріалу змусили авторів викласти тільки частина оптимізаційних проблем. У роботі для непопуляційних ненатуральних метаевристик наведено розроблені авторами алгоритми для оптимізації числових функцій (на прикладі мінімізації числової функції). Мавпячий пошук детально розглянуто для комбінаторної оптимізації (як приклад взята задача пошуку оптимального маршруту), і для оптимізації числових функцій (на прикладі мінімізації числової функції). У генетичному алгоритмі для оператора кросинговеру приведена розроблена авторами поєднання випадкового та лінійно впорядкованого відбору з імітацією відпалу, а для оператора редукції запропонована комбінація селективної схеми і випадкової схеми з імітацією відпал. Для алгоритму бінарного модельованого кросовера, алгоритму одновимірного граничного розподілу, компактного генетичного алгоритму,

максимізації спільної інформації для кластерів вхідного простору наведені розроблені авторамиалгоритми комбінаторної оптимізації (як приклад взято завдання пошуку оптимального маршруту). В алгоритмі клонального відбору для додавання нових антитіл авторами запропоновано варіант з імітацією відпалу, а для оператора редукції застосовується комбінація селективної схеми і випадкової схеми з імітацією відпалу.

Опис метаевристик, наведених в роботі скоротить час вивчення матеріалів і розширить можливості самостійного вдосконалення знань і придбання практичних навичок. Для найпопулярніших метаевристик (пошуку з заборонами, пошуку з розкидом, імітації відпалу, генетичного алгоритму, оптимізація рою частинок, мурашиної системи, алгоритму клонального відбору, алгоритму негативного відбору, штучної імунної мережі)

Пропонована робота містить основні положення сучасних метаевристик для оптимізації числових функцій і комбінаторної оптимізації. При її написанні використовувалися Сучасні зарубіжні та вітчизняні монографії, які наведені в загальному списку літератури, а також статті, які представлені в списку літератури до кожної з частин. В роботі використовуються матеріали курсу лекцій " Сучасні комп'ютерні технології в галузі", які читаються в Донецькій Академії Автомобільного Транспорту, та курсу лекцій "Еволюційні обчислення в технічних задачах", які читаються в Донецькому національному технічному університеті.

У першому розділі викладено основи ненатуральних непопуляційних та популяційних метаевристик - ітеративного локального пошуку, пошуку зі змінною околицею та жадібного рандомізованого адаптивного пошуку.

Другий розділ присвячений натуральним (біологічним і фізичним) непопуляційним метаевристикам - мавпячому пошуку та імітації відпалу;

У третьому розділі викладено основи роєвих біологічних метаевристик - оптимізація рою частин, оптимізація рою кішок, мурашині алгоритми;

РОЗДІЛ 1

НЕНАТУРАЛЬНІ МЕТАЕВРИСТИКИ

В даному розділі кваліфікаційної (магістерської) роботи розглядаються депопуляційні ненатуральні метаевристики:

- ітеративний локальний пошук;

- пошук зі змінною околицею;

- жадібний рандомізований адаптивний пошук;

1.1. Ітеративний локальний пошук

Ітеративний локальний пошук (iterated local search) був запропонований Штютцлом (Stutzle) [1]. Алгоритм складається з двох фаз – збурливий вплив і локальний пошук. Використання збурюючого впливу дозволяє уникнути локальних оптимумів при локальному пошуку. Занадто маленьке збурюючий вплив робить алгоритм жадібним[2]. Для завдання пошуку оптимального маршруту збурюючий вплив полягає у формуванні нового рішення шляхом перестановки " 4-opt " (поточне рішення випадковим чином розбивається на 4 частини, які стають в порядку 1,4,3,2), а локальний пошук полягає у формуванні нового рішення шляхом перестановки званої " 2-opt "(елементи нового рішення, розташовані між двома випадково вибраними вершинами, переставляються в зворотному порядку).

Ключові елементи ітеративного локального пошуку: [3]

1. Ініціалізація:

- ILS починається генерацією початкового рішення задачі оптимізації.

2. Локальний пошук:

- До поточного рішення застосовується алгоритм локального пошуку з метою знаходження локально оптимального рішення в області поточного.

3. Втручання:

- Після досягнення локального оптимуму вводиться втручання в рішення для виходу з локального оптимуму і вивчення іншої області простору рішень.

4. Критерій прийняття:

- Відновлене рішення оцінюється, і критерій прийняття визначає, чи приймається нове рішення як поточне.

5. Ітерація:

- Процес локального пошуку, втручання і прийняття повторюється протягом заданої кількості ітерацій або до досягнення критерію зупинки.

6. Механізм пам'яті:

- Деякі варіанти ILS включають механізм пам'яті для зберігання та використання перспективних рішень, знайдених під час пошуку.

Застосування ітеративного локального пошуку: [4]

1. Задачі комбінаторної оптимізації:

- ILS успішно застосовується до задач комбінаторної оптимізації, таких як Проблема Коміркової Торгівлі (TSP), Планування Завдань і Проблема Маршрутизації Транспорту (VRP).

2. Проблеми планування:

- В задачах планування ILS може бути використаний для пошуку ефективних розкладів для завдань з урахуванням обмежень та цілей.

3. Дизайн мереж:

- ILS має застосування в оптимізації дизайну та розташування мереж зв'язку та транспорту.

4. Машинне навчання:

- ILS використовується для налаштування гіперпараметрів та відбору ознак у завданнях машинного навчання, що покращує ефективність моделей.

5. Управління ланцюгом постачання:

- ILS використовується для оптимізації операцій ланцюга постачання, включаючи управління запасами та дистрибуцією.

Алгоритм для оптимізації числових функцій

1. Ініціалізація

1.1. Завдання параметрів δ^P та δ^{LS} для генерації нового рішення, причому $0 < \delta^P < 1, 0 < \delta^{LS} < 1, \delta^P < \delta^{LS}$

1.2. Завдання максимального числа ітерацій N_1, максимального числа ітерацій локального пошуку N_2, на яких не виявлено нове найкраще рішення; довжини рішення M, мінімальних і максимальних значень для вирішення $x_j^{min}, x_j^{max}, j \in \overline{1, M}.$

1.3. Завдання функції вартості (функція цілі)

$F(x) \to \frac{min}{x}$ де x – рішення

1.4. Створення випадковим чином кращого рішення

$x^* = (x_1^*, \dots, x_M^*), x_j^* = x_j^{min} + \left(x_j^{max} - x_j^{min}\right) rand()$

де $rand()$ - функція, що повертає рівномірно розподілене випадкове число в діапазоні [0,1]

1.5. Локальний пошук

1.5.1. m=0

1.5.2. Генерація рішення x від рішення x^*

1.5.2.1. $x_j = x_j^* + \delta\left(x_j^{max} - x_j^{min}\right)\left(-1 + 2\, rand(\ \ )\right), j \in \overline{1, M}.$

1.5.2.2. $x_j = \max\left\{x_j^{min}, x_j\right\}, x_j = \min\left\{x_j^{max}, x_j\right\}, j \in \overline{1, M}.$

1.5.3. Якщо $F(x) < F(x^*)$, то $x^* = x, m = 0$, інакше $m = m + 1$

1.5.4 Якщо m<N_2 то переходимо на шаг 1.5.2.

2. Номер ітерації n=1.

3. Виконання збурюючого впливу (генерація рішення $\hat{x}$від рішення x^*)

3.1. З безлічі номерів компонент рішення випадковим чином вибирається номер компоненти j^{cur} тобто $j^{cur} = round(1 + (M - 1)rand())$

3.2. $\hat{x}_{j^{cur}} = x^*_{j^{cur}} + \delta^P (x^{\max}_{j^{cur}} - x^{\min}_{j^{cur}})(-1 + 2rand())$

3.3. $\hat{x}_{j^{cur}} = \max\{x^{\min}_{j^{cur}}, \hat{x}_{j^{cur}}\},\ \hat{x}_{j^{cur}} = \min\{x^{\max}_{j^{cur}}, \hat{x}_{j^{cur}}\}$

4. Локальний пошук

4.1. m=0

4.2. Генерація рішення $\check{x}$ від рішення $\hat{x}$

4.2.1. $\check{x}_j = \hat{x}_j + \delta^{LS} (x_j^{\max} - x_j^{\min})(-1 + 2rand()),\ j \in \overline{1, M}$

4.2.2. $\check{x}_j = \max\{x_j^{\min}, \check{x}_j\},\ \check{x}_j = \min\{x_j^{\max}, \check{x}_j\},\ j \in \overline{1, M}$

4.3. Якщо $F(\check{x}) < F(\hat{x})$, то $\hat{x} = \check{x}, m = 0$, інакше $m = m + 1$

4.4. Якщо m<N_2 то переходимо на шаг 4.2.

5. Якщо $F(\hat{x}) < F(x^*)$, то $x^* = \hat{x}$,

6. Якщо m<N_1, то n=n+1 переходимо на шаг 3

Результатом є x^*

Алгоритм задачі на пошук оптимального маршруту

1. Ініціалізація

1.1. Встановлення максимальної кількості ітерацій N_1; Максимальна кількість ітерацій локального пошуку N_2, на яких не виявлено нового найкращого рішення. довжини вектору вершин M

1.2. Задається упорядкована множина вершин $V = \{1,...,M\}$ і матриця ваг ребер $[d_{ij}]\, i, j \in \overline{1,M}$.

1.3. Завдання функції вартості (функція цілі)

$$F(x) = d_{x_M, x_1} + \sum_{i=1}^{M-1} d_{x_i, x_{i+1}} \to \min_x$$

де d_{x_M, x_1} – вага ребра (x_i, x_{i+1}), $x_i, x_{i+1} \in V$, x – вектор вершин.

1.4. Задається, шляхом впорядкування випадковим чином безлічі, кращий вектор вершин x^*.

1.5. Локальний пошук на основі 2-opt

1.5.1. $m = 0$

1.5.2. Випадковим чином вибираються з вектора x^* дві вершини $c1$ і c2, причому вибір цих вершин триває до тих пір, поки не буде виконана умова

$1 < c2 - c1 < M - 1$

1.5.3. На основі вектора

$$x^* = (x_1^*,..., x_{c1-1}^*, x_{c1}^*,..., x_{c2}^*, x_{c2+1}^*,..., x_M^*)$$

Утворимо вектор

$$x = (x_1^*,..., x_{c1-1}^*, x_{c2}^*,..., x_{c1}^*, x_{c2+1}^*,..., x_M^*),$$

Тобто вершини $x_{c1}^*,..., x_{c2}^*$ переставляються в зворотному порядку.

1.5.4. Якщо, $F(x) < F(x^*)$ то , $x^* = x$, $m = 0$, інакше $m = m + 1$

1.5.5. $m < N_2$ Якщо , то перейдіть до кроку 1.5.2.

2. Номер ітерації. $n = 1$

3. Виконання обурюючого ефекту (генерація розчину $\hat{x}$ з x^* 4-оптного розчину)

3.1. Випадковим чином з вектора x^* вибираються три вершини:

$$c1 = 2 + (M/4) * rand(),\ c2 = c1 + 1 + (M/4) * rand(),$$

$$c3 = c2 + 1 + (M/4) * rand(),$$

де $rand()$ — функція, яка повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$

3.2. На основі вектора

$$x^* = (x_1^*, ..., x_{c1-1}^*, x_{c1}^*, ..., x_{c2-1}^*, x_{c2}^*, ..., x_{c3-1}^*, x_{c3}^*, ..., x_M^*)$$

Створюється вектор

$$\hat{x} = (x_1^*, ..., x_{c1-1}^*, x_{c3}^*, ..., x_{cM}^*, x_{c2}^*, ..., x_{c3-1}^*, x_{c1}^*, ..., x_{c2-1}^*)$$

4. Локальний пошук на основі 2-opt

4.1. $m = 0$

4.2. Випадковим чином обираймо з вектора $\hat{x}$ дві вершини c1 і c2, причому вибір цих вершин триває до тих пір, поки не буде виконано умову $1 < c2 - c1 < M - 1$

4.3. На основі вектора

$$\hat{x} = (\hat{x}_1, ..., \hat{x}_{c1-1}, \hat{x}_{c1}, ..., \hat{x}_{c2}, \hat{x}_{c2+1}, ..., \hat{x}_M)$$

Утворимо вектор

$$\check{x} = (\hat{x}_1, ..., \hat{x}_{c1-1}, \hat{x}_{c2}, ..., \hat{x}_{c1}, \hat{x}_{c2+1}, ..., \hat{x}_M)$$

Тобто вершини $\hat{x}_{c1}, ..., \hat{x}_{c2}$ переставляються в зворотному порядку.

4.5. Якщо $F(\check{x}) < F(\hat{x})$, то $\hat{x} = \check{x}, m = 0$, інакше $m = m + 1$

4.6. Якщо m<N_2 то переходимо на шаг 4.2.

5. Якщо $F(\hat{x}) < F(x^*)$, то $x^* = \hat{x}$,

6. Якщо m<N_1, то n=n+1 переходимо на шаг 3

Результатом є x^*

Приклад використання ітеративного локального пошуку на Python

Пошук мінімуму функції однієї змінної.

Наступний приклад реалізує ітеративний локальний пошук для пошуку мінімуму функції f (x) = x^2:

```python
import numpy as np

def initial_solution():
    # Генерує випадкове початкове значення x у діапазоні [-5, 5]
    return np.random.uniform(-5, 5)

def local_search(solution):
    # Виконує локальний пошук - у цьому випадку не робить нічого
    return solution

def perturbation(solution):
    # Вносить випадкову зміну до поточного рішення
    return solution + np.random.uniform(-1, 1)

def accept_criteria(old_solution, new_solution):
    # Приймає нове рішення, якщо значення функції в новому менше
    return new_solution < old_solution

def termination_criteria_not_met(iteration):
    # Умова завершення: кількість ітерацій не перевищує 100
    return iteration < 100

def iterative_local_search():
    # Початкове рішення
    current_solution = initial_solution()

    # Головний цикл ітеративного локального пошуку
    iteration = 0
    while termination_criteria_not_met(iteration):
        # Локальний пошук
        local_solution = local_search(current_solution)

        # Вносимо випадкову зміну
        perturbed_solution = perturbation(local_solution)

        # Порівнюємо рішення та приймаємо його за умови
        if accept_criteria(current_solution, perturbed_solution):
            current_solution = perturbed_solution

        # Збільшуємо лічильник ітерацій
        iteration += 1

    return current_solution

# Запускаємо ітеративний локальний пошук
final_solution = iterative_local_search()

# Виводимо результати
print("Оптимізоване рішення:", final_solution)
print("Значення функції при оптимізованому рішенні:", final_solution**2)
```

Рис.1.1 ітеративний локальний пошук

```
Оптимізоване рішення: -21.791826603841933
Значення функції при оптимізованому рішенні: 474.88370673191304
```

Рис.1.2 Результат ітеративного локального пошуку

У цьому прикладі функція initial_solution () генерує випадкове початкове значення змінної x у заданому діапазоні. local_search (solution) в даному прикладі не виконує ніяких додаткових дій, тому вона повертає передане рішення без змін. perturbation (solution) вносить випадкову зміну поточного рішення, додаючи випадкове значення з діапазону [-1, 1] до поточного значення x. accept_criteria (old_solution, new_solution) визначає, чи слід приймати нове рішення на основі порівняння значення функції в новому та старому рішенні (у цьому випадку, якщо значення функції в новому рішенні менше). termination_criteria_not_met (iteration) визначає, чи виконалася умова закінчення пошуку (в даному випадку, якщо кількість ітерацій не перевищує 100).

Запуск цього коду дозволить знайти мінімум функції f (x) = x^2 за допомогою інтерактивного локального пошуку.

1.2. Пошук зі змінною околицею

Пошук зі змінною околицею (variable neighborhood search) запропонований Младеновичем (Mladenović) і Хансеном (Hansen) [5] і використовує локальний пошук в ростущій околиці.

Даний алгоритм заснований на трьох принципах:

- локальний мінімум для одного району, можливо, не локальний мінімум для іншого району;

- глобальний мінімум є локальним мінімумом для всіх можливих околиць;

- локальні мінімуми відносно близько до глобальних мінімумів.

Для оптимізації числових функцій кожна околиця пов'язана з певною компонентою рішення.

Метод був вперше запропонований Младеновичем та Гансеном в 1997 році. [6]Основна ідея VNS полягає в ітеративному дослідженні різних околиць поточного рішення. Алгоритм підтримує баланс між диверсифікацією та інтенсифікацією, систематично змінюючи рівень деталізації простору пошуку.

На кожній ітерації алгоритм виконує наступні кроки: [7]

1. Локальний пошук:

- Починаємо з початкового рішення і застосовуємо локальний пошук в межах конкретної околиці.

2. Трясіння (Shaking):

- Пертурбуємо поточне рішення за допомогою процедури "трясіння", щоб дослідити нове рішення в іншій околиці.

3. Локальний пошук в новій околиці:

- Застосовуємо локальний пошук до пертурбованого рішення в іншій околиці.

4. Критерій переміщення або прийняття:

- Переходимо до нового рішення на основі певних критеріїв переміщення, які можуть включати порівняння якості поточного та пертурбованого рішень.

5. Ітерація:

- Повторюємо процес протягом заданої кількості ітерацій або до досягнення критерію зупинки. [8]

Алгоритм задачі на пошук оптимального маршруту

1. Ініціалізація

1.1. Встановлення параметра δ для генерації нового рішення, при тому $0 < \delta < 1$.

1.2. Встановлення максимальної кількісті ітерацій N_1; максимальної кількості ітерацій локального пошуку N_2, на яких не знайдено нового кращого рішення; максимального розміру околиці N_3; довжину рішення M; мінімальних і максимальних значення для рішення $x_j^{\min}, x_j^{\max}$. $j \in \overline{1, M}$

1.3. Встановлення функції вартості (функція цілей)

$F(x) \to \min\limits_{x}$, де x рішення.

1.4. Випадкове створення кращого рішення

$$x^* = (x_1^*, ..., x_M^*),\ x_j^* = x_j^{\min} + (x_j^{\max} - x_j^{\min}) rand(),$$

де $rand()$ — функція, яка повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$

2. Номер ітерації $n = 0$.

3. Розмір околиці $Z = 1$

4. Створіти околицю U рішення x^*

4.1. $z = 1$

4.2. Генерація рішення x_z з рішення x^*

4.2.1. $x_{zj} = x_j^* + \delta(x_j^{\max} - x_j^{\min})(-1 + 2rand()),\ j \in \overline{1,M}$

4.2.2. $x_{zj} = \max\{x_j^{\min}, x_{zj}\}$, $x_{zj} = \min\{x_j^{\max}, x_{zj}\},\ j \in \overline{1,M}$

4.3. Якщо , $x_z \notin U$ то , $U = U \bigcup \{x_z\}$ $z = z + 1$

4.4. Якщо $z \le Z$, то перейдіть до кроку 4.2.

5. Випадковим чином обираемо з околиці U вектор $\hat{x}$

6. Локальний пошук

6.1. $m = 0$

6.2. Генерація рішення $\check{x}$ від рішення $\hat{x}$

6.2.1. $\check{x}_j = \hat{x}_j + \delta(x_j^{\max} - x_j^{\min})(-1 + 2rand()),\ j \in \overline{1,M}$

6.2.2. $\check{x}_j = \max\{x_j^{\min}, \check{x}_j\},\ \check{x}_j = \min\{x_j^{\max}, \check{x}_j\},\ j \in \overline{1,M}$

6.3. Якщо $F(\check{x}) < F(\hat{x})$, то $\hat{x} = \check{x}, m = 0$, інакше $m = m + 1$

6.4. Якщо m<N_2 то переходимо на шаг 6.2.

7. Якщо $F(\hat{x}) < F(x^*)$, то $x^* = \hat{x}$, n=0, то переход до шагу 3, інакше $n = n + 1$

8. Якщо Z<N_1, то Z=Z+1 переходимо на шаг 4

9. Якщо $n < N_1$, то переход до шагу 3

Результатом є x^*

Алгоритм задачі на пошук оптимального маршруту

1. Ініціалізація

1.1. Встановлення максимальної кількості ітерацій N_1; Максимальної кількісті ітерацій локального пошуку N_2, на яких не виявлено нового найкращого рішення. максимальний розмір околиці N_3; довжина вектора вершин M .

1.2. Встановлюються упорядкована множина вершин $V = \{1,...,M\}$ і матриця ваг ребер $[d_{ij}]\, i, j \in \overline{1,M}$

1.3. Встановлення функції вартості (функція цілей)

$$F(x) = d_{x_M, x_1} + \sum_{i=1}^{M-1} d_{x_i, x_{i+1}} \to \min_x,$$

де d_{x_M, x_1} - вага ребра (x_i, x_{i+1}), $x_i, x_{i+1} \in V$, x - вектор вершин.

