**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-Кондрацька Соня*

**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

## Програмна реалізація алгоритму

### Вихідний код

class BeeColony:  
 def \_\_init\_\_(self, node\_set):  
 self.role = ''  
 self.path = list(node\_set) # зберігає всі вузли в кожній бджолі, рандомізує фуражирів  
 self.distance = 0  
 self.cycle = 0  
  
def input\_start\_values():  
 top\_count = input('Enter the number of plots(min:10 max:500): ')  
 while not top\_count.isdigit() or (int(top\_count) > 500 or int(top\_count) < 10):  
 top\_count = input('Enter the number of plots(min:10 max:500): ')  
 top\_count = int(top\_count)  
 population = input('Enter the number of bees(min:10 max:1000): ')  
 while not population.isdigit() or (int(population) > 1000 or int(population) < 10):  
 population = input('Enter the number of bees(min:10 max:1000): ')  
 population = int(population)  
  
 cycle\_limit = 100  
  
 forager\_percent = 0.5  
 onlooker\_percent = 0.5  
 scout\_percent = 0.2  
 forager\_limit = 1000  
 role\_percent = [onlooker\_percent, forager\_percent]  
 scout\_count = math.ceil(population \* scout\_percent)  
 data = [[row, random.randint(5, 150), random.randint(5, 150)] for row in range(top\_count)]#енерація відстаней(асиметрична мережа)  
  
 return top\_count,population,role\_percent,scout\_count,forager\_limit,cycle\_limit,data  
def get\_distance\_between\_nodes(n1, n2):  
 *"""*  
 *Обчислює евклідову відстань між двома вузлами.*  
 *"""*  
return distance.euclidean(n1, n2)  
  
  
def make\_distance\_table(data\_list):  
 *"""*  
 *Створює таблицю, яка зберігає відстань між кожною парою вузлів.*  
 *"""*  
length = len(data\_list)  
 table = [[get\_distance\_between\_nodes(  
 (data\_list[i][1],data\_list[i][2]), (data\_list[j][1],data\_list[j][2]))  
 for i in range(0, length)] for j in range(0, length)]  
 return table  
  
  
def get\_total\_distance\_of\_path(path, table):  
 *"""*  
 *Обчислює загальну відстань шляху окремої бджоли.*  
 *Завершується на початковому вузлі для завершення циклу.*  
 *"""*  
new\_path = list(path)  
 new\_path.insert(len(path), path[0])  
 new\_path = new\_path[1:len(new\_path)]  
  
 coordinates = zip(path, new\_path)  
 distance = sum([table[i[0]][i[1]] for i in coordinates])  
 return round(distance, 3)  
  
  
def initialize\_hive(population, data):  
 *"""*  
 *Ініціалізує вулик і заселяє його бджолами.*  
 *Бджоли матимуть випадковий атрибут шляху.*  
 *"""*  
path = [x[0] for x in data]  
 hive = [BeeColony(path) for i in range (0, population)]  
 return hive  
  
  
def assign\_roles(hive, role\_percentiles, table):  
 *"""*  
 *Призначає початкові ролі на основі процентилів ролей*  
 *кожній бджолі у вулику.*  
 *Призначає рандомізований шлях бджолам-фуражирам.*  
 *"""*  
population = len(hive)  
 onlooker\_count = math.floor(population \* role\_percentiles[0])  
 forager\_count = math.floor(population \* role\_percentiles[1])  
  
 for i in range(0, onlooker\_count):  
 hive[i].role = 'O'  
  
 for i in range(onlooker\_count, (onlooker\_count + forager\_count)):  
 hive[i].role = 'F'  
 random.shuffle(hive[i].path)  
 hive[i].distance = get\_total\_distance\_of\_path(hive[i].path, table)  
  
 return hive  
  
def mutate\_path(path):  
 *"""*  
 *Отримує випадковий індекс від 0 до останнього елемента.*  
 *Копіює шлях, міняє місцями два вузли, порівнює відстань.*  
 *Повертає мутований шлях.*  
 *"""*  
random\_idx = random.randint(0, len(path) - 2)  
 new\_path = list(path)  
 new\_path[random\_idx], new\_path[random\_idx + 1] = new\_path[random\_idx + 1], new\_path[random\_idx]  
 return new\_path  
  
