**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-12 Йолкін Даніїл Сергійович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

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** |
| 1 | **Задача про рюкзак** (місткість P=500, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 20 (випадкова)). Для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб сумарна вага не перевищувала задану, а сумарна цінність була максимальною.  Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика. |
| 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. |
| 3 | **Розфарбовування графа** (300 вершин, степінь вершини не більше 30, але не менше 2) – називають таке приписування кольорів (або натуральних чисел) його вершинам, що ніякі дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають хроматичне число.  Застосування:   * розкладу для освітніх установ; * розкладу в спорті; * планування зустрічей, зборів, інтерв'ю; * розклади транспорту, в тому числі - авіатранспорту; * розкладу для комунальних служб; |
| 4 | **Задача вершинного покриття** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Вершинне покриття для неорієнтованого графа G = (V, E) - це множина його вершин S, така, що, у кожного ребра графа хоча б один з кінців входить в вершину з S.  Задача вершинного покриттяполягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).  На вході: Граф G = (V, E).  Результат: множина C ⊆ V - найменше вершинне покриття графа G.    Застосування:   * розміщення пунктів обслуговування; * призначення екіпажів на транспорт; * проектування інтегральних схем і конвеєрних ліній. |
| 5 | **Задача про кліку** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Клікою в неорієнтованому графі називається підмножина вершин, кожні дві з яких з'єднані ребром графа. Іншими словами, це повний підграф первісного графа. Розмір кліки визначається як число вершин в ній.  Задача про кліку існує у двох варіантах: у **задачі розпізнавання** потрібно визначити, чи існує в заданому графі G кліка розміру k, тоді як в **обчислювальному варіанті** потрібно знайти в заданому графі G кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити).  Застосування:   * біоінформатика; * електротехніка; |
| 6 | **Задача про найкоротший шлях** (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях.  Важливість задачі визначається її різними практичними застосуваннями. Наприклад, в GPS-навігаторах здійснюється пошук найкоротшого шляху між точкою відправлення і точкою призначення. Як вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Якщо сума довжин доріг між перехрестями мінімальна, тоді знайдений шлях найкоротший. |

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

|  |  |
| --- | --- |
| **№** | **Алгоритми і досліджувані параметри** |
| 1 | **Генетичний алгоритм:**   * оператор схрещування (мінімум 3); * мутація (мінімум 2); * оператор локального покращення (мінімум 2). |
| 2 | **Мурашиний алгоритм**:   * α; * β; * ρ; * Lmin; * кількість мурах М і їх типи (елітні, тощо…); * маршрути з однієї чи різних вершин. |
| 3 | **Бджолиний алгоритм:**   * кількість ділянок; * кількість бджіл (фуражирів і розвідників). |

Таблиця 2.3 – Варіанти задач і алгоритмів

|  |  |
| --- | --- |
| **№** | **Задачі і алгоритми** |
| 1 | Задача про рюкзак + Генетичний алгоритм |
| 2 | Задача про рюкзак + Бджолиний алгоритм |
| 3 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 4 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 5 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 6 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 7 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 8 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |
| 9 | Задача вершинного покриття + Генетичний алгоритм |
| 10 | Задача вершинного покриття + Бджолиний алгоритм |
| 11 | Задача комівояжера (асиметрична мережа) + Бджолиний алгоритм |
| 12 | Задача комівояжера (симетрична мережа) + Бджолиний алгоритм |
| 13 | Задача комівояжера (змішана мережа) + Бджолиний алгоритм |
| 14 | Розфарбовування графа + Генетичний алгоритм |
| 15 | Розфарбовування графа + Бджолиний алгоритм |
| 16 | Задача про кліку (задача розпізнавання) + Генетичний алгоритм |
| 17 | Задача про кліку (задача розпізнавання) + Бджолиний алгоритм |
| 18 | Задача про кліку (обчислювальна задача) + Генетичний алгоритм |
| 19 | Задача про кліку (обчислювальна задача) + Бджолиний алгоритм |
| 20 | Задача про найкоротший шлях + Генетичний алгоритм |
| 21 | Задача про найкоротший шлях + Мурашиний алгоритм |
| 22 | Задача про найкоротший шлях + Бджолиний алгоритм |
| 23 | Задача про рюкзак + Генетичний алгоритм |
| 24 | Задача про рюкзак + Бджолиний алгоритм |
| 25 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 26 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 27 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 28 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 29 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 30 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |

# Виконання

## Покроковий алгоритм

Здійснити 100 разів:

Перше: Генерація ділянок для пошуку

Друге: Виявлення найперспективніших ділянок

Третє: Відправлення фуражирів на вибрані ділянки

Четверте: Виявлення найкращого розв’язку серед ділянок

## Програмна реалізація алгоритму

### Вихідний код

import random  
  
NUMBER\_OF\_NODES = 100  
NUMBER\_OF\_WORKING\_BEES = 55  
NUMBER\_OF\_RECONNAISSANCE\_BEES = 5  
  
class Solution:  
  
 def \_\_init\_\_(self):  
 self.path\_length = 0  
 self.solution = []  
  
 def generate\_rnd\_solution(self, number\_of\_nodes: int):  
 for i in range(0, number\_of\_nodes):  
 self.solution.append(i)  
 for i in range(0, len(self.solution)):  
 j = random.randint(i, len(self.solution)-1)  
 k = self.solution[i]  
 self.solution[i] = self.solution[j]  
 self.solution[j] = k  
  
 def generate\_neighboring\_solution(self):  
 neighbor\_solution = Solution()  
 neighbor\_solution.solution = self.solution  
 changed\_index = random.randint(0, len(self.solution)-1)  
 if changed\_index == len(self.solution)-1:  
 neighbor\_index = 0  
 else:  
 neighbor\_index = changed\_index + 1  
 k = neighbor\_solution.solution[changed\_index]  
 neighbor\_solution.solution[changed\_index] = neighbor\_solution.solution[neighbor\_index]  
 neighbor\_solution.solution[neighbor\_index] = k  
 return neighbor\_solution  
  
 @staticmethod  
 def calculate\_weight(slt, mtrx):  
 weight = 0  
 for i in range(0, len(slt.solution)-1):  
 weight += mtrx[slt.solution[i]][slt.solution[i+1]]  
 weight += mtrx[slt.solution[len(slt.solution)-1]][slt.solution[0]]  
 slt.path\_length = weight  
 return weight  
  
 def greedy\_algorithm(self, mtrx, number\_of\_nodes):  
 visited = []  
 start = random.randint(0, number\_of\_nodes-1)  
 self.solution.append(start)  
 visited.append(start)  
 for i in range(0, number\_of\_nodes-1):  
 current\_node = self.solution[len(self.solution)-1]  
 local\_best = 200000000  
 local\_best\_index = 200000000  
 for j in range(0, number\_of\_nodes):  
 if mtrx[current\_node][j] == 0:  
 continue  
 if mtrx[current\_node][j] < local\_best and mtrx[current\_node][j] not in visited:  
 local\_best = mtrx[current\_node][j]  
 local\_best\_index = j  
 self.solution.append(local\_best\_index)  
 visited.append(local\_best\_index)  
 Solution.calculate\_weight(self, mtrx)  
  
  
class Algorithm:  
 def \_\_init\_\_(self, bee\_number, scout\_number, forager\_number, solutions\_number, iterations\_number, mtrx, is\_greedy):  
 self.bee\_number = bee\_number  
 self.scout\_number = scout\_number  
 self.forager\_number = forager\_number  
 self.solutions\_number = solutions\_number  
 self.iterations\_number = iterations\_number  
 self.nodes\_number = len(mtrx[0])  
 self.mtrx = mtrx  
 self.is\_greedy = is\_greedy  
 self.best\_solution = None  
 self.solutions = []  
 self.generate\_solutions()  
 self.sort\_solutions()  
  
 def generate\_solutions(self):  
 if self.is\_greedy:  
 for i in range(0, self.solutions\_number):  
 self.solutions.append(Solution())  
 self.solutions[i].greedy\_algorithm(self.mtrx, self.nodes\_number)  
 Solution.calculate\_weight(self.solutions[i], self.mtrx)  
 else:  
 for i in range(0, self.solutions\_number):  
 self.solutions.append(Solution())  
 self.solutions[i].generate\_rnd\_solution(self.nodes\_number)  
 Solution.calculate\_weight(self.solutions[i], self.mtrx)  
  
 def sort\_solutions(self):  
 for i in range(0, len(self.solutions)):  
 for j in range(0, len(self.solutions)):  
 if self.solutions[i].path\_length > self.solutions[j].path\_length:  
 self.solutions[i], self.solutions[j] = self.solutions[j], self.solutions[i]  
  
 def send\_scout(self, current\_best\_index, visited\_solutions):  
 random\_indicator = random.randint(0, 1)  
 if not random\_indicator:  
 random\_index = random.randint(0, len(self.solutions)-1)  
 if self.solutions[random\_index] in visited\_solutions:  
 return False  
 visited\_solutions.append(self.solutions[random\_index])  
 self.send\_foragers(random\_index)  
 else:  
 if current\_best\_index in visited\_solutions:  
 current\_best\_index += 1  
 return False  
 visited\_solutions.append(current\_best\_index)  
 self.send\_foragers(current\_best\_index)  
 current\_best\_index += 1  
 return True  
  