1.4. Задається, шляхом випадкового впорядкування множини V, найкращий вектор вершин . x^*

2. Номер ітерації. $n = 0$

3. Розмір околиці $Z = 1$

4. Створіть околицю U рішення x^* на основі 2-opt

4.1. $z = 1$

4.2. Дві вершини $c1$ і $c2$ з вектора x^*, вибираються випадковим чином, при тому вибір цих вершин триває до пір поки не будуть виконанні умови $1 < c2 - c1 < M - 1$

4.3. На основі вектора

$$x^* = (x_1^*, ..., x_{c1-1}^*, x_{c1}^*, ..., x_{c2}^*, x_{c2+1}^*, ..., x_M^*)$$

Створюється вектор

$$x_z = (x_1^*, ..., x_{c1-1}^*, x_{c2}^*, ..., x_{c1}^*, x_{c2+1}^*, ..., x_M^*)$$

тобто вершини переставляються в зворотному порядку. $x_{c1}^*, ..., x_{c2}^*$

4.4. Якщо $x_z \notin U$, то $U = U \bigcup \{x_z\}$, $z = z + 1$

4.5. Якщо $z \le Z$, то перейдіть до кроку 4.2.

5. Випадково чином обираймо з околиці U вектор $\hat{x}$

6. Локальний пошук на основі 2-опт

6.1. $m = 0$

6.2. Дві вершини $c1$ і $c2$, вибираються випадковим чином з вектора $\hat{x}$, при тому вибір цих вершин триває до пір поки не будуть виконанні умови $1 < c2 - c1 < M - 1$

6.3. На основі вектора

$$\hat{x} = (\hat{x}_1, \ldots, \hat{x}_{c1-1}, \hat{x}_{c1}, \ldots, \hat{x}_{c2}, \hat{x}_{c2+1}, \ldots, \hat{x}_M)$$

створюймо вектор

$$\breve{x} = (\hat{x}_1, \ldots, \hat{x}_{c1-1}, \hat{x}_{c2}, \ldots, \hat{x}_{c1}, \hat{x}_{c2+1}, \ldots, \hat{x}_M)$$

тобто вершини переставляються в зворотному порядку. $x^*_{c1}, \ldots, x^*_{c2}$

6.4. Якщо $F(\breve{x}) < F(\hat{x})$, то $\hat{x} = \breve{x}, m = 0$, інакше $m = m + 1$

6.5. Якщо m<N_2 то переходимо на шаг 6.2.

7. Якщо $F(\hat{x}) < F(x^*)$, то $x^* = \hat{x}$, n=0, то переход до шагу 3, інакше

$n = n + 1$

8. Якщо Z<N_3, то Z=Z+1 переходимо на шаг 4

9. Якщо $n < N_1$, то переход до шагу 3

Результатом є x^*

Різновидами даної метаевристики є асиметричний пошук зі змінною околицею, декомпозиційний пошук зі змінною околицею, паралельний пошук зі змінною кратністю та інші.

Приклад пошуку зі змінною околицею Python

У цьому прикладі буде використовуватися випадковий пошук в околиці, що змінюється:

```python
import numpy as np

def objective_function(x):
    return x**2

def generate_neighborhood(x, radius=1.0):
    # Генерує випадкове значення в околиці поточного рішення x
    return x + np.random.uniform(-radius, radius)

def variable_neighborhood_search():
    # Початкове рішення
    current_solution = np.random.uniform(-5, 5)

    # Початковий радіус околиці
    initial_radius = 1.0

    # Кількість ітерацій
    max_iterations = 100

    # Головний цикл Variable Neighborhood Search
    for iteration in range(max_iterations):
        # Генеруємо випадкове значення в поточній околиці
        neighbor_solution = generate_neighborhood(current_solution, radius=initial_radius)

        # Обчислюємо значення функції для поточного та сусіднього рішення
        current_value = objective_function(current_solution)
        neighbor_value = objective_function(neighbor_solution)

        # Порівнюємо значення та приймаємо нове рішення, якщо воно краще
        if neighbor_value < current_value:
            current_solution = neighbor_solution

        # Зменшуємо радіус околиці для більш точного локального пошуку
        initial_radius *= 0.9

    return current_solution

# Запускаємо Variable Neighborhood Search
final_solution = variable_neighborhood_search()

# Виводимо результати
print("Оптимізоване рішення:", final_solution)
print("Значення функції при оптимізованому рішенні:", objective_function(final_solution))
```

```
Оптимізоване рішення: 1.3805113999348814e-06
Значення функції при оптимізованому рішенні: 1.905811725350166e-12
```

Рис 1.3 пошук зі змінною околицею

У цьому коді використовується простий підхід до Variable Neighborhood Search з випадковим пошуком в околиці, що змінюється. Функція generate_neighborhood генерує випадкове значення в околиці поточного рішення, а основний цикл робить спроби покращити рішення протягом певної кількості ітерацій.

1.3. Жадібний рандомізований адаптивний пошук

Жадібний рандомізований адаптивний пошук (greedy randomized adaptive search) був запропонований Фео (Feo) та Ресенде (Resende)[9]. Алгоритм складається з двох фаз – жадібнарандомізована процедура та локальний пошук[10]. Мета алгоритму полягає в тому, щоб неодноразово генерувати

випадкові жадібні рішення, а потім використовувати локальний пошук, щоб наблизити ці рішення до локальних оптимумів[11].

Алгоритм складається з наступних кроків:

1. Конструкція Рішення:

Вибір жадібного рішення на основі локального пошуку та випадкового фактора.

2. Оцінка та Покращення:

Оцінка якості вибраного рішення та його подальше покращення за допомогою локального пошуку.

3. Оновлення Параметрів:

Адаптивне оновлення параметрів алгоритму для оптимізації здатності вибору рішення.

4 . Ітерація:

Повторення процесу протягом заданої кількості ітерацій або до досягнення критерію зупинки[12].

Алгоритм для оптимізації числових функцій

1. Ініціалізація

1.1. Завдання параметра жадібності α для жадібної рандомізованої процедури, параметра δ для генерації нового рішення, причому

$\alpha \in [0,1]$(при $\alpha = 0$ максимальна жадібність),$0 < \delta < 1$

1.2. Завдання максимальної кількості ітерацій N_1; максимальної кількості ітерацій локального пошуку N_2, на яких не виявляється нове найкраще рішення; довжини вектор вершин M.

1.3. Завдання функції вартості (функція цілі)

$F(x) \to \min\limits_{x}$,

деx- Рішення.

1.4. Створення випадковим чином найкращого рішення

$x^* = (x_1^*,\dots, x_M^*), x_j^* = x_j^{\min} + (x_j^{\max} - x_j^{\min})\text{rand}()$,

де rand() - Функція, що повертає рівномірно розподілене випадкове число в діапазоні$[0,1]$

2. Номер ітерації $n = 1$.

3. Створення випадковим чином рішенняx

$x = (x_1,\ldots, x_M), x_j = x_j^{\min} + (x_j^{\max} - x_j^{\min})\text{rand}()$,

де rand()- Функція, що повертає рівномірно розподілене випадкове число в діапазоні$[0,1]$

4. Виконання жадібної рандомізованої процедури

4.1. З множини номерів компонент рішення випадковим чином вибирається номер компоненти j^{cur}, тобто. $j^{\text{cur}} = \text{round}(1 + (M-1)\text{rand}())$, ініціалізація безлічі заборонених номерів компонентів $V^{\text{tabu}} = \{j^{\text{cur}}\}$, кількість заборонених компонент $l = 1$

4.2. Створення безлічі дозволених номерів компонентів $\widetilde{V} = V \setminus V^{tabu}$, ініціалізація безлічі обмежених кандидатів$V^{\text{RCL}} = \emptyset$, номер компоненти$j = 1$

4.3. Якщо$j \in V^{\text{tabu}}$, то перехід на крок 4.5

4.4. Локальний пошук

4.4.1.$m = 0$

4.4.2. Генерація рішення$\check{x}$від рішенняx

4.4.2.1.$\check{x}_j = x_j + \delta(x_j^{\max} - x_j^{\min})(-1 + 2\text{rand}())$

4.4.2.2.$\check{x}_j = \max\{x_j^{\min}, \check{x}_j\}, \check{x}_j = \min\{x_j^{\max}, \check{x}_j\}$

4.4.3. Якщо$F(\check{x}) < F(x)$, то$y_j = F(\check{x}), z_j = \check{x}_j, m = 0$інакше$m = m + 1$

4.4.4. Якщо$m < N_2$, Перехід на крок 4.4.2.

4.5. Якщо$j < M$, то$j = j + 1$, перехід на крок 4.3

4.6. $y^{\min} = \min\limits_{j \in \not\nabla} y_j, y^{\max} = \max\limits_{j \in \not\nabla} y_j$

4.7. $j = 1$

4.8. Якщо$g_j \le y^{\min} + \alpha(y^{\max} - y^{\min})$, то$V^{\text{RCL}} = V^{\text{RCL}} \cup \{j\}$

4.9. Якщо$j < |\not\nabla|$, то$j = j + 1$, перехід до кроку 4.8

4.10. З безлічіV^{RCL}випадково вибирається номер компонентиj^{cur}, тобто.$j^{\text{cur}} = \text{round}(1 + (M-1)\text{rand}()), V^{\text{tabu}} = V^{\text{tabu}} \cup \{j^{\text{cur}}\}, x_{j^{\text{cur}}} = z_{j^{\text{cur}}}$

4.12. Якщо$l < M$, то$l = l + 1$, перехід на крок 4.2

5. Локальний пошук

5.1.$m = 0$

5.2. Генерація рішення$\check{x}$від рішенняx

5.2.1.$\check{x}_j = x_j + \delta(x_j^{\max} - x_j^{\min})(-1 + 2\text{rand}()), j \in \overline{1, M}$

5.2.2.$\check{x}_j = \max\{x_j^{\min}, \check{x}_j\}, \check{x}_j = \min\{x_j^{\max}, \check{x}_j\}, j \in \overline{1, M}$

5.3. Якщо $F(\check{x}) < F(x)$, то$x = \check{x}, m = 0$інакше$m = m + 1$

5.4. Якщо $m < N_2$, Перехід на крок 5.2.

6. Якщо $F(x) < F(x^*)$, то $x^* = x$

7. Якщо $n < N_1$, то $n = n + 1$, перехід до кроку 3

Результатом є x^*.

Алгоритм для пошуку оптимального маршруту

1. Ініціалізація

1.1. Завдання параметра жадібностіαдля жадібної рандомізованої процедури, причому$\alpha \in [0,1]$(при $\alpha = 0$ максимальна жадібність)

1.2. Завдання максимальної кількості ітерацій N_1; максимальної кількості ітерацій локального пошуку N_2, на яких не виявляється нове найкраще рішення; довжини вектор вершин M.

1.3. Встановлюється безліч вершин $V = \{1,..., M\}$ та матриця ваг ребер $[d_{\text{ij}}], i, j \in \overline{1, M}$.

1.4. Завдання функції вартості (функція цілі)

$F(x) = d_{x_M, x_1} + \sum_{i=1}^{M-1} d_{x_i, x_{i+1}} \to \min\limits_{x},$

де d_{x_M, x_1}- вага ребра $(x_i, x_{i+1}), x_i, x_{i+1} \in V$,

x- вектор вершин.

1.5. Задається, шляхом упорядкування випадковим чином безлічіV, кращий вектор вершинx^*.

2. Номер ітерації$n = 1$.

3. Виконання жадібної рандомізованої процедури

3.1. З безлічі вершинVвипадковим чином вибирається вершина v^{cur}, ініціалізація безлічі заборонених вершин $V^{\text{tabu}} = \{v^{\text{cur}}\}$, кількість заборонених вершин $l = 1, x_1 = v^{\text{cur}}$

3.2. Створення безлічі дозволених вершин $\widetilde{V} = V \setminus V^{tabu}$, ініціалізація безлічі обмежених кандидатів $V^{\text{RCL}} = \emptyset$, номер вершини $j = 1$

3.3.$d^{\min} = \min\limits_{s \in \overline{1,|\mathcal{V}|}} d_{v^{\text{cur}},\mathcal{v}_s}, d^{\max} = \max\limits_{s \in \overline{1,|\mathcal{V}|}} d_{v^{\text{cur}},\mathcal{v}_s},$

3.4. Якщо $d_{v^{\text{cur}},\mathcal{v}_j} \le d^{\min} + \alpha(d^{\max} - d^{\min})$, то$V^{\text{RCL}} = V^{\text{RCL}} \cup \{\mathcal{v}_j\}$

3.5. Якщо $j < |\mathcal{V}|$, то$j = j + 1$, перехід до кроку 3.3

3.6. З безлічі V^{RCL}випадковим чином вибирається вершина v^{cur},$V^{\text{tabu}} = V^{\text{tabu}} \cup \{v^{\text{cur}}\}$,$x_{l+1} = v^{\text{cur}}$

3.7. Якщо $l < M$, то $l = l + 1$, перехід на крок 3.2

4. Локальний пошук на основі 2-opt

4.1.$m = 0$

4.2. Випадково вибираються з вектораxдві вершини $c1$ і $c2$, причому вибір цих вершин продовжується доти, доки не буде виконано умову

$$1 < c2 - c1 < M - 1$$

4.3. На основі вектора

$x = (x_1,\ldots, x_{c1-1}, x_{c1},\ldots, x_{c2}, x_{c2+1},\ldots, x_M)$

створюється вектор

$\check{x} = (x_1,\ldots, x_{c1-1}, x_{c2},\ldots, x_{c1}, x_{c2+1},\ldots, x_M)$

тобто. вершини $x_{c1},\ldots, x_{c2}$ переставляються у зворотному порядку.

4.4. Якщо $F(\check{x}) < F(x)$, то$x = \check{x}$,$m = 0$інакше$m = m + 1$

4.5. Якщо $m < N_2$, Перехід на крок 4.2.

5. Якщо $F(x) < F(x^*)$, то$x^* = x$

6. Якщо $n < N_1$, то $n = n + 1$, перехід до кроку 3

Результатом є x^*

Приклад реалізації GRASP для розв'язання задачі комівояжера на мові програмування Python

```python
import numpy as np

def euclidean_distance(point1, point2):
    # Обчислює відстань між двома точками у просторі
    return np.linalg.norm(point1 - point2)

def total_distance(tour, distances):
    # Обчислює загальну відстань між містами у турі
    total = 0
    for i in range(len(tour) - 1):
        total += distances[tour[i]][tour[i+1]]
    total += distances[tour[-1]][tour[0]]  # Замикання циклу
    return total

def construct_greedy_solution(distances):
    # Жадібне побудова початкового рішення
    num_cities = len(distances)
    unvisited_cities = set(range(num_cities))
    tour = [np.random.choice(list(unvisited_cities))]  # Вибираємо випадкове початкове місто

    while unvisited_cities:
        current_city = tour[-1]
        next_city = min(unvisited_cities, key=lambda city: distances[current_city][city])
        tour.append(next_city)
        unvisited_cities.remove(next_city)

    return tour

def local_search(tour, distances):
    # Локальний пошук для оптимізації туру
    improved = True
    while improved:
        improved = False
        for i in range(1, len(tour) - 1):
            for j in range(i+1, len(tour)):
                new_tour = tour[:i] + tour[i:j][::-1] + tour[j:]
```

```
                if total_distance(new_tour, distances) < total_distance(tour, distances):
                    tour = new_tour
                    improved = True
                    break
            if improved:
                break
    return tour

def grasp(num_iterations, distances):
    best_tour = None
    best_distance = float('inf')

    for _ in range(num_iterations):
        current_tour = construct_greedy_solution(distances)
        current_tour = local_search(current_tour, distances)
        current_distance = total_distance(current_tour, distances)

        if current_distance < best_distance:
            best_tour = current_tour
            best_distance = current_distance

    return best_tour, best_distance

# Приклад використання:
# Представимо матрицю відстаней між містами
cities = np.array([[0, 2, 9, 10],
                   [1, 0, 6, 4],
                   [15, 7, 0, 8],
                   [6, 3, 12, 0]])

# Запускаємо GRASP
best_tour, best_distance = grasp(num_iterations=10, distances=cities)

# Виводимо результат
print("Найкращий тур:", best_tour)
print("Найкраща відстань:", best_distance)
```

Рис. 1.4 жадібний рандомізований адаптивний пошук

```
Найкращий тур: [1, 0, 1, 2, 3]
Найкраща відстань: 20
>
```

Рис. 1.5 Результат жадібного рандомізованого адаптивного пошуку

У цьому прикладі grasp викликається з кількістю ітерацій та матрицею відстаней між містами. Функції construct_greedy_solution та local_search відповідають за побудову початкового рішення та локальний пошук відповідно.

РОЗДІЛ 2

НАТУРАЛЬНІ НЕПОПУЛЯЦІЙНІ МЕТАЕВРИСТИКИ

2.1. Мавповий пошук

Мавповий пошук(monkey search)запропонований Мучеріно (Mucherino) [13] і заснований наповедінці мавпи, яка піднімається по дереву у пошуках їжі. При цьому мавпа пам'ятає, де минулого разу була гарна їжа.

У цьому алгоритмі вузлами дерева є рішення. На кожній ітерації будується нове дерево.

Алгоритм для оптимізації числових функцій

1. Ініціалізація

1.1. Завдання параметрів мутації або кросинговера, ймовірності переміщення по дереву p^{move}

1.2. Завдання максимальної кількості дерев N, максимальної кількості рівнів кожного дерева N_{level}, довжини рішення M, мінімальних та максимальних значень для вирішення $x_j^{\min}, x_j^{\max}, j \in \overline{1, M}$.

1.3. Завдання функції вартості (функція цілі)

$F(x) \to \min\limits_{x},$

деx- Рішення.

1.4. Створення випадковим чином оптимального рішення у вигляді речового вектора

$x^* = (x_1^*,\ldots, x_M^*), x_j^* = x_j^{\min} + (x_j^{\max} - x_j^{\min})\text{rand}(),$

де rand()- Функція, що повертає рівномірно розподілене випадкове число в діапазоні[0,1]

1.5. Задається поточне рішення x^{cur}, причому$x^{\text{cur}} = x^*$.

2. Номер дерева $n = 1$

3. Створення n-го дерева

3.1. Безліч вершин $V_s = \emptyset$

3.2. Безліч ребер $A_s = \emptyset$

3.3. $l = 1$

3.4. Створення рішення $x1$ за допомогою операторів генетичного алгоритму - мутації (використовується рішення x^{cur}) або кросинговера (використовуються рішенняx^{cur} і x^*)

3.5. Створення рішення$x2$за допомогою операторів генетичного алгоритму - мутації (використовується рішення x^{cur}) або кросинговера (використовуються рішення x^{cur} і x^*)

3.6. Якщо $F(x1) < F(x^*)$, то $x^* = x1$

3.7. Якщо $F(x2) < F(x\ \ )$, то $x\ \ = x2$

3.8.$V_s = V_s \cup \{x1, x2\}$

3.9.$A_s = A_s \cup \{(x^{\text{cur}}, x1), (x^{\text{cur}}, x2)\}$

3.10. Обчислити вагу ребра $(x^{\text{cur}}, x1)$

$w_{(x^{\text{cur}},x1)} = F(x^{\text{cur}}) - F(x1)$

3.11. Обчислити вагу ребра $(x^{\text{cur}}, x2)$

$w_{(x^{\text{cur}},x2)} = F(x^{\text{cur}}) - F(x2)$

3.12. Вибрати ребро з найкращою вагою$w_{l+1,l}$

$$w_{l+1,l} = \begin{cases} w_{(x^{\text{cur}},x1)}, & w_{(x^{\text{cur}},x1)} w_{(x^{\text{cur}},x2)} \\ w_{(x^{\text{cur}},x2)}, & w_{(x^{\text{cur}},x1)} w_{(x^{\text{cur}},x2)} \end{cases}$$

3.13. Модифікувати поточне рішення

$$x^{\text{cur}} = \begin{cases} x1, & F(x1) < F(x2) \\ x2, & F(x1) > F(x2) \end{cases}$$

3.14. Якщо $l < N_{\text{level}}$, то $l = l + 1$, перехід на крок 3.4

4. Зміна ваги при переміщенні по дереву від термінальної вершини до кореня по ребрах з найкращою вагою

4.1. $w^* = w_{l,l-1}$

4.2. $l = l - 1$

4.3. $w^{\text{cur}} = w_{l,l-1}$

4.4. Якщо $w^{\text{cur}} > w^*$, то$w^* = w^{\text{cur}}$ інакше $w_{l,l-1} = w^*$

4.5. Якщо $l > 1 \wedge \text{rand}() < p^{\text{move}}$, то перехід на крок 4.2

5. Переміщення по дереву від поточної вершини (в силу $\text{rand}() < p^{\text{move}}$ це не обов'язково корінь) до термінальної вершини по ребрах з найкращою вагою

6. Якщо $n < N$, то $n = n + 1$, перехід до кроку 3

Результатом є x^*.

Алгоритм для пошуку оптимального маршруту

1. Ініціалізація

1.1. Завдання ймовірності переміщення по дереву p^{move}.

1.2. Завдання максимальної кількості дерев N, максимальної кількості рівнів кожного дерева N_{level}, довжини рішення (вектор вершин) M

1.3. Встановлюється безліч вершин $V = \{1,\dots, M\}$ та матриця ваг ребер $[d_{\text{ij}}]$, $i, j \in \overline{1, M}$.