def forage(bee, table, limit):  
 *"""*  
 *Поведінка робочої бджоли, ітеративно уточнює потенційний найкоротший шлях*  
 *шляхом заміни випадково вибраних сусідніх індексів.*  
 *"""*  
new\_path = mutate\_path(bee.path)  
 new\_distance = get\_total\_distance\_of\_path(new\_path, table)  
  
 if new\_distance < bee.distance:  
 bee.path = new\_path  
 bee.distance = new\_distance  
 bee.cycle = 0 # скидаємо цикл, щоб бджола могла продовжувати прогресувати  
 else:  
 bee.cycle += 1  
 if bee.cycle >= limit: # якщо бджола не прогресує  
 bee.role = 'S'  
 return bee.distance, list(bee.path)  
  
  
def scout(bee, table):  
 *"""*  
 *Поведінка бджоли-розвідника, відмовляється від невдалого шляху на новий випадковий шлях.*  
 *Скидає роль до фуражиру.*  
 *"""*  
new\_path = list(bee.path)  
 random.shuffle(new\_path)  
 bee.path = new\_path  
 bee.distance = get\_total\_distance\_of\_path(new\_path, table)  
 bee.role = 'F'  
 bee.cycle = 0  
  
  
def waggle(hive, best\_distance, table, forager\_limit, scout\_count):  
 *"""*  
 *Фіксує результати роботи бджіл-фуражирів,*  
 *вибирає новий випадковий шлях для дослідження розвідників,*  
 *повертає результати для оцінки сторонніми особами.*  
 *"""*  
best\_path = []  
 results = []  
  
 for i in range(0, len(hive)):  
 if hive[i].role == 'F':  
 distance, path = forage(hive[i], table, forager\_limit)  
 if distance < best\_distance:  
 best\_distance = distance  
 best\_path = list(hive[i].path)  
 results.append((i, distance))  
  
 elif hive[i].role == 'S':  
 scout(hive[i], table)  
  
 # after processing all bees, set worst performers to scout  
 results.sort(reverse = True, key=lambda tup: tup[1])  
 scouts = [ tup[0] for tup in results[0:int(scout\_count)] ]  
 for new\_scout in scouts:  
 hive[new\_scout].role = 'S'  
 return best\_distance, best\_path  
  
  
def recruit(hive, best\_distance, best\_path, table):  
 *"""*  
 *Набирає бджіл-спостерігачів, щоб знайти найкраще рішення.*  
 *Повертає оновлені best\_distance, best\_path.*  
 *"""*  
for i in range(0, len(hive)):  
 if hive[i].role == 'O':  
 new\_path = mutate\_path(best\_path)  
 new\_distance = get\_total\_distance\_of\_path(new\_path, table)  
 if new\_distance < best\_distance:  
 best\_distance = new\_distance  
 best\_path = new\_path  
 return best\_distance, best\_path  
  
  
def print\_details(cycle, path, distance, bee):  
 print("Info about the shortest path: ")  
 print("Cycle №: {}".format(cycle))  
 print("Path: {}".format(path))  
 print("Distance: {}".format(distance))  
 print("Bee: {}".format(bee))  
 print("\n")

main()

import sys  
from BeeColony import \*  
  
top\_count,population,role\_percent,scout\_count,forager\_limit,cycle\_limit,data = input\_start\_values()  
  
table = make\_distance\_table(data)  
hive = initialize\_hive(population, data)  
assign\_roles(hive, role\_percent, table)  
  
cycle = 1  
best\_distance = sys.maxsize  
best\_path = []  
result = ()  
while cycle < cycle\_limit:  
 waggle\_distance, waggle\_path = waggle(hive, best\_distance, table, forager\_limit, scout\_count)  
 if waggle\_distance < best\_distance:  
 best\_distance = waggle\_distance  
 best\_path = list(waggle\_path)  
 #print\_details(cycle, best\_path, best\_distance,'F')  
 result = (cycle, best\_path, best\_distance,'F')  
  
 recruit\_distance, recruit\_path = recruit(hive, best\_distance, best\_path, table)  
 if recruit\_distance < best\_distance:  
 best\_distance = recruit\_distance  
 best\_path = list(recruit\_path)  
 #print\_details(cycle, best\_path, best\_distance,'R')  
 result = (cycle, best\_path, best\_distance,'R')  
 cycle += 1  
  
print\_details(result[0],result[1],result[2],result[3])

