**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 5 з дисципліни

«Проектування алгоритмів»

„**Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-11 Головня О.Р.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc52291748)

[2 Завдання 4](#_Toc52291749)

[3 Виконання 10](#_Toc52291750)

[3.1 Покроковий алгоритм 10](#_Toc52291751)

[3.2 Програмна реалізація алгоритму 10](#_Toc52291752)

[3.2.1 Вихідний код 10](#_Toc52291753)

[3.2.2 Приклади роботи 10](#_Toc52291754)

[3.3 Тестування алгоритму 11](#_Toc52291755)

[Висновок 12](#_Toc52291756)

[Критерії оцінювання 13](#_Toc52291757)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму.

# Завдання

Згідно варіанту, формалізувати алгоритм вирішення задачі відповідно загальної методології.

Записати розроблений алгоритм у покроковому вигляді. З достатнім степенем деталізації.

Виконати його програмну реалізацію на будь-якій мові програмування.

Перелік задач наведено у таблиці 2.1.

Перелік алгоритмів і досліджуваних параметрів у таблиці 2.2.

Задача і алгоритм наведені в таблиці 2.3.

Змінюючи параметри алгоритму, визначити кращі вхідні параметри алгоритму. Для цього необхідно:

* обрати критерій зупинки алгоритму (кількість ітерацій або значення ЦФ);
* зафіксувати усі параметри крім одного і змінювати цей параметр, поки не буде досягнуто пікової ефективності;
* після цього параметр фіксується і змінюються інші параметри;
* далі повторюємо процедуру спочатку, з першого зафіксованого параметру;
* зупиняємось коли будуть знайдені оптимальні параметри для даної задачі або встановлена залежність одних параметрів від інших.

Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

|  |  |
| --- | --- |
| **№** | **Задача** |
| 2 | **Задача комівояжера** (300 вершин, відстань між вершинами випадкова від 5 до 150) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів.  **Розглядається симетричний, асиметричний та змішаний варіанти.**  В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напряму, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра.  У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.  У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху.  Застосування:   * доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); * доставка води; * моніторинг об'єктів; * поповнення банкоматів готівкою; * збір співробітників для доставки вахтовим методом. |

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

|  |  |
| --- | --- |
| **№** | **Алгоритми і досліджувані параметри** |
| 2 | **Мурашиний алгоритм**:   * α; * β; * ρ; * Lmin; * кількість мурах М і їх типи (елітні, тощо…); * маршрути з однієї чи різних вершин. |

Таблиця 2.3 – Варіанти задач і алгоритмів

|  |  |
| --- | --- |
| **№** | **Задачі і алгоритми** |
| 7 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |

# Виконання

Варіант 7

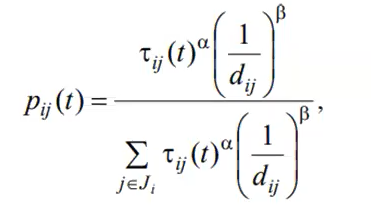
## Алгоритм

**def initilization**():

ініціалізація матриці відстаней

Probabilities()

Розрахунок ймовірності переходу з вершини



def Probabilities():

**для** i **в діапазоні**(0, кількість вершин):

**для** j **в діапазоні** (0, кількість вершин \_nodes):

numerator=pheromone\_matrix[i][j]\*\*ALPHA\*

\*self.visibility\_matrix[i][j] \*\*BETA

denominator = 0

**для** k **в діапазоні**(0, кількість вершин \_nodes):

denominator +=pheromone\_matrix[i][k] \*\*ALPHA \*

self.visibility\_matrix[i][k] \*\*BETA

self.probability\_matrix[i][j] = numerator / denominator

**def Stats\_for\_best**(paths):

#Оцінює рішення мурашок, додаючи відстані між вузлами.

best = float('inf')

scores = np.zeros(len(paths))