1.4. Завдання функції вартості (функція цілі)

$F(x) = d_{x_M, x_1} + \sum_{i=1}^{M-1} d_{x_i, x_{i+1}} \to \min\limits_{x}$,

де d_{x_M, x_1}- Вага ребра (x_i, x_{i+1}), $x_i, x_{i+1} \in V$,

x- Рішення (вектор вершин).

1.5. Задається, шляхом упорядкування випадковим чином безлічі V, оптимальне рішення (вектор вершин)x^*.

1.6. Задається поточне рішення x^{cur}, причому $x^{\text{cur}} = x^*$.

2. Номер дерева$n = 0$

3. Створення n-го дерева

3.1. Безліч вершин $V_s = \emptyset$

3.2. Безліч ребер $A_s = \emptyset$

3.3. $l = 1$

3.4. Створення рішення $x1$ за допомогою операторів генетичного алгоритму - мутації на основі 2-opt (використовується рішення x^{cur}) або частково відповідного кросинговеру (використовуються рішення x^{cur} і x^*)

3.5. Створення рішення $x2$ за допомогою операторів генетичного алгоритму - мутації на основі 2-opt (використовується рішення x^{cur}) або частково відповідного кросинговеру (використовуються рішенняx^{cur} і x^*)

3.6. Якщо $F(x1) < F(x^*)$, то $x^* = x1$

3.7. Якщо $F(x2) < F(x^*)$, то $x^* = x2$

3.8. $V_s = V_s \cup \{x1, x2\}$

3.9. $A_s = A_s \cup \{(x^{\text{cur}}, x1), (x^{\text{cur}}, x2)\}$

3.9. Обчислити вагу ребра $(x^{\text{cur}}, x1)$

$$w_{(x^{\text{cur}},x1)} = F(x^{\text{cur}}) - F(x1)$$

3.10. Обчислити вагу ребра $(x^{\text{cur}}, x2)$

$$w_{(x^{\text{cur}},x2)} = F(x^{\text{cur}}) - F(x2)$$

3.11. Вибрати ребро з найкращою вагою $w_{l+1,l}$

$$w_{l+1,l} = \begin{cases} w_{(x^{\text{cur}},x1)}, & w_{(x^{\text{cur}},x1)} w_{(x^{\text{cur}},x2)} \\ w_{(x^{\text{cur}},x2)}, & w_{(x^{\text{cur}},x1)} w_{(x^{\text{cur}},x2)} \end{cases}$$

3.12. Модифікувати поточне рішення

$$x^{\text{cur}} = \begin{cases} x1, & F(x1) < F(x2) \\ x2, & F(x1) > F(x2) \end{cases}$$

3.13. Якщо $l < N_{\text{level}}$, то $l = l + 1$, перехід на крок 3.4

4. Зміна ваги при переміщенні по дереву від термінальної вершини до кореня по ребрах з найкращою вагою

4.1. $w^* = w_{l,l-1}$

4.2. $l = l - 1$

4.3. $w^{\text{cur}} = w_{l,l-1}$

4.4. Якщо $w^{\text{cur}} > w^*$, то $w^* = w^{\text{cur}}$ інакше $w_{l,l-1} = w^*$

4.5. Якщо $l > 1 \wedge \text{rand}() < p^{\text{move}}$, то перехід на крок 4.2

5. Переміщення по дереву від поточної вершини (в силу $\text{rand}() < p^{\text{move}}$ це не обов'язково корінь) до термінальної вершини по ребрах з найкращою вагою

6. Якщо $n < N$, то$n = n + 1$, перехід до кроку 3

Результатом є x^*.

Приклад використання мавпячого пошуку Python

```python
import random

# Визначення функції проблеми, яку необхідно оптимізувати
def problem_function(x):
    # Приклад проблеми: мінімізувати квадрат різниці між x і 5
    return (x - 5) ** 2

# Визначення меж простору пошуку
нижня_межа = 0
верхня_межа = 10

# Функція алгоритму пошуку мавпи
def monkey_search(problem_function, нижня_межа, верхня_межа, розмір_популяції, максимальна_кількість_ітерацій):
    # Ініціалізація популяції
    популяція = [random.uniform(нижня_межа, верхня_межа) for _ in range(розмір_популяції)]

    # Ітерація для заданої кількості ітерацій
    for ітерація in range(максимальна_кількість_ітерацій):
        # Оцінка придатності кожного рішення в популяції
        значення_придатності = [problem_function(x) for x in популяція]

        # Знаходження найкращого рішення в популяції
        найкраще_рішення = min(популяція, key=problem_function)
        найкраща_придатність = problem_function(найкраще_рішення)

        # Генерація нової популяції
        нова_популяція = []

        # Виконання операцій пошуку мавпи для кожного рішення
        for рішення in популяція:
            # Випадкове збурення рішення
            збурене_рішення = рішення + random.uniform(-1, 1)

            # Обмеження рішення межами простору пошуку
            збурене_рішення = max(нижня_межа, min(верхня_межа, збурене_рішення))

            # Оцінка придатності збуреного рішення
            придатність_збуреного = problem_function(збурене_рішення)

            # Оновлення рішення на основі порівняння придатності
            if придатність_збуреного < найкраща_придатність:
                нова_популяція.append(збурене_рішення)
            else:
                нова_популяція.append(рішення)

        # Оновлення популяції новою популяцією
        популяція = нова_популяція

    # Повернення знайденого найкращого рішення
    return min(популяція, key=problem_function)

# Приклад використання
розмір_популяції = 50
максимальна_кількість_ітерацій = 100

найкраще_рішення = monkey_search(problem_function, нижня_межа, верхня_межа, розмір_популяції, максимальна_кількість_ітерацій)
print("Найкраще рішення:", найкраще_рішення)
print("Найкраща придатність:", problem_function(найкраще_рішення))
```

Рис 2.1 мавпячий пошук

```
Найкраще рішення: 4.9993126269677575
Найкраща придатність: 4.7248168545429445e-07
```

Рис 2.2 Результат мавпячого пошку

У цьому прикладі problem_function представляє проблему, яку ви хочете оптимізувати. Алгоритм пошуку мавп застосовується для мінімізації значення проблемної функції. Ви можете змінити problem_function відповідно до вашого конкретного завдання оптимізації.

Алгоритм пошуку Monkey запускається з ініціалізації сукупності рішень-кандидатів в межах зазначеного простору пошуку. Потім він оцінює придатність кожного рішення, виконує операцію пошуку мавп, змінюючи рішення, і генерує нову сукупність на основі порівнянь придатності. Цей процес повторюється вказану кількість ітерацій, і повертається найкраще знайдене рішення.

Зверніть увагу, що це простий приклад реалізації, і ви можете налаштувати та вдосконалити алгоритм відповідно до ваших вимог та конкретної проблеми, яку ви намагаєтесь вирішити.

2.2. Імітація відпалу

Назва «імітація відпалу» (simulated annealing) запропонував Кірпатрік (Kirkpatrick) [14], який використовував процес, що нагадує процес відпалу, що застосовується в металургії. Цей процес має на увазі нагрівання металу або сплаву до високої температури та охолодження його за деяким законом до кімнатної температури. Імовірність зміни положення атома в кристалічній решітці pпов'язана з потенційною енергією решітки Eта температурою Tза законом Максвелла-Больцмана. З цієї причини при охолодженні металу за певним законом загальна потенційна енергія решітки значно зменшується, що покращує характеристики металу.

На кожній ітерації генерується нове рішення x. Згенероване рішення xпорівнюється з поточним x^{cur}. Якщо це рішення краще поточного, то воно вибирається як поточне рішення для наступної ітерації. Якщо це рішення гірше поточного, воно вибирається як поточне рішення з ймовірністю

$$p = \exp\left(-\frac{\Delta E}{T}\right) = \exp\left(\frac{F(x^{\text{cur}}) - F(x)}{T}\right),$$

де вартості $F(x^{\text{cur}}), F(x)$поточного рішення x^{cur} та отриманого рішення xінтерпретуються як енергії, а T інтерпретується як поточна температура.

Можливість вибору деяких ітераціях менш оптимального рішення пов'язані з необхідністю уникнути передчасної збіжності до локального оптимуму.

На початку роботи методу імітації відпалу T відносно велике, що означає порівняно високу ймовірність вибрати менш оптимальне рішення. У процесі пошуку T зменшується за деяким законом, і через кілька ітерацій менш оптимальне рішення вибирається вже з меншою ймовірністю. Функцію, за якою

Т залежить від числа ітерацій, називають «функцією зменшення температури», або «функцією охолодження», за аналогією з фізичним процесом відпалу. Наприклад, можлива функція виду $f(T) = \beta T$, де

β- Коефіцієнт охолодження.

Алгоритм для оптимізації числових функцій

1. Ініціалізація

1.1. Завдання коефіцієнта охолодження β, максимальної температури $T^{\max}$, параметра δ для генерації нового рішення, причому

$0 < \beta < 1, T^{\max} > 0, \delta \in [0,1]$.

1.2. Завдання максимальної кількості ітерацій N, довжини рішення M, мінімальних та максимальних значень для вирішення $x_j^{\min}, x_j^{\max}, j \in \overline{1,M}$.

1.3. Завдання функції вартості (функція цілі)

$F(x) \to \min\limits_{x}$,

де x - Рішення.

1.4. Створення випадковим чином найкращого рішення

$x^* = (x_1^*,\dots, x_M^*), x_j^* = x_j^{\min} + (x_j^{\max} - x_j^{\min})\text{rand}()$,

де rand()- Функція, що повертає рівномірно розподілене випадкове число в діапазоні$[0,1]$

1.5. Задається поточне рішенняx^{cur}, причому$x^{\text{cur}} = x^*$.

1.6. $T(0) = T^{\max}$.

2. Номер ітерації $n = 1$.

3. Генерація рішенняxвід рішення x^{cur}

3.1. $x1_j = \max\{x_j^{\min}, x_j^{\text{cur}} - \delta(x_j^{\max} - x_j^{\min})\}$,

$x2_j = \min\{x_j^{\max}, x_j^{\text{cur}} + \delta(x_j^{\max} - x_j^{\min})\}, j \in \overline{1,M}$

3.2. $x_j = x1_j + (x2_j - x1_j)\text{rand}(), j \in \overline{1,M}$

4. $T(n) = \beta T(n-1)$

5. Обчислюється ймовірність вибору рішення

$$p = \exp\left(\frac{F(x^{\text{cur}}) - F(x)}{T(n)}\right)$$

6. Якщо $F(x) \leq F(x^{\text{cur}}) \vee p > \text{rand}()$, то $x^{\text{cur}} = x$

7. Якщо $F(x) < F(x^*)$, то $x^* = x$.

8. Якщо $n < N$, то$n = n + 1$, перехід до 3, інакше зупин.

Результатом є x^*.

Алгоритм для пошуку оптимального маршруту

1. Ініціалізація

1.1. Завдання коефіцієнт охолодження β, максимальної температури $T^{\max}$, причому $0 < \beta < 1, T^{\max} > 0$.

1.2. Завдання максимальної кількості ітерацій N, розміру популяції K, довжини рішення M, кількості класів жабQ, кількості найкращих жаб у класі D.

1.3. Встановлюється безліч вершин $V = \{1,\ldots, M\}$та матриця ваг ребер $[d_{\text{ij}}], i, j \in \overline{1, M}$.

1.4. Завдання функції вартості (функція цілі)

$F(x) = d_{x_M, x_1} + \sum_{i=1}^{M-1} d_{x_i, x_{i+1}} \to \min\limits_{x}$,

де d_{x_M, x_1}- Вага ребра(x_i, x_{i+1}), $x_i,\ x_{i+1} \in V$,

x- Вектор позиції жаби (вектор вершин).

1.5. Задається, шляхом упорядкування випадковим чином безлічіV, оптимальне вирішення x^*.

1.6. Задається поточне рішення x^{cur}, причому $x^{\text{cur}} = x^*$.

1.7. $T(0) = T^{\max}$.

2. Номер ітерації $n = 1$.

3. Генерація рішенняxвід рішення x^{cur}на основі перестановки 2-opt

3.1. Випадково вибираються з вектора x^{cur}дві вершини $c1$ і $c2$, причому вибір цих вершин продовжується доти, доки не буде виконано умову

$$1 < c2 - c1 < M - 1$$

3.2. На основі рішення

$x^{\text{cur}} = (x_1^{\text{cur}},\ldots, x_{c1-1}^{\text{cur}}, x_{c1}^{\text{cur}},\ldots, x_{c2}^{\text{cur}}, x_{c2+1}^{\text{cur}},\ldots, x_M^{\text{cur}})$

створюється рішення

$x = (x_1^{\text{cur}},\ldots, x_{c1-1}^{\text{cur}}, x_{c2}^{\text{cur}},\ldots, x_{c1}^{\text{cur}}, x_{c2+1}^{\text{cur}},\ldots, x_M^{\text{cur}})$,

тобто. вершини $x_{c1}^{\text{cur}},\ldots, x_{c2}^{\text{cur}}$ переставляються у зворотному порядку.

4. $T(n) = \beta T(n-1)$

5. Обчислюється ймовірність вибору рішення

$$p = \exp\left(\frac{F(x^{\text{cur}}) - F(x)}{T(n)}\right)$$

6. Якщо $F(x) \le F(x^{\text{cur}}) \vee p >$ rand(), то$x^{\text{cur}} = x$,

derand()- Функція, що повертає рівномірно розподілене випадкове число в діапазоні[0,1]

7. Якщо $F(x) < F(x^*)$, то$x^* = x$.

8. Якщо$n < N$, то$n = n + 1$, перехід до 3, інакше зупин.

Результатом є x^*.

Окрім цього алгоритму також використовується стохастичне тунелювання.

Приклад використання алгоритму імітації відпалу на Python

Алгоритм імітації відпалу може бути використаний для знаходження мінімуму функції Розенброка (Rosenbrock's function), яка є стандартним тестовим випадком для оптимізаційних алгоритмів через свою характеристику "долини". Функція Розенброка визначається наступним чином:

$$f(x, y) = (a - x)^2 + b \times (y - x^2)^2$$

Де зазвичай $a = 1$ та $b = 100$

```python
import math
import random

# Визначення функції Розенброка
def rosenbrock_function(x, y):
    a = 1
    b = 100
    return (a - x)**2 + b * (y - x**2)**2

# Функція алгоритму імітації відпалу
def simulated_annealing_rosenbrock(problem_function, lower_bound, upper_bound, initial_temperature, cooling_rate, num_iterations):
    current_solution = [random.uniform(lower_bound, upper_bound), random.uniform(lower_bound, upper_bound)]
    best_solution = current_solution

    for iteration in range(num_iterations):
        temperature = initial_temperature * math.exp(-cooling_rate * iteration)

        # Генерація нового рішення
        new_solution = [current_solution[0] + random.uniform(-1, 1), current_solution[1] + random.uniform(-1, 1)]
        new_solution[0] = max(lower_bound, min(upper_bound, new_solution[0]))
        new_solution[1] = max(lower_bound, min(upper_bound, new_solution[1]))

        # Обчислення пригодності
        current_fitness = problem_function(*current_solution)
        new_fitness = problem_function(*new_solution)

        # Прийняття нового рішення
        if new_fitness < current_fitness or random.uniform(0, 1) < math.exp(-(new_fitness - current_fitness) / temperature):
            current_solution = new_solution

        # Оновлення найкращого рішення
        if problem_function(*current_solution) < problem_function(*best_solution):
            best_solution = current_solution

    return best_solution

# Параметри алгоритму
lower_bound = -5
upper_bound = 5
initial_temperature = 100.0
cooling_rate = 0.01
num_iterations = 1000

# Застосування алгоритму імітації відпалу до функції Розенброка
result = simulated_annealing_rosenbrock(rosenbrock_function, lower_bound, upper_bound, initial_temperature, cooling_rate, num_iterations)

# Виведення результатів
print("Мінімум функції Розенброка знаходиться при x =", result[0], ", y =", result[1])
print("Значення функції при цьому x та y:", rosenbrock_function(result[0], result[1]))
```

Рис 2.3 імітації відпалу

```
Мінімум функції Розенброка знаходиться при x = 0.811927842327534 , y = 0.6405998620841586
Значення функції при цьому x та y: 0.07006749688313073
>
```

Рис 2.4 Результат імітації відпалу

У цьому прикладі ми застосовуємо алгоритм імітації відпалу до функції Розенброка на площині. Результат виводить мінімум функції Розенброка та відповідне значення функції у цій точці.

Реалізація імітації відпалу для мінімізації просторової прив'язки в пакеті Matlab

Реалізація імітації відпалу для мінімізації

дійсної функції в пакеті Matlab

search_space=[-5 5; -5 5;-5 5]; % Діапазон зміни значень ознак

```
max_it = 100; % максимальна кількість ітерацій
% крок зміни значень ознак
step_size=(search_space(1,2)-search_space(1,1))*0.005;
TMax = 100000.0; % Максимальна температура
beta = 0.98; % параметр beta
best = SimulatedAnnealingSearch(search_space, max_it, TMax, beta, step_size);
best=[];
search_space=[];
% імітація відпалу
function best=SimulatedAnnealingSearch(search_space,max_it,TMax,beta,step_size)
current.vector=random_vector(search_space);
current.cost=Cost(current.vector);
T=TMax;
best=current;
for iter=1:max_it
 candidate=CreateNeighbor(current, search_space, step_size);
 T=T*beta;
 if ShouldAccept(candidate, current, beta)
 current=candidate;
 end
 if candidate.cost<best.cost
 best=candidate;
 end
end
current=[];
% формування рішення і обчислення його вартості
function candidate = CreateNeighbor(current, search_space, step_size)
candidate.vector=current.vector;
candidate.vector=take_step(search_space,current.vector,step_size);
```

```
candidate.cost=Cost(candidate.vector);
% вибір рішення на основі імітації відпалу
function b = Should Accept(candidate, current, temp)
function b = Should Accept(candidate, current, T)
if candidate.cost<=current.cost
 b=1;
end
if exp((current.cost-candidate.cost)/T)>rand()
 b=1;
else
 b=0;
end
% генерація нового рішення
function vector = take_step(min max, current, step_size)
n=length(current);
for i=1:n
 min 1=max(min max(i,1),current(i)-step_size);
 max 1=min(min max(i,2),current(i)+step_size);
 vector(i)=min1+(max1-min1)*rand();
end
% обчислення вартості рішення
function cost = Cost(vector)
cost=0.0;
n=length(vector);
for i=1:n
 cost=cost+vector(i)*vector(i);
end
% евклідова відстань між двома рішеннями
function distance=Distance(c1, c2)
```

```
distance=sqrt(sum((c1-c2).^2));
% створення випадковим чином рішення
function array = random_vector(minmax)
[[n m] = size(min max);
for i=1:n
 array(i)=min max(i,1)+(min max(i,2)-min max(i,1))*rand();
end
```