 def send\_foragers(self, index):  
 best\_neighbor = Solution()  
 best\_neighbor\_path\_length = 2000000  
 for i in range(0, self.forager\_number):  
 current\_neighbor = self.solutions[index].generate\_neighboring\_solution()  
 Solution.calculate\_weight(current\_neighbor, self.mtrx)  
 if current\_neighbor.path\_length < best\_neighbor\_path\_length:  
 best\_neighbor = current\_neighbor  
 best\_neighbor\_path\_length = current\_neighbor.path\_length  
 if best\_neighbor.path\_length < self.solutions[index].path\_length:  
 self.solutions[index] = best\_neighbor  
  
 def launch\_algorithm(self):  
 for i in range(0, self.iterations\_number):  
 visited\_solutions = []  
 current\_best\_index = 0  
 number\_of\_scouts\_sent = 0  
 while number\_of\_scouts\_sent != self.scout\_number and number\_of\_scouts\_sent != self.solutions\_number:  
 if self.send\_scout(current\_best\_index, visited\_solutions):  
 number\_of\_scouts\_sent += 1  
 self.sort\_solutions()  
 if (i+1) % 10 == 0 or i == self.iterations\_number:  
 print(f'Best solution found on iteration # {i+1} with path length {self.solutions[0].path\_length}')  
 return self.solutions[0]  
  
  
def generate\_graph():  
 graph = [[0 for \_ in range(0, NUMBER\_OF\_NODES)] for \_ in range(0, NUMBER\_OF\_NODES)]  
 for i in range(0, NUMBER\_OF\_NODES):  
 for j in range(0, NUMBER\_OF\_NODES):  
 distance = random.randint(5, 150)  
 if i != j:  
 graph[i][j] = distance  
 graph[j][i] = distance  
 else:  
 graph[i][j] = 0  
 return graph  
  
  
def print\_matrix(matrix):  
 for i in range(0, NUMBER\_OF\_NODES):  
 string = ""  
 for j in range(0, NUMBER\_OF\_NODES):  
 string += str(matrix[i][j]) + " "  
 print(string)  
 return 0  
  
  
def print\_hi(name):  
 # Use a breakpoint in the code line below to debug your script.  
 print(f'Hi, {name}') # Press ⌘F8 to toggle the breakpoint.  
  
  
# Press the green button in the gutter to run the script.  
if \_\_name\_\_ == '\_\_main\_\_':  
 matrix = generate\_graph()  
 print\_matrix(matrix)  
 bee\_colony\_algorithm = Algorithm(NUMBER\_OF\_WORKING\_BEES + NUMBER\_OF\_RECONNAISSANCE\_BEES, NUMBER\_OF\_RECONNAISSANCE\_BEES, NUMBER\_OF\_WORKING\_BEES, 50, 1000, matrix, False)  
 best\_solution = bee\_colony\_algorithm.launch\_algorithm()  
 print(best\_solution.solution)

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

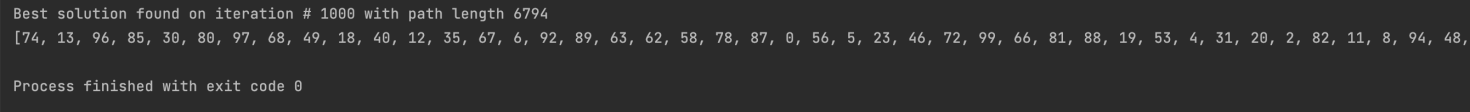


Рисунок 3.1 – Приклад роботи програми

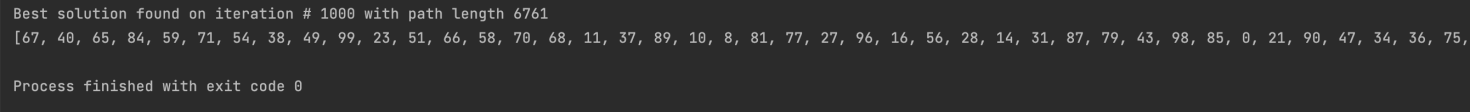


Рисунок 3.2 – Приклад роботи програми

## Тестування алгоритму

Для знаходження оптимальної кількості фуражирів проведемо випробування зафіксувавши число розвідників.

На рисунку 3.3 зображено залежність рішення задачі комівояжера(мінімальна довжина між усіма вузлами) від кількості фуражирів.