**для** index, path **в перелічених**(paths):

score = 0

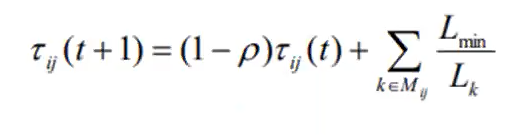
**для** i **в діапазоні**(0, len(path) - 1):

score += map[path[i], path[i + 1]]

scores[index] = score

best = np.argmin(scores)

**повернути** paths[best], scores[best], scores



Функції підрахунку випаровування та рівню феромону

**def evaporation**():

**def lvl\_pheromone**():

**def fit**(map\_matrix):

map = map\_matrix

initialize()

num\_equal = 0

for i in range(ITERATION\_LIMIT):

start\_iter = time.time()

all\_paths = []

ant\_type = "default"

**для** ant **в діапазоні**(ANTS\_AMOUNT + WILD\_ANTS\_AMOUNT):

**якщо** ant >= ANTS\_AMOUNT:

ant\_type = "wild"

path = []

**якщо** START\_FROM\_DIFFERENT\_POINTS:

current\_node = available\_nodes[np.random.randint(0,

len(self.available\_nodes))]

**інакше**:

current\_node = self.available\_nodes[0]

start\_node = current\_node

**поки** True:

path.append(current\_node)

available\_nodes.remove(current\_node)

**якщо** len(self.available\_nodes) != 0:

current\_node\_index = self.choose\_next\_node(current\_node,

ant\_type)

current\_node = self.available\_nodes[current\_node\_index]

**інакше**:

**вихід з циклу**

path.append(start\_node)

available\_nodes = list(range(map.shape[0]))

all\_paths.append(path)

best\_path, best\_score, all\_scores =evaluate\_distance(all\_paths)

**якщо** i == 0:

self.best\_score = best\_score

self.best\_path = best\_path

**інакше**:

**якщо** best\_score < self.best\_score:

self.best\_score = best\_score

self.best\_path = best\_path

**якщо** best\_score == self.best\_score:

num\_equal += 1

**інакше**:

num\_equal = 0

all\_best\_scores.append(best\_score)

evaporation()

intensify(all\_scores, all\_paths)

get\_probabilities()

**Повернути** self.best\_score

## Програмна реалізація алгоритму

### Вихідний код

class AntColony:

def fit(self, map\_matrix):

self.map = map\_matrix

start = time.time()

self.initialize()

#################################

num\_equal = 0

for i in range(self.ITERATION\_LIMIT):

start\_iter = time.time()

all\_paths = []

ant\_type = "default"

for ant in range(self.ANTS\_AMOUNT + self.WILD\_ANTS\_AMOUNT):

if ant >= self.ANTS\_AMOUNT:

ant\_type = "wild"

path = []

if self.START\_FROM\_DIFFERENT\_POINTS: #Початок з різних точок чи навпаки

current\_node = self.available\_nodes[np.random.randint(0, len(self.available\_nodes))]

else:

current\_node = self.available\_nodes[0]

start\_node = current\_node

while True:

path.append(current\_node)

self.available\_nodes.remove(current\_node)

if len(self.available\_nodes) != 0:

current\_node\_index = self.choose\_next\_node(current\_node, ant\_type)

current\_node = self.available\_nodes[current\_node\_index]

else:

break

path.append(start\_node) # Назад в початкову точку

self.available\_nodes = list(range(self.map.shape[0]))

all\_paths.append(path)

best\_path, best\_score, all\_scores = self.Stats\_for\_best(all\_paths)#Статистика

if i == 0:

self.best\_score = best\_score

self.best\_path = best\_path

else:

if best\_score < self.best\_score:

self.best\_score = best\_score

self.best\_path = best\_path

if best\_score == self.best\_score:

num\_equal += 1

else:

num\_equal = 0

self.all\_best\_scores.append(best\_score)

self.evaporation()#Підрахунок випаровування

self.lvl\_pheromone(all\_scores, all\_paths)#Інтенсивність або ж рівень феромону

self.Probabilities()

print("Найкраща відстань на ітерації {}: {};".format(i, round(best\_score)))#({:.2f}s), time.time() - start\_iter

self.fit\_time = (time.time() - start)

print("Найкраща відстань: {}. \n".format(self.best\_score))#{} round(self.fit\_time, 2) {} self.best\_path

return self.best\_score

def initialize(self):