Реалізація імітації відпалу для завдання комівояжера в пакеті Matlab

```
% множина вершин (міст), представлених координатами
cities = [
[565,575]; [25,185]; [345,750]; [945,685]; [845,655];
[880,660]; [25,230]; [525,1000]; [580,1175]; [650,1130];
[1605,620]; [1220,580]; [1465,200]; [1530,5]; [845,680];
[725,370]; [145,665]; [415,635]; [510,875]; [560,365];
[300,465]; [520,585]; [480,415]; [835,625]; [975,580];
[1215,245]; [1320,315]; [1250,400]; [660,180]; [410,250];
[420,555]; [575,665]; [1150,1160];[700,580]; [685,595];
[685,610]; [770,610]; [795,645]; [720,635]; [760,650];
[475,960]; [95,260]; [875,920]; [700,500]; [555,815];
[830,485]; [1170,65]; [830,610]; [605,625]; [595,360];
[1340,725]; [1740,245]];
max_it = 100; % максимальна кількість ітерацій
num_ants = 30; % Кількість мурах
ro = 0.6; % параметр ro
beta = 2.5; % параметр beta
alpha = 1.0; % параметр alpha
best1 = AntSearch(cities, max_it, num_ants, ro, beta, alpha);
% мурашиний алгоритм
function best = AntSearch(cities, max_it, num_ants, ro, beta, alpha)
```

```
% визначення кількості вершин
num_cities=size(cities,1);
% випадковим чином генерується рішення
best.vector=randperm(num_cities);
% обчислення вартості рішення
best.cost=Cost(best.vector, cities);
% ініціалізація рівня феромону
pheromone=InitialisePheromone(num_cities, best.cost);
solutions=[];
for iter=1:max_it
 % формування рішення і обчислення його вартості для мурах
 for k=1:num_ants
 % формування рішення для поточного мурашки
 candidate.vector=DecisionGenerating(cities, pheromone, beta, alpha);
 % обчислення вартість рішення для поточного мурашки
 candidate.cost=Cost(candidate.vector, cities);
 % порівняння отриманого рішення з кращим
 if candidate.cost<best.cost
 best=candidate;
 end
solutions(k).vector=candidate.vector;
 solutions(k).cost=candidate.cost;
 candidate.vector=[];
 end
 % зміни рівня феромону
 pheromone=UpdatePheromone(pheromone, ro, solutions, num_ants);
 solutions=[];
end
% ініціалізація рівня феромону
```

```
function pheromone = InitialisePheromone(num_cities, naive_score)
v=num_cities/naive_score;
for i=1:num_cities
 for j=1:num_cities
 pheromone(i,j)=v;
 end
end
% формування рішення для поточного мурашки
function perm = DecisionGenerating(cities, pheromone, beta, alpha)
% визначення кількості вершин
num_cities=size(cities,1);
% випадковим чином вибирається вершина
perm(1)=round(1+(num_cities-1)*rand);
i=1;
for i=2:num_cities
 % обчислення ймовірностей переходу
 % для всіх незаборонених вершин
 choices=CalculateChoices(cities, perm(i-1), perm, pheromone, beta, alpha);
 % вибір незабороненої вершини для списку заборон
 perm(i)=SelectNextCity(choices);
end
% зміни рівня феромону
function pheromone = UpdatePheromone(pheromone, ro, solutions, num_ants)
% визначення кількості вершин
num_cities=size(pheromone,1);
for i=1:num_cities
 for j=1:num_cities
 pheromone(i,j)=(1.0-ro)*pheromone(i,j);
 end
```

```
end
for k=1:num_ants
 for i=1:num_cities
 x=solutions(k).vector(i);
 if i==num_cities
 y=solutions(k).vector(1);
 else
 y=solutions(k).vector(i+1);
 end
 pheromone(x,y)=pheromone(x,y)+(1.0/solutions(k).cost);
 pheromone(y,x)=pheromone(y,x)+(1.0/solutions(k).cost);
 end
end
% обчислення ймовірностей переходу для всіх незаборонених вершин
function choices=CalculateChoices(cities, last_city, exclude, pheromone, beta, alpha)
% визначення кількості вершин
num_cities=size(cities,1);
j=1;
for i=1:num_cities
 % якщо вершина незаборонена
 if length(find(exclude==i))==0
 % обчислення ваги ребра
 distance=Distance(cities(last_city,:), cities(i,:));
 % обчислення ймовірності переходу
 prob=(pheromone(last_city,i).^beta)*((1/distance).^alpha);
 choices(j).city=i;
 choices(j).prob=prob;
 j=j+1;
 end
```

```
end
% вибір незабороненої вершини для списку заборон
function next_city = SelectNextCity(choices)
% визначення кількості незаборонених вершин
num_choices=size(choices,1);
% обчислення суми ймовірностей
sum=0.0;
for i=1:num_choices
 sum=sum+choices(i).prob;
end
% вибір незабороненої вершини для списку заборон
i=1;
while i<=num_choices
 v=rand;
 if v<=choices(i).prob/sum
 next_city=choices(i).city;
 return;
 end
 i=i+1;
 if i>num_choices
 i=1;
 end
end
% обчислення вартості рішення
function cost = Cost(permutation, cities)
% визначення кількості вершин
num_cities=size(cities,1);
cost=0;
for i=1:num_cities
```

```
c1=permutation (i);
if i==num_cities
c2=permutation(1);
else
c2=permutation(i+1);
end
cost=cost+Distance(cities(c1,:),cities(c2,:));
end
% евклідова відстань між двома вершинами
function distance=Distance(c1, c2)
distance=sqrt(sum((c1-c2).^2))
```

РОЗДІЛ 3

РОЄВІ БІОЛОГІЧНІ МЕТАЕВРИСТИКИ

В даному розділі роботи розглядаються популяційні роєві біологічні метаевристики

Формально, рій (swarm) можна визначити як групу мобільних агентів, які спілкуються один з одним (прямо або побічно), впливаючи на своє оточення. Взаємодії між агентами призводять до розподілених колективних стратегій, що вирішують проблему. Ройовий (колективний) інтелект пов'язаний із розв'язуючою проблему поведінкою, яка з'являється внаслідок взаємодії цих агентів, і обчислювальний ройовий інтелект пов'язаний з алгоритмічними моделями такої поведінки.

До біологічних ройових систем належать системи птахів, риб, мурах, бджіл, ос, бактерій, жаб, світлячків, кажанів, зозуль, кішок, бур'янів тощо.

До фізичних ройових систем належать системи частинок (крапель) води; системи частинок, на які впливає гравітація, електромагнітні сили або електричні сили; системи частинок, які беруть участь у дифузійних процесах; системи частинок, що ґрунтуються на великому вибуху - великому стисненні тощо.

У межах рою особини (або частинки) відносно прості в структурі, але їхня колективна поведінка зазвичай дуже складна. Поведінка рою - результат взаємодії між особинами (або частинками) рою протягом довгого часу. Ця поведінка не є властивістю будь-якої поодинокої особини (або частинки), а є властивістю всього рою, і зазвичай її не можна передбачити та вивести з простих поведінок особин (або частинок), вона називається "емерджентною" (emergent). Така поведінка є децентралізованою (не координується жодною системою управління), самоорганізуючою та розподіленою.

У рої існує зв'язок між індивідуальною та колективною поведінкою. З одного боку, колективна поведінка особин (або частинок) формує і диктує поведінку рою. З іншого боку, поведінка рою впливає на умови, за яких кожна особина (або частинка) виконує свої дії. Ці дії можуть змінити навколишнє

середовище, і, таким чином, поведінка цієї особини (або частинки) та її сусідів може також змінитися і знову змінити колективну поведінку рою. Звідси, найважливішим компонентом ройового інтелекту є взаємодія (співпраця). Взаємодія серед особин (або частинок) допомагає поліпшити практичні (засновані на досвіді) знання про навколишнє середовище. Взаємодія може бути прямою (за допомогою фізичного контакту, або за допомогою візуального, звукового, або хімічного сприйняття) або непрямим (через зміну деякої області навколишнього середовища, причому ця зміна сприймається тільки особинами (або частинками), які відвідали цю область).

Мета обчислювальних моделей ройового інтелекту полягає в тому, щоб змоделювати прості поведінки особин (або частинок), і локальні взаємодії з навколишнім середовищем і сусідніми особинами (або частинками), щоб отримати більш складні поведінки для вирішення складних проблем, головним чином проблем оптимізації.

3.1. Оптимізація рою частинок

Оптимізація рою частинки (PSO) (particle swarm optimization) була запропонована Еберхартом (Eberhart) і Кеннеді (Kennedy) [15]. PSO, також як і еволюційні метаевристики, використовує популяцію особин - потенційних рішень проблеми і метод стохастичної оптимізації, який заснований на соціальній поведінці птахів або риб у зграї чи комах у рої (рис. 3.1).

Рис. 3.1. Зграї птахів, риб і рій комах

Аналогічно до еволюційних метаевристик тут також початкова популяція потенційних рішень генерується випадковим чином і далі шукається (суб)оптимальне рішення проблеми в процесі свого розвитку. Спочатку в PSO зроблено спробу моделювати поведінку зграї птахів, яка має здатність часом раптово і синхронно перегрупуватися і змінювати напрямок польоту під час виконання деякого завдання. На відміну від еволюційних алгоритмів тут не використовуються генетичні оператори, у PSO особини (звані частинками) літають у процесі пошуку в гіперпросторі пошуку рішень і враховують успіхи своїх сусідів. Якщо одна частинка бачить хороший (перспективний) шлях (у пошуках їжі або захисту від хижаків), то інші частинки здатні швидко піти за нею, навіть якщо вони перебували в іншому кінці рою. нею, навіть якщо вони перебували в іншому кінці рою. З іншого боку у рої, для збереження досить великого простору пошуку мають бути частинки з часткою "божевілля" або випадковості у своїй поведінці (русі)

3.1.1. Алгоритм оптимізації рою частинок

Отже, PSO використовує рій частинок, де кожна частинка представляє потенційне рішення проблеми. Спочатку все частинки Рою займають випадкове положення в просторі і мають маленькі випадкові швидкості. На заключних ітераціях багато частинок сходяться до одного або декількох оптимумів Поведінка частинки в гіперпросторі пошуку рішення весь час підлаштовується відповідно до свого досвіду і досвідом своїх сусід. Крім цього, кожна частинка пам'ятає свою кращу позицію з досягнутим локальним кращим значенням цільової (фітнес) функції і знає найкращу позицію частинок своїх сусідів, де досягнуто глобального на поточний момент оптимуму. У процесі пошуку частинки Рою обмінюються інформацією про досягнуті кращих результатах і змінюють свої позиції і швидкості по певним правилам на основі наявної на поточний момент інформації про локальні та глобальні досягнення. При цьому, глобальний кращий результат відомий всім частинкам і відразу коригується в тому випадку, коли деяка частка Рою знаходить кращу позицію з результатом, що перевершує поточний глобальний оптимум. Кожна частинка Рою підпорядковується досить простим правилам поведінки, що враховує локальний успіх кожної особини і глобальний оптимум всіх особин (або деякого безлічі сусідів) рою.

Кожна i частка характеризується на ітерації n своєю позицією $x_i(n)$ в гіперпросторі і швидкістю руху $v_i(n)$.

Швидкість i частки обчислюється у вигляді

$$v_i(n+1) = v_i(n) + \alpha_1(x_i^{best}(n) - x_i(n))r_1 + \alpha_2(x^*(n) - x_i(n))r_2, \quad (1.1)$$

позиція i-ї частки обчислюється у вигляді

$$x_i(n+1) = x_i(n) + v_i(n+1), \quad (1.2)$$

де $x_i(n) = (x_{i1}(n),..., x_{iM}(n))$ - позиція i-ї частки на ітерації n,

$x_i^{best}(n) = (x_{i1}^{best}(n),..., x_{iM}^{best}(n))$ – найкраща позиція i-ї частки (персональна найкраща позиція),

$x^*(n) = (x_1^*(n),...,x_M^*(n))$ - найкраща позиція по всій популяції (глобальна найкраща позиція),

$v_i(n) = (v_{i1}(n),...,v_{iM}(n))$ - Вектор швидкості i-й частинки на ітерації n

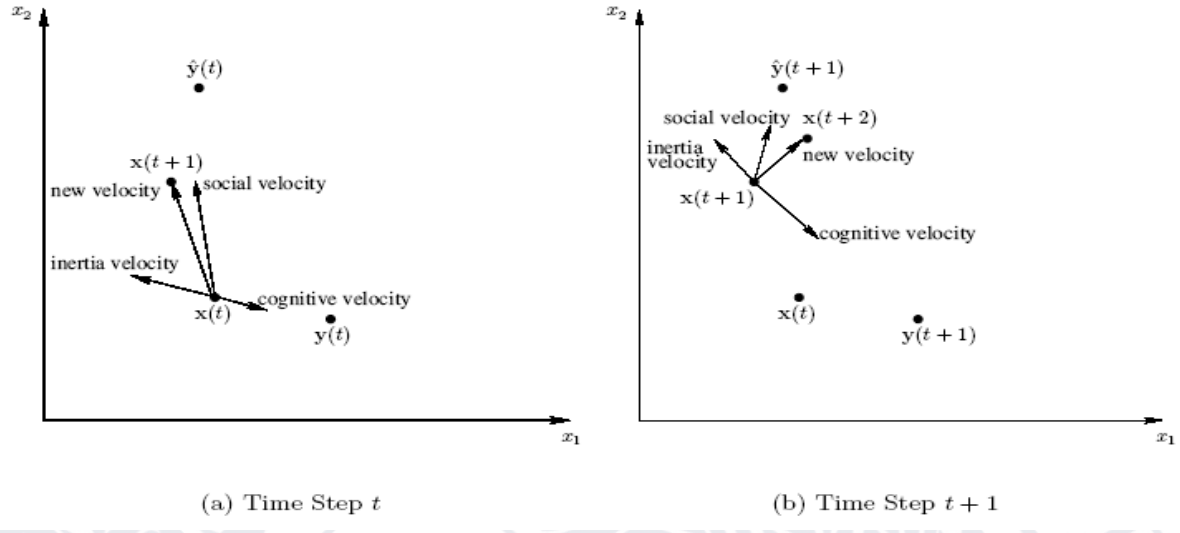
,

α_1, α_2 - Позитивні коефіцієнти прискорення, що регулюють внесок когнітивної та соціальної компонент ,

$r_1 = (r_{11},...,r_{1M}), r_2 = (r_{21},...,r_{2M})$ – вектор випадкових чисел, які вносять елемент випадковості в процес пошуку

Розглянемо вплив різних складових при обчисленні швидкості частки відповідно до (1.1). Перший доданок (1.1) $v_i(n)$ зберігає попередній напрям швидкості i-й частки і може розглядатися як момент, який перешкоджає різкій зміні напрямку швидкості і виступає в ролі *інерційної компоненти* (inertia velocity). Когнітивна компонента (cognitive velocity) $\alpha_1(x_i^{best}(n) - x_i(n))r_1$ визначає характеристики частки щодо її передісторії, яка зберігає кращу позицію цієї частки. Ефект цього доданка в тому, що він намагається повернути частину назад у кращу досягнуту позицію. Третій доданок $\alpha_2(x^*(n) - x_i(n))r_2$ визначає соціальну компоненту (social velocity), яка характеризує частку щодо своїх сусідів. Ефект соціальної компоненти полягає в тому, що вона намагається спрямувати кожну частинку у бік досягнутого роєм (або його деяким найближчим оточенням) глобального оптимуму.

Графічно це наочно ілюструється для двовимірного випадку, як показано на рис.3.2.



(a) Time Step t (b) Time Step $t+1$

Рис.3.2. Приклад зміни позиції та швидкості частки

Алгоритм для оптимізації числових функцій

1. Ініціалізація

1.1. Завдання параметрів α_1, α_2 та максимально допустимих значень для вектора швидкості $V_j^{\max}$, $j \in \overline{1,M}$, причому $\alpha_1, \alpha_2 \in (0,4), V_j^{\max} > 0$

1.2. Завдання максимального числа ітерацій N, розміру популяції K, довжини вектора позиції частки M, мінімальних і максимальних значень для позиції вектора $x_j^{\min}, x_j^{\max}$, $j \in \overline{1,M}$, мінімальних і максимальних значень для вектора швидкості $v_j^{\min}, v_j^{\max}$, $j \in \overline{1,M}$.

1.3. Завдання функції вартості (функція цілі)

$$F(x) \to \min_x,$$

де x - Вектор позиції частинки.

1.4. Створення вихідної популяції P

1.4.1. Номер частинки $k = 1, P = \varnothing$

1.4.2. Створення випадковим чином векторної позиції x_k

$$x_k = (x_{k1}, ..., x_{kM}),\ x_{kj} = x_j^{\min} + (x_j^{\max} - x_j^{\min}) rand(),$$

де $rand()$ - функція, що повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$

1.4.3. Створення вектора найкращої позиції x_k^{best}

$x_k^{best} = x_k$

1.4.4. Створення випадковим чином вектора швидкості v_k

$v_k = (v_{k1},...,v_{kM})$, $v_{kj} = v_j^{\min} + (v_j^{\max} - v_j^{\min})rand()$ або $v_{kj} = 0$

1.4.5. Якщо $(x_k, x_k^{best}, v_k) \notin P$, то $P = P \bigcup \{(x_k, x_k^{best}, v_k)\}, k = k+1$

1.4.6. Якщо $k \le K$ перехід на крок 1.4.2

1.4.7. Визначити частку поточної популяції з найкращою позицією

$k^* = \arg\min_k F(x_k), x^* = x_{k^*}$

2. Номер ітерації $n = 1$

3. Номер частки $k = 1$

4. Модифікація вектора швидкості

4.1. $r_1 = rand()$

4.2. $r_2 = rand()$

4.3. $v_k = v_k + \alpha_1 \cdot (x_k^{best} - x_k)r_1 + \alpha_2 (x^* - x_k)r_2$

4.4. Обмеження на швидкість v_k є, тобто .

$v_{kj} = \max\{-V_j^{\max}, v_{kj}\}$, $v_{kj} = \min\{V_j^{\max}, v_{kj}\}$, $j \in \overline{1, M}$,

або відсутня

5. Модифікація позиції

5.1. $x_k = x_k + v_k$

5.2. $j = 1$

5.3. Якщо $x_{kj} < x_j^{\min}$, то $x_{kj} = x_j^{\min} + | x_{kj} - x_j^{\min} |, v_{kj} = -v_{kj}$

5.4. Якщо $x_{kj} > x_j^{\max}$, то $x_{kj} = x_j^{\max} - | x_{kj} - x_j^{\max} |, v_{kj} = -v_{kj}$

5.5. Якщо $j < M$, то $j = j + 1$перехід на крок 5.3

6. Визначення персональної (локальної) кращої позиції

Якщо $F(x_k) \le F(x_k^{best})$, то $x_k^{best} = x_k$

7. Якщо $k < K$, то $k = k + 1$

8. Визначити частку поточної популяції найкращу за функцією мети

$$k^* = \arg\min_k F(x_k)$$

9. Визначення глобальної кращої позиції

Якщо $F(x_{k^*}) < F(x^*)$, то $x^* = x_{k^*}$

10. Умова зупинки

Якщо $n < N$, то $n = n + 1$перехід до кроку 3

Результатом є x^*.

Поданий алгоритм PSO часто називають *глобальним PSO* , оскільки при корекції швидкості частки використовується інформація про становище досягнутого глобального оптимуму, яка визначається на підставі інформації, що передається всіма частинками рою. На противагу цьому підходу іноді використовується *локальний PSO* , де при корекції швидкості частки використовується інформація, яка передається тільки в якомусь сенсі найближчими сусідніми частинками рою.

3.1.2. Основні аспекти роєвих алгоритмів

Далі розглянемо деякі аспекти представленого вище алгоритму PSO, яких ставляться питання ініціалізації, умов зупинки, обчислення фітнес-функцій тощо. Як було показано раніше, процес пошуку рішення в PSO згідно з наведеним алгоритмом є ітеративним, який триває доки не виконано умову зупинки. Одна ітерація містить корекцію вектора швидкості та позиції, визначення персональних та глобальних кращих позицій. У кожній ітерації

проводиться оцінка значень фітнес-функцій частинок. Оцінка якості потенційного рішення виконується на основі обчислення значення фітнес-функції , яка характеризує це завдання оптимізації. В алгоритмі PSO виконується K обчислень фітнес-функції за ітерацію, де K число частинок у рої.

Перший аспект PSO стосується ініціалізації рою та параметрів алгоритму, що управляють. При цьому визначаються початкові значення швидкостей, позицій і персональних кращих позицій частинок, коефіцієнти прискорення α_1, α_2 і т.п. Для локальних PSO необхідно також визначити відношення сусідства та розмір околиці найближчих сусідів.

Зазвичай позиції частинок ініціалізуються таким чином, щоб пошук простір покривався рівномірно. Слід зазначити, що ефективність PSO істотно залежить від початкового розмаїття безлічі потенційних рішень, тобто від того, як спочатку покритий простір пошуку та як розподілені частки у ньому. Якщо деякі області простору пошуку не покриті початковим роєм, то PSO буде важко знайти оптимум у тому випадку, коли він розташований у не охопленій ділянці. У цьому випадку PSO може знайти такий оптимум завдяки моменту частки, який може спрямувати її в недосліджену область.

Припустимо, що оптимум розташований всередині області, яка визначається двома векторами $x^{\min}, x^{\max}$, які представляють мінімальні та максимальні значення кожної координати. Тоді ефективним методом ініціалізації початкової позиції частинок є :

$$x_j(0) = x_j^{\min} + (x_j^{\max} - x_j^{\min})rand(),$$

де $rand()$ - функція, що повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$

Початкові швидкості частинок у своїй можна покласти нульовими, тобто. $v_j(0) = 0$. В іншому варіанті початкові швидкості можна ініціалізувати випадковими значеннями деякого діапазону, але робити це необхідно обережно. Випадкова ініціалізація позицій частинок визначає початковий рух у випадкових

напрямках. Якщо, однак, виконується випадкова ініціалізація швидкостей, їх значення не повинні бути занадто великими, щоб частинки не вийшли з «зони інтересу», що може суттєво погіршити збіжність.

Персональна найкраща позиція для кожної частки визначається позицією частки у момент $n = 0$, тобто $x_i^{best} = x_i(0)$. Різні схеми ініціалізації позицій частинок досліджені для покриття простору пошуку: на основі послідовностей Соболя, нелінійного симплексу методу і т.д. При вирішенні реальних завдань важливо насамперед, щоб частинки рівномірно покривали простір пошуку.

Другий аспект PSO стосується умови зупинення процесу пошуку рішення, де необхідно враховувати наступне:

– критерій не повинен викликати передчасної збіжності PSO у локальних оптимумах;

– критерій повинен запобігати надміру великих обчислень внаслідок частої оцінки фітнес-функції частинок.

Нині запропоновано низку критеріїв зупинки, основними у тому числі є такі .

1. *Зупинка за максимальною кількістю ітерацій* (при перевищенні заданого порога). Очевидно, що у разі малого порога числа ітерацій процес може зупинитися до того, як буде знайдено гарне рішення. Цей критерій зазвичай використовується разом із критерієм збіжності. Цей критерій корисний, коли необхідно за обмежений заданий час знайти найкраще рішення.