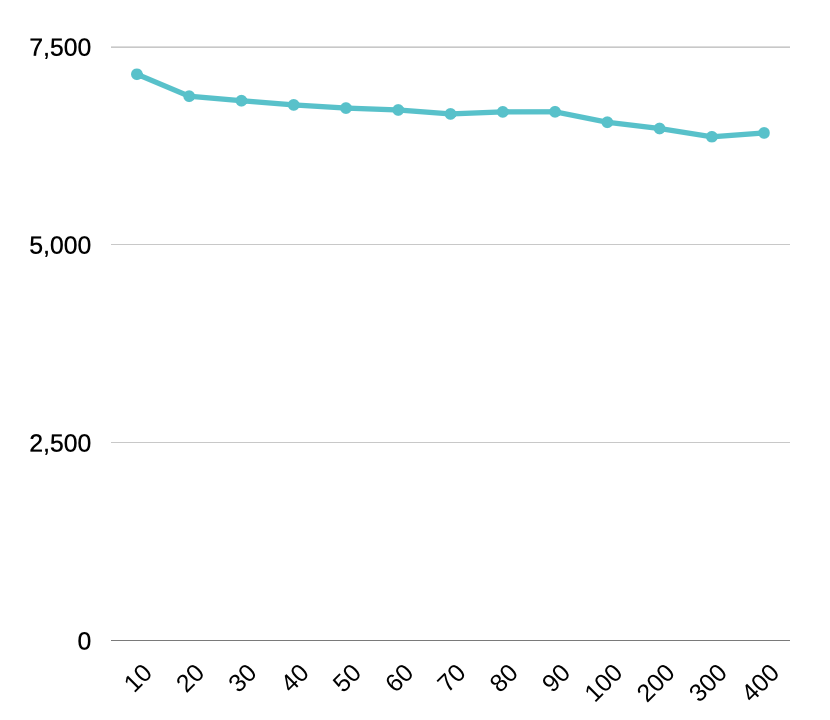


Рисунок 3.3 – Графік залежності рішення задачі від фуражирів

Бачимо, що найоптимальнішою кількістю фуражирів є 300, тому зафіксовуємо кількість фуражирів на цьому значені і змінюємо кількість розвідників.

На рисунку 3.4 зображено залежність кількості розвідників від розв’язку.

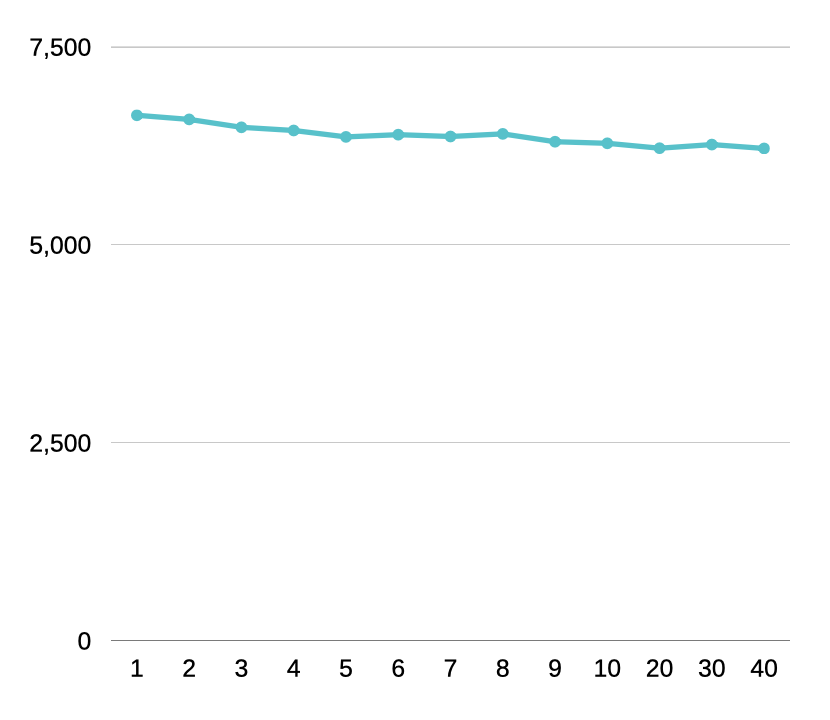


Рисунок 3.4 – Графік залежності рішення задачі від розвідників.

Бачимо, що найоптимальніший розв’язок знайдений при 40 розвідниках, проте час рішення в ~3 рази перевищує час рішення при 20 розвідниках та результати майже однакові. Через це найоптимальнішим є вибір 20 розвідників.

Висновок

В рамках даної лабораторної роботи було розроблено бджолиний алгоритм для рішення задачі комівояжера.

Було проведено тестування алгоритму. Протягом тестування було з’ясовано, що оптимальною кількістю фуражирів для рішення задачі є 300, а розвідників 20.

Критерії оцінювання

При здачі лабораторної роботи до 11.12.2022 включно максимальний бал дорівнює – 5. Після 11.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* покроковий алгоритм – 15%;
* програмна реалізація алгоритму – 50%;
* тестування алгоритму– 30%;
* висновок – 5%.