#Ініціалізує модель, створюючи різні матриці та генеруючи список доступних вузлів

num\_nodes = self.map.shape[0]

self.available\_nodes = list(range(num\_nodes))

self.pheromone\_matrix = np.zeros((num\_nodes, num\_nodes)) #Матриця заповнена нулями

# видалити діагональ, оскільки немає феромону від вузла i до самого себе

self.pheromone\_matrix[np.eye(num\_nodes) == 0] = 0.1

self.visibility\_matrix = 1 / self.map

self.probability\_matrix = np.zeros((num\_nodes, num\_nodes))

self.Probabilities() #Функція розрахунку P ймовірність переходу шляхом

def Probabilities(self):

#Отримання матриці ймовірностей переходу в сусідню вершину

num\_nodes = self.map.shape[0]

for i in range(num\_nodes):

for j in range(num\_nodes):

numerator = self.pheromone\_matrix[i][j] \*\* self.ALPHA \* self.visibility\_matrix[i][j] \*\* self.BETA

denominator = 0

for k in range(num\_nodes):

denominator += self.pheromone\_matrix[i][k] \*\* self.ALPHA \* self.visibility\_matrix[i][k] \*\* self.BETA

self.probability\_matrix[i][j] = numerator / denominator

def choose\_next\_node(self, from\_node, ant\_type="default"):

#Вибирає наступний вузол на основі ймовірностей

probabilities = self.probability\_matrix[from\_node, self.available\_nodes]

if ant\_type == "default":

probabilities /= sum(probabilities)

next\_node = np.random.choice(range(len(probabilities)), p=probabilities)

else:

next\_node = np.random.choice(range(len(probabilities)))

return next\_node

def Stats\_for\_best(self, paths):

#Оцінює рішення мурашок, додаючи відстані між вузлами.

best = float('inf')

scores = np.zeros(len(paths))

for index, path in enumerate(paths):

score = 0

for i in range(len(path) - 1):

score += self.map[path[i], path[i + 1]]

scores[index] = score

best = np.argmin(scores)

return paths[best], scores[best], scores

def evaporation(self):

#Випаровування феромону.

self.pheromone\_matrix \*= (1 - self.EVAPORATION\_RATE)

def lvl\_pheromone(self, scores, paths):

#Збільшення феромону на пройдених шляхах, і на кращому шляху, якщо присутні елітні мурахи.

i = self.best\_path[:-1]

j = self.best\_path[1:]

self.pheromone\_matrix[i, j] += self.ELITE\_ANTS\_AMOUNT \* (self.L\_MIN / self.best\_score)

for index, score in enumerate(scores):

i = paths[index][:-1]

j = paths[index][1:]

self.pheromone\_matrix[i, j] += self.L\_MIN / score

def Create\_graph(self):

fig, ax = plt.subplots(figsize=(10, 5))

ax.plot(self.all\_best\_scores, label="Best Run")

ax.set\_xlabel("Ітерація")

ax.set\_ylabel("Відстань")

plt.show()

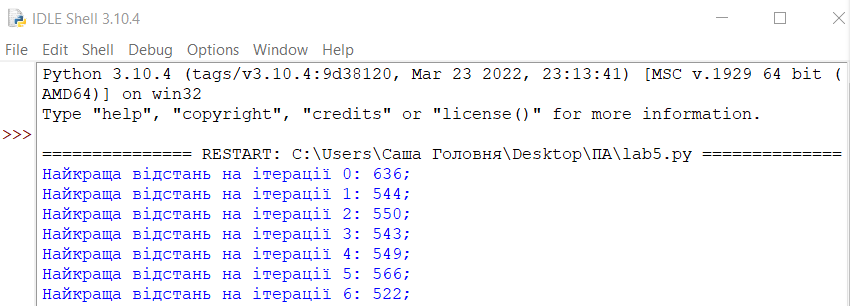
A = np.random.randint(5, 150, size=(30, 30))