2. *Зупинка за знайденим прийнятним рішенням.* Припустимо, що x^* є оптимум для цільової функції f. Тоді критерій закінчення можна сформулювати в термінах близькості знайденого кращого рішення x до оптимуму $f(x_i) \le \left| f(x^*) - \varepsilon \right|$, тобто при досягненні малої помилки ε. Значення порога помилки εнеобхідно обирати обережно. Якщо значення εзанадто велике, то процес пошуку може зупинитися, якщо знайдено не дуже хороше рішення. З іншого боку, якщо значення ε замало, процес може зовсім не зійтися. Це особливо притаманно глобальному PSO. Крім цього, цей критерій передбачає апріорне

знання оптимального значення, яке не завжди відоме, крім випадків мінімізації помилки (наприклад, у процесі навчання).

3. *Зупинення щодо відсутності поліпшення рішення за задану кількість ітерацій.* Існують різні способи вимірювання поліпшення отримуваних рішень. Наприклад, середня зміна позиції частинок мало, можна припустити, що процес пошуку зійшовся. З іншого боку, якщо середнє значення швидкості частинки за кілька ітерацій приблизно дорівнює нулю, то можливі лише незначні зміни позиції частинок і процес пошуку можна зупинити . На жаль, цей критерій вимагає введення двох параметрів:

- Діапазон зміни ітерацій;
- Порогове значення спостережуваних величин.

4. *Зупинка при прагненні нормалізованого радіусу рою до нуля.* Визначимо нормалізований радіус рою в такий спосіб

$$R_{norm} = \frac{R_{\max}}{diameter(S)},$$

$$R_{\max} = \left\| x_m - x^* \right\| \text{з} \left\| x_m - x^* \right\| \geq \left\| x_i - x^* \right\|,\ m, i \in \overline{1, K},$$

де $diameter(S)$ - Діаметр початкового рою,

$R_{\max}$ - Максимальний радіус.

Можна вважати, що алгоритм зійшовся, якщо $R_{norm} < \varepsilon$. Якщо ε занадто велике, процес пошуку може закінчитися раніше, ніж буде знайдено хороше рішення. З іншого боку, при малому значенні ε може знадобитися занадто велика кількість ітерацій для формування компактного рою.

5. *Зупинка за малим значенням нахилу (крутизни) цільової функції.* Розглянуті критерії враховують лише відносне розташування частинок у просторі пошуку та не беруть до уваги інформацію про крутизну цільової функції. Для врахування змін цільової функції часто використовують таке відношення

$$f'(n) = \frac{f(x^*(n)) - f(x^*(n-1))}{f(x^*(n))}$$

Якщо $f'(t) < \varepsilon$ певного числа послідовних ітерацій, можна вважати, що рій зійшовся. Цей критерій збіжності, на думку багатьох фахівців, перевищує наведені раніше, оскільки він враховує динамічні характеристики рою. Однак використання крутизни цільової функції як критерію зупинки може призвести до передчасної збіжності в локальному екстремумі . Тому доцільно використовувати цей критерій у поєднанні з наведеним раніше критерієм нормалізованого радіуса рою.

Слід зазначити, що збіжність за наведеними критеріями, у випадку, може відповідати досягненню оптимуму (глобального чи локального). В даному випадку збіжність означає, що рій досяг стану рівноваги, коли частинки прагнуть деякої точки в просторі пошуку (в загальному випадку необов'язково точки оптимуму).

3.1.3. Основні параметри роєвих алгоритмів

Ефективність PSO залежить від низки параметрів, до яких відносяться: розмірність задачі, число частинок, коефіцієнти прискорення, вага інерції, тип і розмір сусіднього оточення, число ітерацій, коефіцієнти, що визначають внесок когнітивної та соціальної компонентів. У разі накладання обмежень на можливі швидкості часток необхідно також визначити максимальні значення та деякі коефіцієнти. Далі розглянемо основні параметри PSO.

Розмір рою , число частинок K, відіграє велику роль: чим більше частинок, тим більше розмаїтість потенційних рішень (при гарній схемі ініціалізації, що забезпечує однорідний розподіл частинок). Багато частинок дозволяє покрити більшу частину простору пошуку за ітерацію. З іншого боку, велика кількість частинок підвищує обчислювальну складність ітерації і при цьому PSO може виродитися у випадковий паралельний пошук. Хоча трапляються випадки, що більше частинок веде до зменшення кількості ітерацій при пошуку хороших

рішень. Експериментально показано, що PSO здатні знаходити оптимальне рішення з малим розміром рою від 10 до 30 частинок. У загальному випадку оптимальний розмір рою залежить від розв'язуваної задачі та визначається експериментально.

Число ітерацій, що забезпечує знаходження хорошого рішення залежить від вирішуваного завдання. При малій кількості процес пошуку може не встигнути зійтися. З іншого боку, велика кількість ітерацій, природно, підвищує обчислювальну складність.

Коефіцієнти прискорення α_1, α_2 разом із випадковими векторами r_1, r_2 визначають вклад когнітивної та соціальної компонент у результуючу швидкість частинки. При $\alpha_1 = \alpha_2 = 0$ частинки літають з колишньою швидкістю доки не досягнуть (за інерцією) межі простору пошуку. Якщо $\alpha_1 > 0, \alpha_2 = 0$ частка не залежить від інших особин. Кожна частка знаходить свою найкращу позицію у своєму оточенні шляхом заміни кращої позиції в тому випадку, якщо поточна позиція краща. У разі $\alpha_1 = 0, \alpha_2 > 0$ весь рій прагне однієї точки x^*. Експерименти показують, ефективність пошуку збільшується при балансі цих коефіцієнтів тобто. $\alpha_1 \approx \alpha_2$ Якщо $\alpha_1 = \alpha_2$, то частинки прагнуть середньої точки між x_i^{best} і x^*. Часто при вирішенні завдань вважають $\alpha_1 = \alpha_2$, Проте у випадку ставлення цих коефіцієнтів залежить від розв'язуваного завдання. При $\alpha_1 >> \alpha_2$ кожна частка більш прагне до своєї кращої позиції, що в результаті веде до надмірного блукання частинок. Навпаки, $\alpha_2 >> \alpha_1$ визначає більше прагнення частинок до глобального екстремуму. Зазвичай значення α_1, α_2 у процесі пошуку постійні, але іноді використовуються адаптивні схеми, коли величини α_1, α_2 змінюються.

3.1.4. Основні модифікації PSO

В даний час PSO застосовується при вирішенні багатьох проблем, включаючи стандартні завдання чисельної та комбінаторної оптимізації, навчання нейронних мереж тощо. та показав непогані результати. Але в деяких випадках викладена базова версія PSO має тенденцію до передчасної збіжності та не знаходить оптимальних рішень. Тому розроблено низку модифікацій PSO, основні з яких включають введення ваги інерції, обмеження та звуження діапазону швидкості частинок, різні способи визначення персональних та глобальних позицій та різні моделі швидкості, які будуть розглянуті нижче. При цьому найважливішим аспектом, який визначає ефективність та точність алгоритму оптимізації, є співвідношення (exploration-exploitation) дослідження-локалізація простору пошуку. Дослідження (exploration) характеризує здатність алгоритму досліджувати різні області простору пошуку (розширювати його) для того, щоб локалізувати гарне рішення. З іншого боку локалізація (exploitation – розробка, експлуатація) визначає здатність алгоритму концентрувати пошук у перспективній області для того, щоб покращити рішення. Хороший алгоритм оптимізації має підтримувати баланс між цими двома протилежними властивостями.

Обмеження швидкості є одним з основних методів підвищення ефективності PSO, де властивості дослідження-локалізації простору пошуку визначаються рівнянням зміни швидкості (3.1), яке містить три доданки, що регулюють величину та напрямок зміни швидкості частинки. Вже ранніх роботах було виявлено, що часто швидкість часток різко збільшується, особливо це характерно для частинок далеких від кращих локальних і глобальних позицій. В результаті частки одержують великі позитивні збільшення і залишають межі зони інтересу в просторі пошуку (частки розходяться). Тому бажано обмежити зміну швидкостей частинок у певному діапазоні. Якщо швидкість частинки перевищує деякий поріг, вона штучно встановлюється в деяке максимально допустиме значення. Нехай$V_j^{\max}$ позначає максимально допустиму швидкість j компоненті. Тоді швидкість частинки регулюється (перед зміною позиції відповідно до рівняння (3.2)) таким чином

$$v_{ij}(n+1) = \begin{cases} v'_{ij}(n+1), & v'_{ij}(n+1) < V_j^{\max} \\ V_j^{\max}, & v'_{ij}(n+1) \geq V_j^{\max} \end{cases},$$

де $v'_{ij}(n+1)$ обчислюється відповідно до (1.1).

При цьому значення $V_j^{\max}$ має велике значення, оскільки воно визначає рівень розбиття простору пошуку шляхом обмеження збільшення швидкості. Велике значення $V_j^{\max}$ сприяє глобальному дослідженню простору пошуку, тоді як мале – локалізації гарного рішення. Крім цього, малі значення збільшують кількість ітерацій при пошуку рішення. Більше того, при цьому рій може потрапити у пастку локального екстремуму. З іншого боку, великі значення $V_j^{\max}$ збільшую ризик пропуску перспективної галузі. Частинки можуть «перестрибнути» через добрі рішення та продовжувати пошук у неперспективній області простору пошуку. Часто $V_j^{\max}$ вважають

$$V_j^{\max} = \delta(x_j^{\max} - x_j^{\min}),$$

де $x_j^{\max}, x_j^{\min}$ відповідно максимальне та мінімальне значення j-ої компоненти, $\delta \in (0,1]$.

Значення коефіцієнта залежить від проблемної області і визначається експериментально. Слід зазначити такі важливі аспекти даної модифікації PSO:

1. Утримування швидкості у певному діапазоні не обмежує явно позицію частки і безпосередньо впливає лише розмір кроку переміщення, який визначається швидкістю.

2. Максимальне значення швидкості асоціюється з кожною j компонентою і пропорційно її області визначення. Для спрощення припустимо, що компоненти значення швидкості обмежені однією константою $V^{\max}$. Тоді у разі $x_j^{\max} - x_j^{\min} << V^{\max}$ частки можуть значно відхилитися від оптимуму в j компоненті .

Цей підхід має перевагу в тому, що стримує різке збільшення швидкості, але користувач повинен стежити за цим. По-перше, обмеження швидкості змінює як крок зміни, а й напрям руху частки. Цей ефект показано на рис.3 для двовимірного випадку.

У цьому рис.3.3 $x_i(t+1)$ позначає позицію i-й частки без обмеження швидкості на момент часу t, а $x_i'(t+1)$ – її позицію внаслідок обмеження швидкості з другого компоненті. Зауважимо, що при цьому напрямок пошуку та величина кроку змінилися.

Крім цього, існує інша проблема у разі рівності всіх швидкостей максимальних значень у всіх компонентах. Якщо не передбачено вимірювання для запобігання такій ситуації, то частки залишаються на межі гіперкубу, що визначається $\left[x_i(t) - V^{\max}, x_i(t) + V^{\max}\right]$.

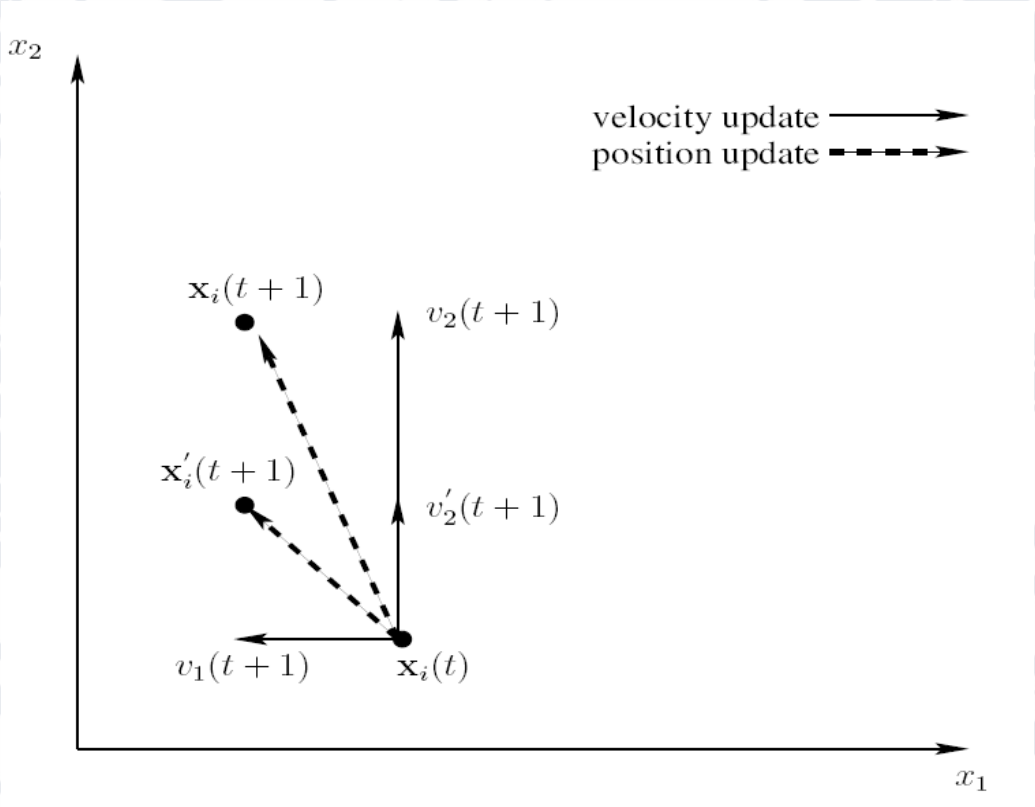


Рис.3.3. Ефект обмеження швидкості.

Можливо, що частинки можуть затриматися в оптимумі, але в загальному випадку важко шукати рішення в цій локальній області. Ця проблема може бути

вирішена по-різному - наприклад, шляхом введення ваги інерції або зменшенням тимчасовим значення $V_j^{\max}$.

Вступ ваги інерції є також популярною модифікацією PSO , де контролюється момент частки шляхом регулювання вкладу попередньої швидкості наступним чином

$$v_{ij}(n+1) = wv_{ij}(n) + \alpha_1 r_{1j}(n)\left[x_{ij}^{best}(n) - x_{ij}(n)\right] + \alpha_2 r_{2j}(n)\left[x_j^*(n) - x_{ij}(n)\right]$$

для глобального PSO. Аналогічно це і для локального PSO.

Значення коефіцієнта w відіграє велику роль і визначає компроміс між дослідженням та локалізацією у просторі пошуку. При $w \geq 1$ швидкість частки збільшується (з урахуванням раніше розглянутого обмеження) і рій «розходиться». Частинкам важко змінити напрямок руху для того, щоб повернутися до перспективної області пошуку рішення. При $w < 1$ частинки сповільнюються до тих пір, поки їхня швидкість не стане рівною нулю. Отже, великі значення w сприяють дослідженню простору пошуку, а малі – локалізації рішення. Однак , занадто малі значення w позбавляють рій здатності досліджувати прозранство пошуку. Чим менше w, тим більший вплив когнітивної та соціальної компоненти. Як та інші параметри, оптимальне значення w проблемно-ориентировано (залежить завдання). У перших реалізаціях цього підходу використовувалися постійні значення w. Далі стали використовувати динамічні w, де старт проводиться з великим значенням w, яке поступово зменшується. Вибір значення w можна поєднати з визначенням значень коефіцієнтів α_1, α_2. Наприклад, $w > \frac{1}{2}(\alpha_1 + \alpha_2) - 1$ гарантує збіжність траєкторії частки. Якщо ця умова не виконується, то можлива розбіжність або зациклювання руху частинок.

При динамічній зміні ваги інерції зазвичай використовуються такі методи.

1. Випадкове регулювання, де різні ваги інерції випадково генеруються кожної ітерації. У деяких випадках використовується Гауссовий розподіл, де $w \sim G(0.72, \sigma)$ і σ досить мало, щоб w не перевищувало 1. Іноді вважають

$w = (\alpha_1 r_1 + \alpha_2 r_2)$ без випадкової зміни вкладу когнітивний та соціальний компонент.

2. Лінійне зменшення ваги інерції, наприклад

$$w(n) = (w(0) - w(N))\frac{(N-n)}{N} + w(N),$$

де N -м максимальна кількість ітерацій виконання алгоритму,

$w(0)$ - Початкова вага,

$w(N)$ - Кінцева вага,

$w(n)$ - Текуча вага на ітерації n.

Зауважимо, що $w(0) > w(N)$.

3. Нелінійне зменшення ваги інерції, яке часто дозволяє скоротити етап дослідження простору пошуку та може бути виконано по-різному:

$$w(n+1) = \frac{(w(n) - 0.4)(N-n)}{N + 0.4} \text{ з } w(0) = 0.9$$

$$w(n+1) = \alpha w(n) \text{ з } w(0) = 0.975$$

4. Нечіткі (fuzzy) ваги інерції на основі нечітких множин та правил регулювання.

Використання коефіцієнта стиснення є модифікацією PSO, яка схожа використання коефіцієнта інерції. Тут рівняння зміни швидкості частки виконується як

$$v_{ij} = (n+1) = \chi[v_{ij}(n) + \phi_1(x_{ij}^{best}(n) - x_{ij}(n)) + \phi_2(x_j^*(n) - x_{ij}(n))],$$

де $\chi = \frac{2k}{|2 - \phi - \sqrt{\phi(\phi-4)}|}$ з $\phi = \phi_1 + \phi_2, \phi_1 = \alpha_1 r_1, \phi_2 = \alpha_2 r_2$

Цей підхід є альтернативою методу обмеження швидкості та при $\phi \geq 4, k \in [0,1]$ гарантовано збіжність рою.

Асинхронні версії PSO також застосовуються практично разом із синхронними, яких належить основний PSO представлений вище. У синхронних

PSO зміна персональних кращих позицій та глобальних кращих позицій виконується синхронно незалежно від зміни позицій частинок. У асинхронних PSO навпаки зміна нової кращої позиції проводиться після того, як кожна частка рою змінила свою позицію.

3.1.5. Порівняння роєвих та генетичних алгоритмів

Сила генетичних алгоритмів насамперед у паралельній природі пошуку рішень. У генетичному алгоритмі реалізована псевдовипадкова форма пошуку рішення, яка зберігає кратні рішення, знищує неперспективні та дає прийнятні рішення. Генетичні оператори дозволяють навіть слабким рішенням залишатися в числі потенційних рішень і генерувати їх прийнятні. Використання генетичних операторів забезпечує успіх пошуку гарного рішення. Всі генетичні алгоритми використовують деяку форму рекомбінації, що дозволяє породжувати нові рішення, які зберігають хороші якості батьків і з ймовірністю показують хороші характеристики. Кросинговер є основним оператором генетичного алгоритму , тоді як мутація використовується набагато рідше. Більшість еволюційних методів використовують таку схему:

1. Випадкова генерація початкової популяції.

2. Обчислення значення фітнес-функції кожної особини, визначального її близькість до оптимуму.

3. Репродукція популяції на основі отриманих значень фітнес-функції .

4. Кінець у виконанні умов зупинки. Інакше перехід на 2.

З цієї схеми видно, що PSO має багато спільного з генетичним алгоритмом . Обидва алгоритми стартують із випадково генерованої популяції та використовують оцінку значень фітнес-функції . У обох алгоритмах еволюція популяції і пошук рішення засновані на випадкових методах. Обидва не гарантують успіху у пошуку рішення. Однак , PSO не має генетичних операторів, подібних до кросинговеру і мутації. Тут потенційні рішення – частки взаємодіють та змінюють свої швидкості. Частинки мають пам'ять, що дуже

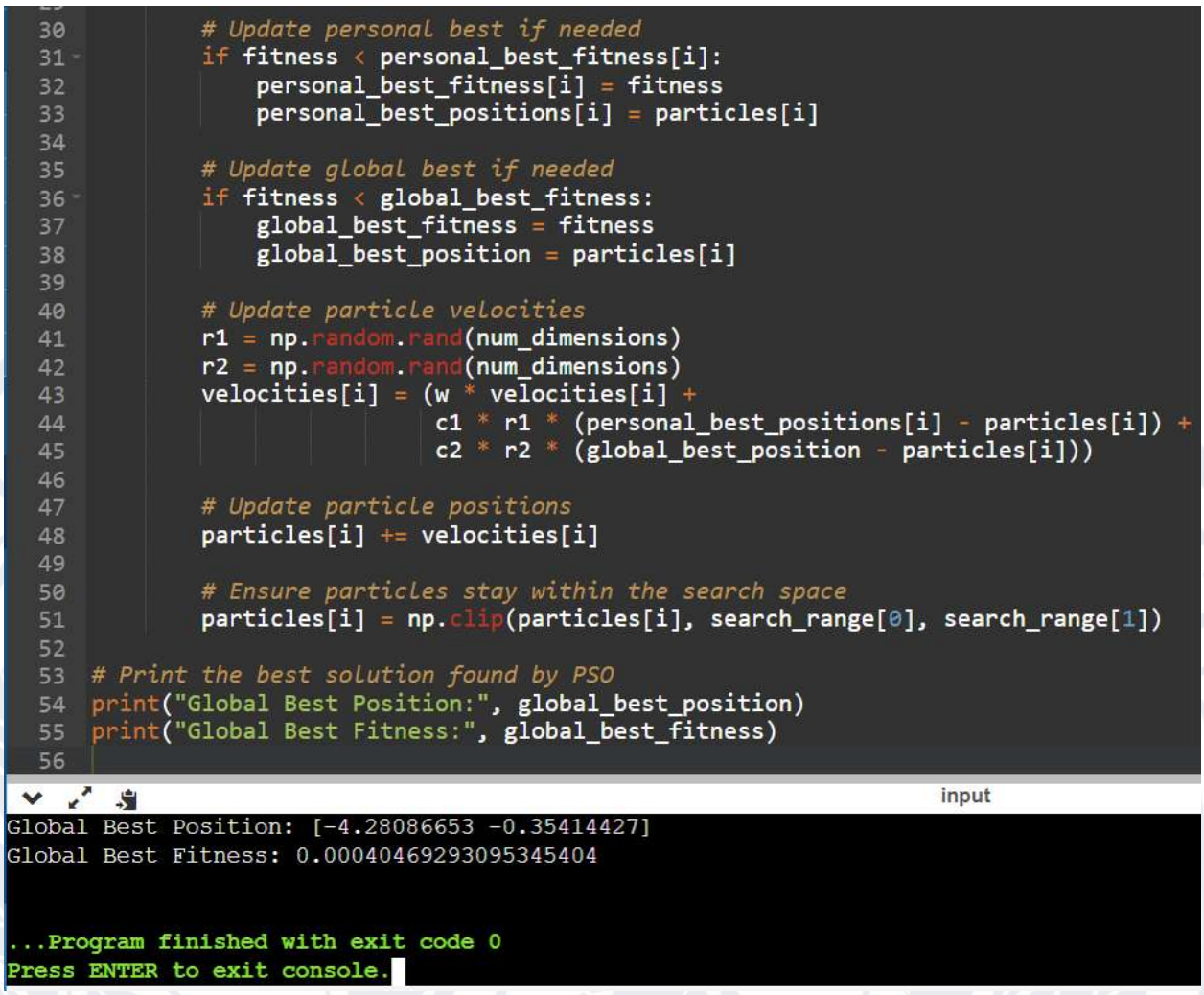
важливе для алгоритму. Механізми передачі в PSO і генетичному алгоритмі зовсім різні. У генетичному алгоритмі хромосоми обмінюються інформацією друг з одним. Тому вся населення рухається як єдина група в область оптимуму. У PSO лише глобальна (локальна) найкраща позиція передається іншим часткам. Це єдиний механізм передачі. У процесі еволюції ведеться пошук найкращого рішення. Порівняно з генетичним алгоритмом усі частки в більшості випадків прагнуть кращого рішення швидше. PSO має багато спільного з еволюційними обчисленнями загалом та з генетичним алгоритмом зокрема. Усі три методи стартують із випадково генерованої початкової популяції та використовують оцінку значень фітнес-функції . Перевага PSO також у тому, що вони, як правило , простіші в реалізації і мають менше параметрів управління. В даний час PSO застосовуються при вирішенні задач чисельної та комбінаторної оптимізації, навчанні штучних нейронних мереж, побудові нечітких контролерів і т.д. у різних галузях науки техніки:

- Управління енергетичних систем;
- Вирішення NP-важких комбінаторних проблем;
- Завдання календарного планування;
– оптимізація у мобільному зв'язку;
- Оптимізація процесів пакетної обробки;
- Оптимізація багатокритеріальних завдань;
- Обробка зображень;
- Розпізнавання образів;
- Кластеризація даних;
- Біоінформатика ;
- Проектування складних технічних систем тощо.

3.1.6. Приклад реалізації алгоритму (PSO) (particle swarm optimization) на мові python

```python
import numpy as np

# Define the fitness function to be optimized (replace with your own function)
def fitness_function(position):
    return sum(x ** 2 for x in position)

# Define the PSO parameters
num_particles = 50
num_dimensions = 2
max_iterations = 100
c1 = 2.0  # Cognitive coefficient (local influence)
c2 = 2.0  # Social coefficient (global influence)
w = 0.7   # Inertia weight
search_range = (-5.0, 5.0)  # Search space range

# Initialize the particles
particles = np.random.uniform(search_range[0], search_range[1], size=(num_particles, num_dimensions))
velocities = np.zeros((num_particles, num_dimensions))
personal_best_positions = particles.copy()
personal_best_fitness = np.zeros(num_particles)
global_best_position = np.zeros(num_dimensions)
global_best_fitness = float('inf')

# Main PSO loop
for iteration in range(max_iterations):
    for i in range(num_particles):
        # Evaluate the fitness of the current particle
        fitness = fitness_function(particles[i])

        # Update personal best if needed
        if fitness < personal_best_fitness[i]:
            personal_best_fitness[i] = fitness
            personal_best_positions[i] = particles[i]

        # Update global best if needed
        if fitness < global_best_fitness:
```

Рис 3.4 (PSO)



```
        # Update personal best if needed
        if fitness < personal_best_fitness[i]:
            personal_best_fitness[i] = fitness
            personal_best_positions[i] = particles[i]

        # Update global best if needed
        if fitness < global_best_fitness:
            global_best_fitness = fitness
            global_best_position = particles[i]

        # Update particle velocities
        r1 = np.random.rand(num_dimensions)
        r2 = np.random.rand(num_dimensions)
        velocities[i] = (w * velocities[i] +
                         c1 * r1 * (personal_best_positions[i] - particles[i]) +
                         c2 * r2 * (global_best_position - particles[i]))

        # Update particle positions
        particles[i] += velocities[i]

        # Ensure particles stay within the search space
        particles[i] = np.clip(particles[i], search_range[0], search_range[1])

# Print the best solution found by PSO
print("Global Best Position:", global_best_position)
print("Global Best Fitness:", global_best_fitness)

input
Global Best Position: [-4.28086653 -0.35414427]
Global Best Fitness: 0.00040469293095345404


...Program finished with exit code 0
Press ENTER to exit console.
```

Рис 3.5 (PSO)

Цей код надає базову реалізацію алгоритму PSO. Ви можете замінити fitness_function на власну цільову функцію для оптимізації конкретної задачі. Крім того, ви можете налаштувати параметри PSO та діапазон простору пошуку відповідно до вашої проблемної області. На практиці, можливо, вам також захочеться реалізувати більш просунуті варіанти PSO і додаткові механізми, такі як коефіцієнти звуження для кращої збіжності.

3.2. Оптимізація рою кішок

Оптимізація рою кішок (cat swarm optimization) була запропонована Чу (Chu), Цаєм (Tsai) і Пеном (Pan) [16]. Вона заснована на соціальній поведінці кішок, пов'язаних із добуванням їжі. Кішка основний час перебуває у спокої,

оглядаючись навколо для переходу на нову позицію, і невеликий час переміщується для пошуку мети.

Мета алгоритму полягає в тому, щоб, використовуючи всіх кішок рою, визначити місцезнаходження оптимумів у багатовимірному просторі. Спочатку всі кішки роя займають випадкове становище у просторі та мають маленькі випадкові швидкості. У ході роботи алгоритму змінюється позиція кожної кішки на основі її швидкості та кращої попередньої позиції, визначеної у всьому рою. Найкраща позиція (рішення) визначаться на основі функції мети. На останніх ітераціях безліч кішок сходиться до одного або кількох оптимумів.

3.2.1 Алгоритм для оптимізації числових функцій

1. Ініціалізація

1.1. Завдання ймовірності режиму пошуку p^{search}, параметра α обчислення швидкості, параметра δ для генерації нової позиції, причому $0 < \delta < 1,\ \alpha \in [0,4]$.

1.2. Завдання максимального числа ітерацій N, розміру популяції K, довжини вектора позиції кішки M, мінімальних і максимальних значень для позиції вектора $x_j^{\min}, x_j^{\max},\ j \in \overline{1,M}$, мінімальних і максимальних значень для вектора швидкості $v_j^{\min}, v_j^{\max},\ j \in \overline{1,M}$.

1.3. Завдання функції вартості (функція цілі)

$$F(x) \to \min_x,$$

де x - Вектор позиції кішки.

1.4. Створення випадковим чином вектора кращої позиції

$$x^* = (x_1^*,...,x_M^*),\ x_j^* = x_j^{\min} + (x_j^{\max} - x_j^{\min})rand(),$$

де $rand()$ - функція, що повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$

1.2. Створення вихідної популяції P

1.2.1. Номер $k = 1$ кішки $P = \varnothing$

1.2.2 . Створення випадковим чином векторної позиції x_k

$x_k = (x_{k1},...,x_{kM})$, $x_{kj} = x_j^{\min} + (x_j^{\max} - x_j^{\min})rand()$,

де $rand()$ - функція, що повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$

1.2.3 . Створення випадковим чином вектора швидкості v_k

$v_k = (v_{k1},...,v_{kM}), v_{ij} = v_j^{\min} + (v_j^{\max} - v_j^{\min})rand()$

1.2.4 . Якщо $(x_k, v_k) \notin P$, то $P = P \bigcup \{(x_k, v_k)\}, k = k + 1$

1.2.5. Якщо $k \le K$ перехід на крок 1.5.2

2. Номер ітерації $n = 1$.

3. Визначити кішку найкращу за функцією мети

$$k^* = \arg\min_k F(x_i)$$

4. Якщо $F(x_{k^*}) < F(x^*)$, то $x^* = x_{k^*}$

5. Номер кішки $k = 1$

6. Якщо $rand() < p^{search}$, то перехід до кроку 8

7. Режим пошуку (кішка у спокої, але оглядається навколо для переходу на нову позицію)

7.1. Створити L копії векторів позицій k -й кішки

7.1.1. $l = 1$.

7.1.2. $j = 1$

7.1.3 $\lambda = rand()$

7.1.4. $\tilde{x}_{lj} = \begin{cases} x_{kj} + \delta x_{kj}, & \lambda > 0.5 \\ z_{kj} - \delta x_{kj}, & \lambda \le 0.5 \end{cases}$,

7.1.5. $\tilde{x}_{lj} = \max\{x_j^{\min}, \tilde{x}_{lj}\}$, $\tilde{x}_{kj} = \min\{x_j^{\max}, \tilde{x}_{lj}\}$,

7.1.6. Якщо $j < M$, то $j = j + 1$ перейти на крок 7.1.3

7.1.7. Якщо $l < L$, то $l = l + 1$ перейти на крок 7.1.2

7.2. Обчислення ймовірності

$$p_l = \begin{cases} 1 - \dfrac{F(\tilde{x}_l) - \min\limits_{m \in \overline{1,L}} F(\tilde{x}_m)}{\max\limits_{m \in \overline{1,L}} F(\tilde{x}_m) - \min\limits_{m \in \overline{1,L}} F(\tilde{x}_m)}, & \max\limits_{m \in \overline{1,L}} F(\tilde{x}_m) \neq \min\limits_{m \in \overline{1,L}} F(\tilde{x}_m) \\ 1, & \max\limits_{m \in \overline{1,L}} F(\tilde{x}_m) = \min\limits_{m \in \overline{1,L}} F(\tilde{x}_m) \end{cases},$$

$l \in \overline{1,L}$

7.3. Заміна векторів позицій k.

Якщо $\sum_{l=1}^{c-1} p_l < rand() \le \sum_{l=1}^{c} p_l$, то $x_k = \tilde{x}_c$

7.4. Перехід до кроку 9

8. Режим стеження (кішка шукає мету)

8.1. Модифікувати швидкість k кішки

8.1.1. $v_k = v_k + \alpha(x_{k^*} - x_k) rand()$

8.1.2. $v_{kj} = \max\{-\gamma, v_{kj}\}$, $v_{kj} = \min\{\gamma, v_{kj}\}$, $j \in \overline{1,M}$

8.2. Модифікувати позицію k кішки відповідно до швидкості

8.2.1. $x_k = x_k + v_k$

8.2.2 $j = 1$

8.2.3. Якщо $x_{kj} < x_j^{\min}$, то $x_{kj} = x_j^{\min} + | x_{kj} - x_j^{\min} |, v_{kj} = -v_{kj}$

8.2.4. Якщо $x_{kj} > x_j^{\max}$, то $x_{kj} = x_j^{\max} - | x_{kj} - x_j^{\max} |, v_{kj} = -v_{kj}$

8.2.5. Якщо $j < M$, то $j = j + 1$ перехід на крок 6.2.3

9. Якщо $k < K$, то $k = k + 1$ перейти на крок 6

10. Якщо $n < N$, то $n = n + 1$перехід до кроку 3

Результатом є r^*.

3.2.2. Приклад реалізації алгоритму Cat Swarm Optimization (CSO) на Python

Оптимізація котячим роєм (Cat Swarm Optimization, CSO) - це алгоритм оптимізації, натхненний природою, який є відносно менш поширеним, ніж інші алгоритми оптимізації, такі як PSO або генетичні алгоритми.

Для реалізації алгоритму потрібно буде встановити бібліотеку numpy

```python
import numpy as np

# Define the objective function to be optimized
def objective_function(x):
    return sum(x ** 2)

# CSO parameters
num_cats = 30
num_dimensions = 3
max_iterations = 100
lb = -10  # Lower bound of the search space
ub = 10   # Upper bound of the search space
c = 1     # Individuality coefficient
s = 1     # Collective coefficient

# Initialize cat positions and velocities
cat_positions = np.random.uniform(lb, ub, size=(num_cats, num_dimensions))
cat_velocities = np.random.uniform(-0.1, 0.1, size=(num_cats, num_dimensions))

# Initialize the best position and fitness
best_position = cat_positions[0]
best_fitness = objective_function(best_position)

# Main CSO loop
for iteration in range(max_iterations):
    for i in range(num_cats):
        # Evaluate the fitness of the current cat
        fitness = objective_function(cat_positions[i])
```

Рис 3.6 (CSO)

```
        fitness = objective_function(cat_positions[i])

        # Update the best position and fitness
        if fitness < best_fitness:
            best_fitness = fitness
            best_position = cat_positions[i]

        # Update cat velocities
        cat_velocities[i] = s * cat_velocities[i] + c * np.random.rand() *
            (best_position - cat_positions[i])

        # Update cat positions
        cat_positions[i] += cat_velocities[i]

        # Ensure cats stay within the search space
        cat_positions[i] = np.clip(cat_positions[i], lb, ub)

# Print the best solution found by CSO
print("Best Position:", best_position)
print("Best Fitness:", best_fitness)

```

Рис 3.7 (CSO)

Результат

```
Best Position: [ 10. -10.  10.]
Best Fitness: 1.1131530485617738
```

Рис 3.8 (CSO)

3.3. Мурашині алгоритми

Мурахи з'явилися землі більше 100 мільйонів років тому й нині їх населення становить 10^{16} особин, загальну вагу якої можна порівняти з вагою проживаючих людей. Більшість мурах є соціальними комахами, котрі живуть колоніями від 30 до мільйона особин. При відносно простому поведінці кожної окремої особини мурашині колонії представляють складну соціальну структуру і здатні вирішувати складні завдання, наприклад, знаходити оптимальні шляхи від гнізда до джерела їжі. Це привернула увагу багатьох дослідників, які вивчали механізми взаємодії особин колонії. Серед них, перш за все, привернула увагу

дослідників непряма форма зв'язку між особами, яка була названа стигмергією (stigmergy) і є рознесеною в часі взаємодією, при якій одна особина змінює деяку область навколишнього середовища, а інші особи використовують цю інформацію в процесі вирішення задачі . Ця інформація (зміна навколишнього середовища) носить локальний характер - вона може бути змінена (і сприйнята) тільки комахами, що відвідали цей локус - ділянка середовища. Стигмергія є непрямою та асинхронною формою комунікації в якій комахи змінюють навколишнє середовище для передачі інформації іншим комахам, які реагують на цю зміну. Слово стигмергія утворене з двох грецьких слів: " stigma ", що означає знак; « ergon » - робота. Особи сприймають сигнали (як знаків), які породжують певний відгук чи дію. Визначено дві форми стигмергії : сематектонічна (sematectonic) та знакова (sign-based). Сематектонічна відноситься до комунікації за допомогою зміни фізичних характеристик довкілля. Прикладом сематектонічної стигмергії є дії при будівництві гнізда, його очищенні та вирощуванні виводка. Сигнальна стигмергія реалізує комунікацію за допомогою сигнального механізму у вигляді хімічних сполук, що відкладаються мурахами.

Саме, у багатьох мурашиних колоніях стигмергія реалізується з допомогою спеціального ферменту « феромона », який відкладається мурашкою у процесі руху. При цьому мурашка позначає феромоном відвідану ділянку середовища. Інші мурахи сприймають «запах» відкладеного феромона і намагаються слідувати по зазначеному шляху. Це породжує асинхронну та непряму схему комунікації, де мурахи передають інформацію один одному за допомогою феромону . При цьому виникає позитивний зворотний зв'язок - навіть мала феромона змушує мурах йти поміченим шляхом і відкладати на ньому все більшу кількість ферменту. Адаптивність поведінки мурах заснована на сприйнятті випарів феромону , яке у природі триває кілька діб. Можна провести аналогію між розподілом феромону в навколишньому просторі і глобальною пам'яттю мурашника, яка носить динамічний характер.

Мурашині алгоритми, як і більшість раніше розглянутих видів еволюційних алгоритмів, засновані на використанні популяції потенційних

рішень і розроблені для вирішення задач комбінаторної оптимізації, перш за все, пошуку різних шляхів на графах. Кооперація між особинами (штучними мурахами) тут реалізується на основі моделювання стигмергії . При цьому кожен агент, званий штучною мурахою, шукає вирішення поставленого завдання. Штучні мурахи послідовно будують рішення задачі, пересуваючись по графу, відкладають феромон і при виборі подальшої ділянки шляху враховують концентрацію цього ферменту. Чим більша концентрація феромону в наступній ділянці, тим більша ймовірність його вибору.

Реальні мурахи завдяки стигмергії здатні знаходити найкоротший шлях від гнізда до джерела їжі досить швидко та без візуального (прямого контакту). Більше того, вони здатні адаптуватися до змін навколишнього середовища. Провели численні експерименти з реальними мурахами, які показали такі результати.

Розглянемо експерименти з перешкодами, що показані на рис.3.4-3.7.

Тут на рис.3.4 показано рух мурах між гніздом та джерелом їжі без перешкоди. Далі на рис.3.5-3.7 показаний характер руху у тому випадку, коли на шляху виникла перешкода.

З малюнків видно, що при появі перешкоди в початковій фазі руху рис.1.6 мурахи з однаковою ймовірністю вибирають і короткий і довгий шлях , оскільки концентрація феромону спочатку однакова для обох варіантів.

Але через деякий час за рахунок того, що по короткому шляху мурахи швидше проходять шлях, на ньому концентрація феромону стає вищою і тому мурахи вибирають оптимальний шлях.

Рис.3.9. Рух мурах без перешкод.

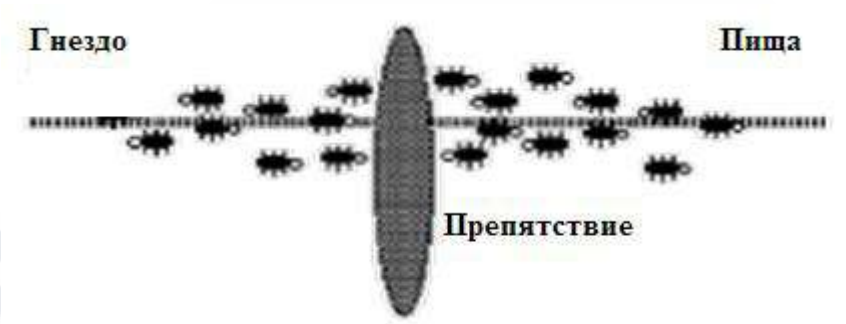


Рис.3.10. Перешкода на шляху між гніздом та їжею.

Рис.3.11. Початкова фаза руху мурах із перешкодою.

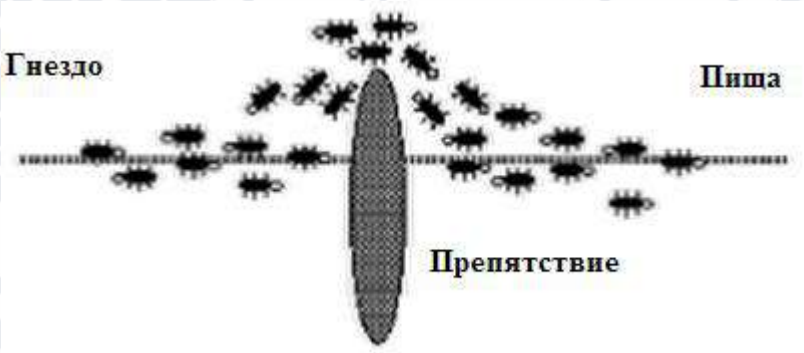


Рис.3.12. Вибір мурах найкоротшого шляху.

Не менш відомий експеримент із двома мостами був проведений із колонією аргентинських мурах, який представлений на рис. 3.8 –3.9. Тут на шляху між гніздом та їжею необхідно зробити вибір одного із 2-х мостів.