MyAntColony = AntColony()

best = MyAntColony.fit(A)

#MyAntColony.Create\_graph()

### Приклади роботи



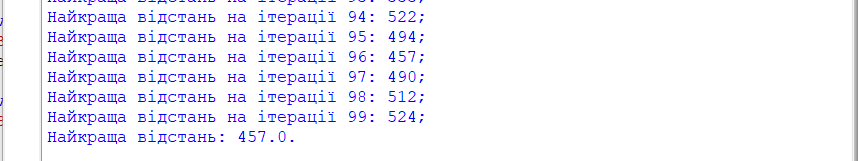


Рисунок 3.1 – Приклад роботи програми

## Тестування алгоритму

Щоб визначити оптимальні дані, я зафіксую усі крім одного, та буду проводити декілька тестів.

Таблиця 3.1 – Тестування алгоритму

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ELITE\_ANTS | WILD\_ANTS | EVAPORATION\_RATE | ALPHA | BETA | L\_MIN | BEST DISTANCE | | |  |
| 1 | 1 | 0.2 | 1 | 3 | 30 | 63 | 67 | 62 | 64 |
| 2 | 1 | 0.2 | 1 | 3 | 30 | 69 | 76 | 66 | 70,33333 |
| 5 | 1 | 0.2 | 1 | 3 | 30 | 64 | 75 | 62 | 67 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 1 | 0.2 | 1 | 3 | 30 | 68 | 62 | 74 | 68 |
| 1 | 2 | 0.2 | 1 | 3 | 30 | 59 | 63 | 64 | 62 |
| 1 | 3 | 0.2 | 1 | 3 | 30 | 66 | 57 | 74 | 65,66667 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 2 | 0.1 | 1 | 3 | 30 | 70 | 62 | 74 | 68,66667 |
| 1 | 2 | 0.2 | 1 | 3 | 30 | 62 | 73 | 70 | 68,33333 |
| 1 | 2 | 0.3 | 1 | 3 | 30 | 73 | 61 | 72 | 68,66667 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 2 | 0.2 | 1 | 3 | 30 | 63 | 71 | 65 | 66,33333 |
| 1 | 2 | 0.2 | 2 | 3 | 30 | 71 | 65 | 67 | 67,66667 |
| 1 | 2 | 0.2 | 3 | 3 | 30 | 80 | 88 | 75 | 81 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 2 | 0.2 | 1 | 1 | 30 | 64 | 56 | 81 | 67 |
| 1 | 2 | 0.2 | 1 | 2 | 30 | 60 | 71 | 76 | 69 |
| 1 | 2 | 0.2 | 1 | 3 | 30 | 68 | 77 | 64 | 69,66667 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 2 | 0.2 | 1 | 1 | 25 | 64 | 70 | 68 | 67,33333 |
| 1 | 2 | 0.2 | 1 | 1 | 35 | 58 | 58 | 56 | 57,33333 |
| 1 | 2 | 0.2 | 1 | 1 | 45 | 54 | 68 | 73 | 65 |

Щодо тестування алгоритму: для коефіцієнтів α, β, ρ краще підходять значення: 1, 1, 0.2 відповідно

має бути трішки більше за найкращий прогнозований шлях

Кількість елітних мурах краще встановити на рівні 2% від звичайних мурах. Якщо взяти занадто багато, то мурахи не будуть приділяти достатню увагу іншим

Дикі мурахи є абсолютно хаотичними у виборі міста, що з одного боку, є корисним, оскільки з’являється шанс перевірити нові шляхи, а з іншого боку(якщо диких багато) роблять вирішення неправильним

Починати завжди краще з різних, випадкових міст, що інтуїтивно зрозуміло

**Висновок:** В рамках даної лабораторної роботи я вивчив основний підхід розробки метаеврестичного алгоритму мурашиного алгоритму для типової задачі комівояжера. Опрацював методологію підбору прийнятних параметрів алгоритму.