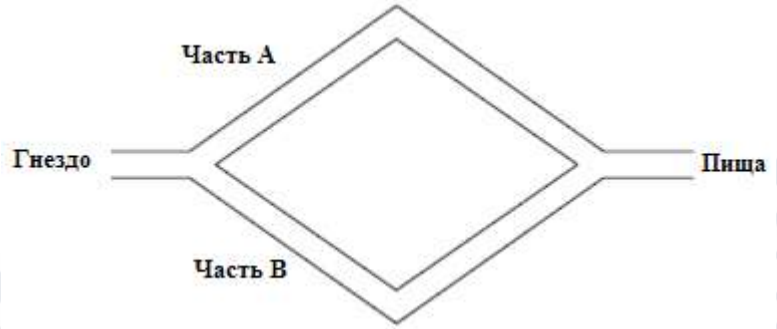


Рис.3.13. Експеримент із 2-ма мостами

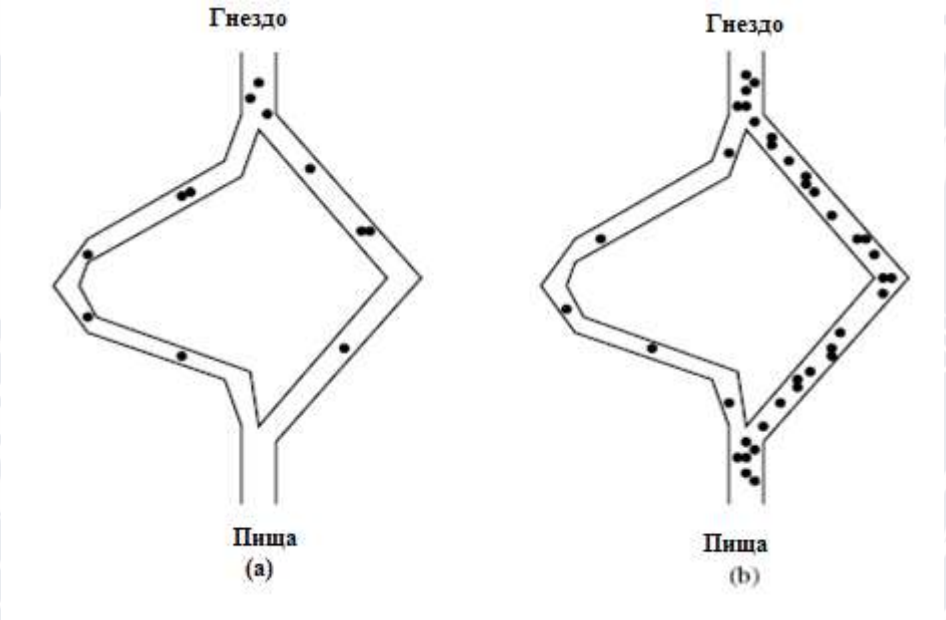


Рис.3.14. Вибір найкоротшого шляху

Тут на рис.1.8 показаний випадок із двома еквівалентними шляхами між гніздом та їжею. Експерименти показали однакову концентрацію мурах на обох можливих шляхах. Далі на рис.1.9 представлений випадок, коли мости мають різну довжину. У початковій фазі мурахи з ймовірністю вибирають мости. Але далі, зі збільшенням концентрації феромона короткому шляху вони вибирають оптимальний шлях.

Нехай $K_A(n)$ і $K_B(n)$ позначають число мурах на шляхах А і В на ітерації відповідно n. Емпірично було знайдено, що ймовірність вибору мосту на ітерації n відбувається відповідно до наступної формули

$$p_A(n+1) = \frac{(c + K_A(n))^{\alpha}}{(c + K_A(n))^{\alpha} + (c + K_B(n))^{\alpha}} = 1 - p_B(n+1) \qquad (1.3)$$

де c – ступінь «привабливості» гілки, що не досліджується, і α визначає зсув при використанні феромону в процесі вибору варіанта рішення.

На основі ймовірностей, що визначаються (1.3), правило вибору мурахою моста можна сформулювати в такий спосіб. Нехай генерується рівномірно розподілене випадкове число r інтервалі (0,1). Якщо $r \le p_A(n+1)$, то мурашка вибирає шлях А, інакше – шлях В.

Зазначимо, що незважаючи на те, що мурашина колонія демонструє складну адаптивну поведінку, яка дозволяє їй вирішувати важкі завдання, поведінка однієї мурахи підпорядковується досить простим правилам. Мураха можна розглядати як агента, що піддається впливу та формує на нього відповідну реакцію: мураха сприймає концентрацію феромону і на цій основі виконує дію. Тому мураха абстрактно може розглядатися як простий обчислювальний агент. Штучна мураха алгоритмічно моделює просту поведінку реальної мурашки (точніше її аспекти, що нас цікавлять).

1.3.1. Мурашина система

Перший мурашиний алгоритм, названий мурашиною системою (ant system), був розроблений Доріго (Dorigo) [8] , який за сучасною класифікацією відноситься до мурашиної системи.

Імовірність переходу k-го мурахи з вершини i на вершину j на ітерації n визначається наступним чином

$$p_{ij} = \begin{cases} \dfrac{(\tau_{ij}(n))^{\alpha}(\eta_{ij})^{\beta}}{\sum_{l \in V^{acept}} (\tau_{il}(n))^{\alpha}(\eta_{il})^{\beta}}, & j \in V^{acept} \\ 0, & j \notin V^{acept} \end{cases} \qquad (1.4)$$

де $\tau_{ij}(n)$- Апостеріорна ефективність переходу з вершини i в вершину j, яка визначається інтенсивністю феромона для ребра (i, j);

η_{ij} – апріорна ефективність переходу з вершини i на вершину j на основі деякої евристики,

V^{acept} - Безліч допустимих вершин.

Імовірність переходу в мурашиній системі, визначена (3.1), має такі особливості:

1. При обчисленні ймовірності переходу зроблено спробу збалансувати вплив інтенсивності феромону $\tau_{ij}(n)$(що відображає передісторію успішних дій) та евристичної інформації η_{ij}(що виражає перевагу деякого вибору). Цей баланс управляє процесом експлуатації-розширення у просторі пошуку рішення. Баланс регулюється значеннями коефіцієнтів α та β. При $\alpha = 0$ інформація про концентрацію феромону немає і попередній досвід ігнорується, тобто. відбувається вибір лише з урахуванням евристики. Якщо $\beta = 0$, то не враховується евристична інформація та відбувається вибір лише на основі феромону . Евристична інформація про перевагу вибору наступної вершини може подаватись у різній формі і залежить від завдання. Наприклад, для вибору найкоротшого шляху можна використовувати $\eta_{ij} = \frac{1}{d_{ij}}$, де d_{ij} відстань між вершинами i і j. Очевидно, що в цьому випадку краще коротка дуга, що виходить з вершини і.

2. Безліч V^{acept} визначає безліч допустимих вершин для k мурашки. Це безліч може включати сусідні до i вершини, які не відвідувалися k мурашкою. Для цього для кожної мурашки створюється та відстежується табу-список V^{tabu} . Вершини цього списку видаляються з V^{acept}, оскільки кожна вершина може відвідуватися лише один раз

Замість (1.4) можна використовувати

$$p_{ij} = \begin{cases} \dfrac{\alpha\tau_{ij}(n) + (1-\alpha)\eta_{ij}}{\sum\limits_{l \in V^{acept}} \alpha\tau_{ij}(n) + (1-\alpha)\eta_{ij}}, & j \in V^{acept} \\ 0, & j \notin V^{acept} \end{cases} \quad (1.5)$$

Тут параметр α визначає відносну важливість концентрації феромону $\tau_{ij}(n)$ в порівнянні з евристикою η_{ij}. Варіант (3.3) у порівнянні з варіантом (3.2) не вимагає завдання параметра β.

Правило випаровування (зміни рівня) феромону представлено у вигляді

$$\tau_{ij}(n) = (1-\rho)\tau_{ij}(n-1) + \Delta\tau_{ij}(n) \quad (1.6)$$

Доріго розробив три варіанти мурашиної системи, які відрізняються методом обчислення $\Delta\tau_{ij}(n)$ (у припущенні, що вирішується завдання мінімізації):

1. Перший варіант (Ant-cycle AS)

$$\Delta\tau_{ij}(n) = \gamma \sum_{\tilde{x} \in Q_{ij}} \frac{1}{F(\tilde{x})}$$

де $Q_{ij} \subset \{x_k\}$ - безліч шляхів пройдених мурахами, які містять ребро (i, j) або (j, i), γ - Позитивна константа.

Тут феромон відкладається назад пропорційно до якості $F(\tilde{x})$ на ребрах повного шляху, побудованого мурахою. Для зміни концентрації феромону використовується глобальна інформація.

2. Другий варіант (Ant-density AS)

$$\Delta\tau_{ij}(n) = \gamma \mid Q_{ij} \mid$$

де $Q_{ij} \subset \{x_k\}$ - безліч шляхів пройдених мурахами, які містять ребро (i, j) або (j, i), γ - Позитивна константа.

У цій модифікації кожна мураха відкладає однакову кількість феромону на будь-якому ребрі побудованого шляху. Цей підхід враховує лише кількість мурах,

що пройшли по цьому ребру (i, j). Чим вище щільність трафіку на ребрі, тим більше вона є привабливою для остаточного рішення.

3. Третій варіант (Ant-quantity AS)

$$\Delta\tau_{ij}(n) = \frac{\gamma}{d_{ij}} \mid Q_{ij} \mid$$

де $Q_{ij} \subset \{x_k\}$- безліч шляхів пройдених мурахами, які містять ребро (i, j) або $(j, i), \gamma$ - Позитивна константа.

В цьому випадку при корекції концентрації феромону використовується тільки локальна інформація - відстань d_{ij} і мурашина система вважає за краще вибирати короткі ребра.

3.3.1.1. Алгоритм для пошуку оптимального маршруту

1. Ініціалізація

1.1. Завдання коефіцієнта історії (ваги, інтенсифікації феромону) α, коефіцієнта (інтенсифікації) евристики β, коефіцієнта інтенсивності випаровування (керує швидкістю втрати інформації) ρ, параметра γ для правила зміни рівня феромону , $\beta \in [2,5]$ причому $0 < \alpha < 1,,,, \; 0 < \rho < 1. \gamma > 0$

При $\alpha = 0$ відбувається вибір найближчої вершини.

При $\beta = 0$ відбувається вибір тільки на основі феромону .

1.2. Завдання максимальної кількості ітерацій N, розміру популяції K, довжина вектора вершин M .

1.3. Встановлюється безліч вершин $V = \{1,...,M\}$ і матриця ваг ребер $[d_{ij}]$, $i, j \in \overline{1, M}$.

1.4. Завдання функції вартості (функція цілі)

$$F(x) = d_{x_M, x_1} + \sum_{i=1}^{M-1} d_{x_i, x_{i+1}} \to \min_x,$$

де d_{x_M,x_1} - вага ребра (x_i, x_{i+1}), $x_i, x_{i+1} \in V$,

x - Вектор вершин.

1.5. Задається, шляхом упорядкування випадковим чином множини V, оптимальний вектор вершин x^*.

1.6. Задається початковий рівень феромону $\tau_{ij}(0)$ на ребрі (i, j) в останній момент часу $n = 0$, $i, j \in \overline{1, M}$. Він ініціалізується невеликим позитивним числом. Наприклад,

$$\tau_{ij}(0) = \frac{M}{F(x^*)},\ \tau_{ii}(0) = 0,\ i, j \in \overline{1, M}.$$

2. Номер ітерації $n = 1$.

3. Номер мурахи $k = 1$.

4. Вибір номера стартової вершини одним із двох способів:

– всі мурахи містяться у одну вершину, тобто. $i = 1$

– мурахи містяться у різні вершини випадковим чином, тобто.

$$i = round(1 + (M - 1)rand()),$$

де $rand()$ - функція, що повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$,

$round()$ - Функція, що округлює число до найближчого цілого.

5. Безліч заборонених вершин $V^{tabu} = \{i\}, x_{k1} = i$

6. Обчислюються ймовірності переходу k-го мурахи з вершини i до інших вершин

$$p_{ij} = \begin{cases} \dfrac{\left(\tau_{ij}(n-1)\right)^{\alpha}\left(1/d_{ij}\right)^{\beta}}{\sum_{l \in V \setminus V^{tabu}} \left(\tau_{il}(n-1)\right)^{\alpha}\left(1/d_{il}\right)^{\beta}}, & j \in V \setminus V^{tabu} \\ 0, & j \notin V \setminus V^{tabu} \end{cases}, \quad j \in \overline{1, M}$$

7. Вибирається вершина j_c, яка задовольняє нерівності

$$\sum_{j=1}^{j_c-1} p_{ij} < rand() \le \sum_{j=1}^{j_c} p_{ij}$$

8. $V^{tabu} = V^{tabu} \bigcup \{j_c\}, x_{k,|V^{tabu}|} = j_c$

9. Якщо $|V^{tabu}| < M$, то $i = j_c$ перехід до 6.

10. Якщо $F(x_k) < F(x^*)$, то $x^* = x_k$.

11. Якщо $k < K$, то $k = k + 1$ перехід до 4.

12. Правило зміни рівня феромону подане у вигляді

$$\tau_{ij}(n) = \tau_{ji}(n) = (1-\rho)\tau_{ij}(n-1) + \gamma \sum_{\tilde{x} \in Q_{ij}} \frac{1}{F(\tilde{x})}, \qquad i \in \overline{1, M-1},$$

$j \in \overline{i+1, M}$

або

$$\tau_{ij}(n) = \tau_{ji}(n) = (1-\rho)\tau_{ij}(n-1) + \gamma |Q_{ij}|, \; i \in \overline{1, M-1}, \; j \in \overline{i+1, M}$$

або

$$\tau_{ij}(n) = \tau_{ji}(n) = (1-\rho)\tau_{ij}(n-1) + \frac{\gamma}{d_{ij}} |Q_{ij}|, \; i \in \overline{1, M-1}, \; j \in \overline{i+1, M},$$

де $Q_{ij} \subset \{x_k\}$ - безліч векторів вершин, які містять ребро (i, j) або (j, i).

13. Якщо $n < N$, то $n = n + 1$ перехід до 3, інакше зупин.

Результатом є x^*.

3.3.2. Система мурашиної колонії

Система мурашиної колонії (ant colony system) запропонована Доріго (Dorigo) і Гамбарделлом (Gambardella) [18] і є модифікацією мурашиної системи[19].

Відмінності від мурашиної системи:

1. Рівень феромонів на ребрах оновлюється у кінці чергової ітерації (тобто глобально) як правила

$$\tau_{ij}(n) = (1-\rho_g)\tau_{ij}(n-1) + \rho_g \frac{1}{F(x^*)}, \quad (1.7)$$

але і після проходження маршруту черговою мурахою (тобто локально) у вигляді правила

$$\tau_{ij}(n) = (1-\rho)\tau_{ij}(n-1) + \rho\tau_{ij}(0) \quad (1.8)$$

Для малих значень ρ_g поточна концентрація на дугах відбувається повільно і вплив побудованого кращого шляху послаблюється. З іншого боку, для великих значень ρ_g відкладений феромон випаровується швидко і вплив збудованого кращого шляху посилюється. Це сприяє розширенню простору пошуку. Іноді значення ρ_g дозволяють змінюватися у процесі пошуку рішення: на початковій стадії використовуються великі значення, але в кінцевої – малі.

2. Наприкінці ітерації рівень феромонів підвищується лише на найкоротшому зі знайдених шляхів згідно (1.7). Це сприяє більш спрямованому пошуку, змушуючи мурах рухатися у бік знайдених найкращих рішень.

3. Алгоритм використовує змінене правило переходу: з певною ймовірністю мурашка або вибирає краще (максимально ймовірне) ребро, або робить вибір так само, як і в класичному алгоритмі

4. Імовірність переходу визначається як

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}(n-1)\left(1/d_{ij}\right)^{\beta}}{\sum\limits_{l \in V \setminus V^{tabu}} \tau_{il}(n-1)\left(1/d_{il}\right)^{\beta}}, & j \in V \setminus V^{tabu} \\ 0, & j \notin V \setminus V^{tabu} \end{cases}, \quad (1.9)$$

тобто. коефіцієнт історії (вага, інтенсифікація феромону) $\alpha = 1$

3.3.2.1. Алгоритм для пошуку оптимального маршруту

1. Ініціалізація

1.1. Завдання коефіцієнт евристики β, коефіцієнтів інтенсивності випаровування (керує швидкістю втрати інформації) ρ_l і ρ_g, ймовірності правила переходу p^{trans}, причому $0 < \beta < 1,\ 0 < \rho_l < 1,\ 0 < \rho_g < 1.$

При $\beta = 0$ відбувається вибір лише на підставі феромону .

1.2. Завдання максимальної кількості ітерацій N, розміру популяції K, довжина вектора вершин M .

1.3. Встановлюється безліч вершин $V = \{1,...,M\}$ і матриця ваг ребер $[d_{ij}]$, $i, j \in \overline{1,M}$.

1.4. Завдання функції вартості (функція цілі)

$$F(x) = d_{x_M, x_1} + \sum_{i=1}^{M-1} d_{x_i, x_{i+1}} \to \min_x,$$

де d_{x_M, x_1} - вага ребра $(x_i, x_{i+1}),\ x_i, x_{i+1} \in V$,

x - Вектор вершин.

1.5. Задається, шляхом упорядкування випадковим чином множини V, оптимальний вектор вершин x^*.

1.6. Задається початковий рівень феромону $\tau_{ij}(0)$ на ребрі (i, j) в останній момент часу $n = 0,\ i, j \in \overline{1,M}$. Він ініціалізується невеликим позитивним числом. Наприклад,

$$\tau_{ij}(0) = \frac{1}{M \cdot F(x^*)},\ \tau_{ii}(0) = 0,\ i, j \in \overline{1,M}.$$

2. Номер ітерації $n = 1$.

3. Номер мурахи $k = 1$.

4. Вибір номера стартової вершини одним із двох способів:

– всі мурахи містяться у одну вершину, тобто. $i = 1$

– мурахи містяться у різні вершини випадковим чином, тобто.

$$i = round(1 + (M - 1)rand()),$$

де $rand()$ - функція, що повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$,

$round()$ - Функція, що округлює число до найближчого цілого.

5. Безліч заборонених вершин $V^{tabu} = \{i\}, x_{k1} = i$

6. Обчислюються ймовірності переходу k -го мурахи з вершини i до інших вершин

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}(n-1)\left(1/d_{ij}\right)^{\beta}}{\sum\limits_{l \in V \setminus V^{tabu}} \tau_{il}(n-1)\left(1/d_{il}\right)^{\beta}}, & j \in V \setminus V^{tabu} \\ 0, & j \notin V \setminus V^{tabu} \end{cases}, \quad j \in \overline{1, M}$$

7. Якщо $rand() > p^{trans}$, то вибирається вершина j_c, яка задовольняє нерівності $\sum\limits_{j=1}^{j_c - 1} p_{ij} < rand() \le \sum\limits_{j=1}^{j_c} p_{ij}$, інакше $j_c = \max\limits_{j} p_{ij}$.

8. $V^{tabu} = V^{tabu} \bigcup \{j_c\}, x_{k,|V^{tabu}|} = j_c$

9. Якщо $|V^{tabu}| < M$, то $i = j_c$ перехід до 6.

10. Якщо $F(x_k) < F(x^*)$, то $x^* = x_k$.

11. Правило локальної зміни рівня феромону наведено у вигляді

$$\tau_{ij}(n) = \tau_{ji}(n) = (1-\rho)\tau_{ij}(n-1) + \rho\tau_{ij}(0),\ i \in \overline{1, M-1},\ j \in \overline{i+1, M}$$

12. Якщо $k < K$, то $k = k + 1$ перехід до 4.

13. Правило глобальної зміни рівня феромону наведено у вигляді

$$\tau_{ij}(n) = \tau_{ji}(n) = (1-\rho_g)\tau_{ij}(n-1) + \rho_g \frac{1}{F(x^*)}, \qquad i \in \overline{1, M-1},$$

$j \in \overline{i+1, M}$

14. Якщо $n < N$, то $n = n + 1$ перехід до 3, інакше зупин.

Результатом є x^*.

3.3.3 . Max - Min мурашина система

Max - Min мурашина система (Max - Min ant system) запропонована Штютцлем (Stutzle) і Хоосом (Hoos) [20], є модифікацією мурашиної системи та розроблений для подолання проблеми передчасної стагнації

Відмінності від мурашиної системи:

1. Наприкінці ітерації рівень феромонов підвищується лише найкоротшому (глобально кращому) зі знайдених шляхів, тобто. у вигляді правила

$$\tau_{ij}(n) = (1-\rho_g)\tau_{ij}(n-1) + \rho_g \frac{1}{F(x^*)} \qquad (1.10)$$

Це сприяє більш спрямованому пошуку, змушуючи мурах рухатися у бік знайдених найкращих рішень.

2. Алгоритм використовує обмеження до рівня феромона як

$$\tau_{ij}(n) = \max\{\tau^{\min}, \tau_{ij}(n)\},\ \tau_{ij}(n) = \min\{\tau^{\max}, \tau_{ij}(n)\} \qquad (1.11)$$

3. Імовірність переходу визначається як

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}(n-1)\left(1/d_{ij}\right)^{\beta}}{\sum_{l \in V \setminus V^{tabu}} \tau_{il}(n-1)\left(1/d_{il}\right)^{\beta}}, & j \in V \setminus V^{tabu} \\ 0, & j \notin V \setminus V^{tabu} \end{cases}, \qquad (1.12)$$

тобто. коефіцієнт історії (вага, інтенсифікація феромону) $\alpha = 1$

3.3.3.1. Алгоритм для пошуку оптимального маршруту

1. Ініціалізація

1.1. Завдання коефіцієнта історії (ваги феромону) α, коефіцієнт евристики β, коефіцієнта інтенсивності випаровування (керує швидкістю втрати

інформації) ρ_g, мінімального та максимального рівня феромону $\tau^{\min}, \tau^{\max}$, причому $0 < \alpha < 1,\ \beta \in [2,5], 0 < \rho < 1$

При $\beta = 0$ відбувається вибір лише на підставі феромону .

1.2. Завдання максимальної кількості ітерацій N, розміру популяції K, довжина вектора вершин M .

1.3. Встановлюється безліч вершин $V = \{1,...,M\}$ і матриця ваг ребер $[d_{ij}]$, $i, j \in \overline{1,M}$.

1.4. Завдання функції вартості (функція цілі)

$$F(x) = d_{x_M, x_1} + \sum_{i=1}^{M-1} d_{x_i, x_{i+1}} \to \min_x,$$

де d_{x_M, x_1} - вага ребра (x_i, x_{i+1}), $x_i, x_{i+1} \in V$,

x - Вектор вершин.

1.5. Задається, шляхом упорядкування випадковим чином множини V, оптимальний вектор вершин x^*.

1.6. Задається початковий рівень феромону $\tau_{ij}(0)$ на ребрі (i, j) в останній момент часу $n = 0,\ i, j \in \overline{1,M}$. Він ініціалізується невеликим позитивним числом. Наприклад,

$$\tau_{ij}(0) = \frac{M}{F(x^*)},\ \tau_{ii}(0) = 0,\ i, j \in \overline{1,M}.$$

2. Номер ітерації $n = 1$.

3. Номер мурахи $k = 1$.

4. Вибір номера стартової вершини одним із двох способів:

– всі мурахи містяться у одну вершину, тобто. $i = 1$

– мурахи містяться у різні вершини випадковим чином, тобто.

$$i = round(1 + (M - 1)rand()),$$

де $rand()$ - функція, що повертає рівномірно розподілене випадкове число в діапазоні $[0,1]$,

$round()$ - Функція, що округлює число до найближчого цілого.

5. Безліч заборонених вершин $V^{tabu} = \{i\}, x_{k1} = i$

6. Обчислюються ймовірності переходу k -го мурахи з вершини i до інших вершин

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}(n-1)\left(1/d_{ij}\right)^{\beta}}{\sum\limits_{l \in V \setminus V^{tabu}} \tau_{il}(n-1)\left(1/d_{il}\right)^{\beta}}, & j \in V \setminus V^{tabu} \\ 0, & j \notin V \setminus V^{tabu} \end{cases}, \quad j \in \overline{1, M}$$

7. Вибирається вершина j_c, яка задовольняє нерівності

$$\sum_{j=1}^{j_c-1} p_{ij} < rand() \le \sum_{j=1}^{j_c} p_{ij}$$

8. $V^{tabu} = V^{tabu} \bigcup \{j_c\}, x_{k,|V^{tabu}|} = j_c$

9. Якщо $|V^{tabu}| < M$, то $i = j_c$ перехід до 6.

10. Якщо $F(x_k) < F(x^*)$, то $x^* = x_k$.

11. Якщо $k < K$, то $k = k + 1$ перехід до 4.

12. Правило зміни рівня феромону подане у вигляді

$$\tau_{ij}(n) = \tau_{ji}(n) = (1-\rho_g)\tau_{ij}(n-1) + \rho_g \frac{1}{F(x^*)}, \qquad i \in \overline{1, M-1},$$

$j \in \overline{i+1, M}$

13. Обмеження на рівень феромону

$$\tau_{ij}(n) = \max\{\tau^{\min}, \tau_{ij}(n)\}, \tau_{ij}(n) = \min\{\tau^{\max}, \tau_{ij}(n)\}$$

14. Якщо $n < N$, то $n = n + 1$ перехід до 3, інакше зупин.

Результатом є x^*.

Крім наведених трьох мурашиних алгоритмів використовуються також система елітних мурах, ранжована мурашина система, Q мурашина система, швидка мурашина система, апроксимований недетермінований деревоподібний пошук та ін.

3.3.4. Параметри мурашиних алгоритмів

Ефективність мурашиних алгоритмів залежить від ряду параметрів, що управляють:

1. Число мурах K істотно впливає характеристики мурашиних алгоритмів – очевидно велике число K веде до більшої обчислювальної складності. Чим більше мурах використовується, тим більше шляхів будується і відкладається більше феромону . Наприклад, обчислювальна складність системи мурашиної колонії оцінюється як $O(n_c, M^2, K)$, де $n_c = N \cdot K$ загальна кількість циклів, N число ітерацій, M довжина рішення. Успішне застосування мурашиних алгоритмів обумовлено, насамперед, спільною поведінкою безлічі мурах. Завдяки відкладаному феромону , мурахи передають отриманий досвід та знання. Чим менше використовується мурах, тим слабша здатність алгоритму до дослідження і, отже, менше інформації про простір пошуку доступно іншим мурах. Невелика кількість мурах може викликати передчасну стагнацію або перебування субоптимальних рішень. Експериментально показано, що при розв'язанні задачі комівояжера число мурах, порівнянне з числом вершин графа $K \approx M$ дає хороші результати. Також хороші результати для завдання комівояжера досягається при $K = 10$. Слід наголосити, що ці оцінки отримані тільки для певного алгоритму, і в загальному випадку оптимальні значення K можуть бути різними для різних завдань.

2. Максимальна кількість ітерацій N відіграє важливу роль для пошуку якісних рішень. При малому числі N мурах може не вистачити часу для побудови оптимального шляху. З іншого боку, якщо N занадто велике, будуть зроблені зайві обчислення.

3. Значення початкової концентрації $\tau_{ij}(0)$ також впливають на характеристики мурашиних алгоритмів . При початковій ініціалізації дугам зазвичай надається або мале постійне позитивне значення $\tau_{ij}(0)$. Велике значення $\tau_{ij}(0)$ у разі випадкового вибору може давати великі відмінності початкової концентрації, що може призвести до початкового вибору неперспективного рішення.

Загалом при вирішенні деякої проблеми з використанням мурашиних алгоритмів бажано провести експериментальні дослідження з метою оптимізації параметрів, що управляють.

3.3.5. Застосування мурашиних алгоритмів

Мурашині алгоритми використовувалися під час вирішення багатьох реальних завдань, передусім, завдань комбінаторної оптимізації, у тому числі найпопулярнішою є завдання комівояжера. Для того щоб розробити мурашиний алгоритм для вирішення конкретного завдання, необхідно:

1. Відповідне подання у вигляді графа для опису дискретного простору пошуку. Граф повинен представляти всі стани та переходи між ними. Необхідна схема представлення потенційного рішення.

2. Визначити правила корекції концентрації феромону , які визначають позитивний зворотний зв'язок у процесі.

3. При необхідності розробити евристику визначення переваги дуги в графі.

4. Визначити евристику поведінки мурашки під час побудови рішення як ймовірності переходу.

5. Визначити засоби перевірки здійсненності потенційного рішення з урахуванням обмежень задачі.

6. Визначити основні параметри мурашиного алгоритму (кількість штучних мурах тощо).

Оскільки в основі мурашиних алгоритмів лежить пересування мурах по деяких шляхах, то мурашині алгоритми ефективні, перш за все, при вирішенні завдань, які припускаю інтерпретацію у вигляді графа. Проведені комп'ютерні експерименти показали, що ефективність мурашиних алгоритмів зростає зі збільшенням розмірності завдання й у завдань на графах високої розмірності вони працюють швидше, ніж інші еволюційні алгоритми. Відзначено також хороші результати при вирішенні нестаціонарних завдань на графах із середовищем, що змінюється.

В даний час мурашині алгоритми отримали застосування при вирішенні наступних практичних завдань:

– маршрутизація (насамперед у мережах);

- Завдання про призначення;

- Машинне навчання;

- Кластеризація даних;

- Транспортна маршрутизація;

– календарне планування та розклад;

- Завдання про покриття;

- Біоінформатика та ін.

3.3.6. Реалізація мурашиної системи для задачі комівояжера в пакеті Matlab

```
% множину вершин (міст), представлених координатами
cities = [
[565,575]; [25,185]; [345,750]; [945,685]; [845,655];
[880,660]; [25,230]; [525,1000]; [580,1175]; [650,1130];
```

```
[1605,620]; [1220,580]; [1465,200]; [1530,5]; [845,680];
[725,370]; [145,665]; [415,635]; [510,875]; [560,365];
[300,465]; [520,585]; [480,415]; [835,625]; [975,580];
[1215,245]; [1320,315]; [1250,400]; [660,180]; [410,250];
[420,555]; [575,665]; [1150,1160];[700,580]; [685,595];
[685,610]; [770,610]; [795,645]; [720,635]; [760,650];
[475,960]; [95,260]; [875,920]; [700,500]; [555,815];
[830,485]; [1170,65]; [830,610]; [605,625]; [595,360];
[1340,725]; [1740,245]];
max_it = 100; % максимальна кількість ітерацій
num_ants = 30; % кількість мурах
ro = 0.6; % параметр ro
beta = 2.5; % параметр beta
alpha = 1.0; % параметр alpha
best1 = AntSearch(cities, max_it, num_ants, ro, beta, alpha);

% мурашиний алгоритм
function best = AntSearch(cities, max_it, num_ants, ro, beta, alpha)
% визначення кількості вершин
num_cities=size(cities,1);
% випадковим чином генерується рішення
best.vector=randperm(num_cities);
% обчислення вартості рішення
best.cost=Cost(best.vector, cities);
% ініціалізація рівня феромону
pheromone=InitialisePheromone(num_cities, best.cost);
solutions=[];
for iter=1:max_it
    % формування рішення та обчислення його вартості для мурах
    for k=1:num_ants
```

```
% формування рішення для поточної мурашки
candidate.vector=DecisionGenerating(cities, pheromone, beta, alpha);
% обчислення вартості рішення для поточної мурашки
candidate.cost=Cost(candidate.vector, cities);
% порівняння отриманого рішення з найкращим
if candidate.cost<best.cost
        best=candidate;
end
solutions(k).vector=candidate.vector;
solutions(k).cost=candidate.cost;
кандидат.вектор=[];
end
% зміни рівня феромону
pheromone=UpdatePheromone(pheromone, ro, solutions, num_ants);
solutions=[];
end

% ініціалізація рівня феромону
function pheromone = InitialisePheromone(num_cities, naive_score)
    v=num_cities/naive_score;
    for i=1:num_cities
      for j=1:num_cities
       pheromone(i,j)=v;
end
end

% формування рішення для поточної мурашки
function perm = DecisionGenerating(cities, pheromone, beta, alpha)
% визначення кількості вершин
num_cities=size(cities,1);
```

```
% випадковим чином вибирається вершина
perm(1)=round(1+(num_cities-1)*rand);
i=1;
for i=2:num_cities
% обчислення ймовірностей переходу
% для всіх незаборонених вершин
choices=CalculateChoices(cities, perm(i-1), perm, pheromone, beta,  alpha);
% вибір незабороненої вершини для списку заборон
perm(i)=SelectNextCity(choices);
end


% зміни рівня феромону
function pheromone = UpdatePheromone(pheromone, ro, solutions, num_ants)
% визначення кількості вершин
num_cities=size(pheromone,1);
for i=1:num_cities
  for j=1:num_cities
        pheromone(i,j)=(1.0-ro)*pheromone(i,j);
end
end
for k=1:num_ants
 for i=1:num_cities
  x=solutions(k).vector(i);
if i==num_cities
      y=solutions(k).vector(1);
else
y=solutions(k).vector(i+1);
end
pheromone(x,y)=pheromone(x,y)+(1.0/solutions(k).cost);
pheromone(y,x)=pheromone(y,x)+(1.0/solutions(k).cost);
```

```
end
end

% обчислення ймовірностей переходу для всіх незаборонених вершин
function choices=CalculateChoices(cities, last_city, exclude, pheromone, beta, alpha)
% визначення кількості вершин
num_cities=size(cities,1);
j=1;
for i=1:num_cities
% якщо вершина незаборонена
if length(find(exclude==i))==0
% обчислення ваги ребра
distance=Distance(cities(last_city,:), cities(i,:));
% обчислення ймовірності переходу
prob=(pheromone(last_city,i).^beta)*((1/distance).^alpha);
choices(j).city=i;
choices(j).prob=prob;
j=j+1;
end
end
% вибір незабороненої вершини для списку заборон
function next_city = SelectNextCity(choices)
% визначення кількості незаборонених вершин
num_choices=size(choices,1);
% обчислення суми ймовірностей
sum=0.0;
for i=1:num_choices
sum=sum+choices(i).prob;
end
```

```
% вибір незабороненої вершини для списку заборон
i=1;
while i<=num_choices
v=rand;
if v<=choices(i).prob/sum
next_city=choices(i).city;
return;
end
i=i+1;
if i>num_choices
   i=1;
end
end

% обчислення вартості рішення
function cost = Cost(permutation, cities)
% визначення кількості вершин
num_cities=size(cities,1);
cost=0;
for i=1:num_cities
c1=permutation(i);
if i==num_cities
      c2=пермутація(1);
else
c2=пермутація(i+1);
end
         cost=cost+Distance(cities(c1,:),cities(c2,:));
end
```

% евклідова відстань між двома вершинами

function distance=Distance(c1, c2)

distance=sqrt(sum((c1-c2).^2));

3.3.7 Приклад використання мурашиного алгоритму на Python

Реалізація алгоритму мурашиної системи для розв'язання задачі комівояжера (TSP) мовою Python вимагає декількох кроків. Нижче наведено приклад коду на Python, який допоможе вам розпочати роботу. Цей код передбачає спрощену задачу з невеликою кількістю міст і демонструє базову структуру алгоритму мурашиної системи.

```python
import numpy as np

# Define the number of cities and ants
num_cities = 5
num_ants = 10

# Define the distance matrix (replace this with your actual distance matrix)
distance_matrix = np.array([
    [0, 2, 9, 10, 7],
    [2, 0, 6, 4, 8],
    [9, 6, 0, 8, 3],
    [10, 4, 8, 0, 5],
    [7, 8, 3, 5, 0]
])

# Initialize parameters
pheromone_matrix = np.ones((num_cities, num_cities))
alpha = 1.0  # Importance of pheromone
beta = 2.0   # Importance of distance
rho = 0.5    # Pheromone evaporation rate
Q = 100      # Pheromone deposit constant
iterations = 100

# Main loop
for _ in range(iterations):
    ant_tours = []
    for ant in range(num_ants):
        current_city = np.random.randint(num_cities)
        visited_cities = [current_city]
        tour_length = 0

        for _ in range(num_cities - 1):
            # Calculate the probability of moving to each unvisited city
            unvisited_cities = [city for city in range(num_cities) if city not in visited_cities]
            probabilities = []

```

Рис 3.15 мурашковий алгоритм

```
            for city in unvisited_cities:
                pheromone = pheromone_matrix[current_city, city]
                distance = distance_matrix[current_city, city]
                probability = (pheromone ** alpha) * ((1.0 / distance) ** beta)
                probabilities.append(probability)

            # Select the next city based on the probabilities
            selected_city = np.random.choice(unvisited_cities, p=(probabilities / np.sum(probabilities)))
            visited_cities.append(selected_city)
            tour_length += distance_matrix[current_city, selected_city]
            current_city = selected_city

        # Complete the tour by returning to the starting city
        tour_length += distance_matrix[current_city, visited_cities[0]]
        ant_tours.append((visited_cities, tour_length))

    # Update pheromone levels
    pheromone_delta = np.zeros((num_cities, num_cities))
    for tour in ant_tours:
        tour_path, tour_length = tour
        for i in range(len(tour_path) - 1):
            from_city = tour_path[i]
            to_city = tour_path[i + 1]
            pheromone_delta[from_city, to_city] += Q / tour_length

    pheromone_matrix = (1 - rho) * pheromone_matrix + pheromone_delta

# Find the best tour
best_tour = min(ant_tours, key=lambda x: x[1])
print("Best tour:", best_tour)
```

Рис 3.16 мурашковий алгоритм

Результат:

```
Best tour: ([1, 3, 4, 2, 0], 23)


...Program finished with exit code 0
Press ENTER to exit console.
```

Рис 3.17 мурашковий алгоритм

ВИСНОВКИ

В ході виконання магістерської роботи було проведено дослідження метаевристичних методів розв'язання оптимізаційних завдань пошуку оптимального маршруту

Були проаналізовані алгоритми метаевристичних методів розв'язання оптимізаційних завдань та пошуку оптимального маршруту і встановлено їх актуальність

В результаті досліджень були розроблені приклади застосування метаевристичних методів розв'язання оптимізаційних завдань та пошуку оптимального маршруту алгоритмів за допомогою бібліотека мови програмування Python

СПИСОК ЛІТЕРАТУРИ

1. Stutzle T.G. Analyzing the run-time behavior of iterated local search for the TSP / T.G. Stutzle, H.H. Hoos // Proc. III Metaheuristics International Conference. – Angra dos Reis, 1999. – P. 1-18.

2. Lourenço, H. R., Martin, O. C., & Stützle, T. (2003). Iterated local search. B Handbook of Metaheuristics (ст. 321-353). Springer.

3. Resende, M. G. C., & Ribeiro, C. C. (2003). Greedy randomized adaptive search procedures. B Handbook of Metaheuristics (ст. 219-249). Springer.

4. Pohl, E. A. (1970). Biased random-walk models for local search algorithms. Management Science, 16(6), 357-368.

5. Mladenović N. Variable neighborhood search / N. Mladenović, P. Hansen // Computers & Operations Research. – 1997. – Vol. 24, № 11. – P. 1097-1100.

6. Mladenović, N., & Hansen, P. (1997). Variable neighborhood search. Computers & Operations Research, 24(11), 1097-1100.

7. Mladenović, N., & Hansen, P. (1997). Variable neighborhood search for the p-median. Location Science, 5(4), 207-221.

8. Petch, R., Salhi, S., & Nagy, G. (2000). A variable neighborhood search for the p-median. Location Science, 8(1-2), 87-97.

9. Feo T.A. A probabilistic heuristic for a computationally difficult set covering problem / T.A. Feo, M.G.C. Resende // Operations Research Letters. – 1989. – Vol. 8. – P. 67-71.

10. Resende, M. G. C., & Ribeiro, C. C. (2003). Greedy randomized adaptive search procedures. B Handbook of Metaheuristics (ст. 219-249). Springer.

11. Resende, M. G. C., & Ribeiro, C. C. (2003). Greedy randomized adaptive search procedures. B Handbook of Metaheuristics (ст. 219-249). Springer.

12. Pohl, E. A., & Thomas, L. J. (1994). A greedy randomized adaptive search procedure for the job-shop scheduling problem. Management Science, 40(9), 1229-1241.

13. Mucherino A. Monkey Search: A Novel Meta-Heuristic Search for Global Optimization / A. Mucherino, O. Seref // “Data Mining, System Analysis and Optimization in Biomedicine”, AIP Conference Proceedings – 2007. – P. 162-173.

14. Kirkpatrick S. Optimization by simulated annealing / S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi // Science. – Vol. 220. – 1983. – P. 671–680.

15. Eberhart R.C. A new optimizer using particle swarm theory / R.C. Eberhart, J. Kennedy // Proc. of the sixth international symposium on micro machine and human science. – 1995. – P. 39-43.

16. Chu S.C. Cat swarm optimization / S.C. Chu, P.W. Tsai, J.S. Pan // Proc. of the 9th Pacific Rim International Conference on Artificial Intelligence. – 2006. – P. 854-858.

17. Dorigo M. The ant system: Optimization by a colony of cooperating agents / M. Dorigo // IEEE Transactions on systems, man and Cybernetics. – Vol. 26, № 1. – 1996. – P. 1-13

18. Dorigo M. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem / M. Dorigo, L.M. Gambardella // IEEE Transactions on Evolutionary Computation. – 1997. – Vol. 1, № 1. – P. 53-66.

19. Dorigo, M., & Gambardella, L. M. (1997). Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. IEEE Transactions on Evolutionary Computation, 1(1), 53–66. doi: 10.1109/4235.585892

20. Stutzle T. MAX–MIN ant system, future generation computer systems / T. Stutzle, H.H. Hoos // Future Generation Computer Systems. – 2000. – Vol. 16. – P. 889–914. 9. Stutzle T. MAX–MIN ant system, future generation computer systems / T. Stutzle, H.H. Hoos // Future Generation Computer Systems. – 2000. – Vol. 16. – P. 889–914.

Додаток 2 до наказу
від «31» березня 2023 року
№119/05

ДЕКЛАРАЦІЯ
про дотримання академічної доброчесності

Я, _____
_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)