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

Рисунок 3.1 –

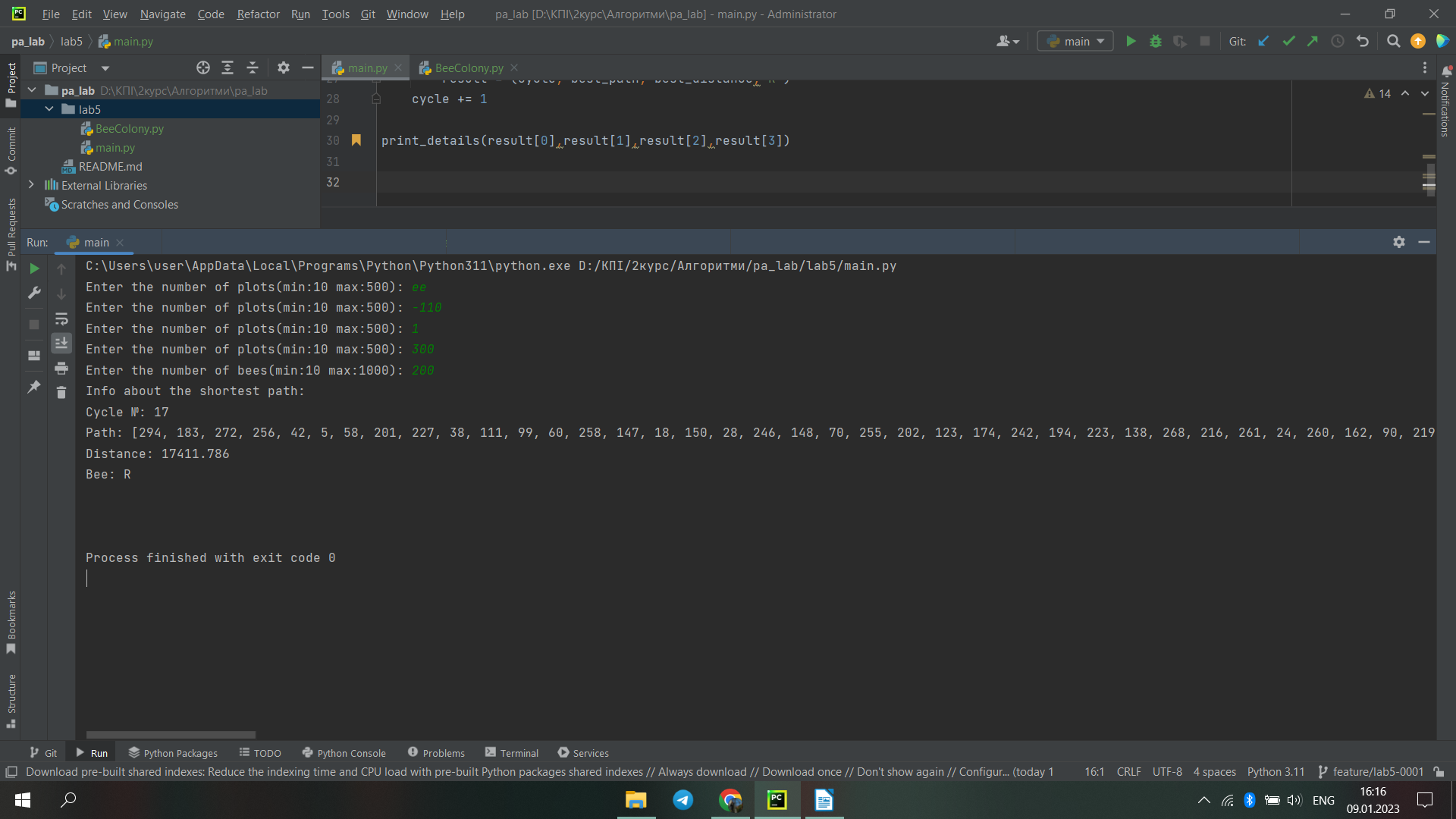
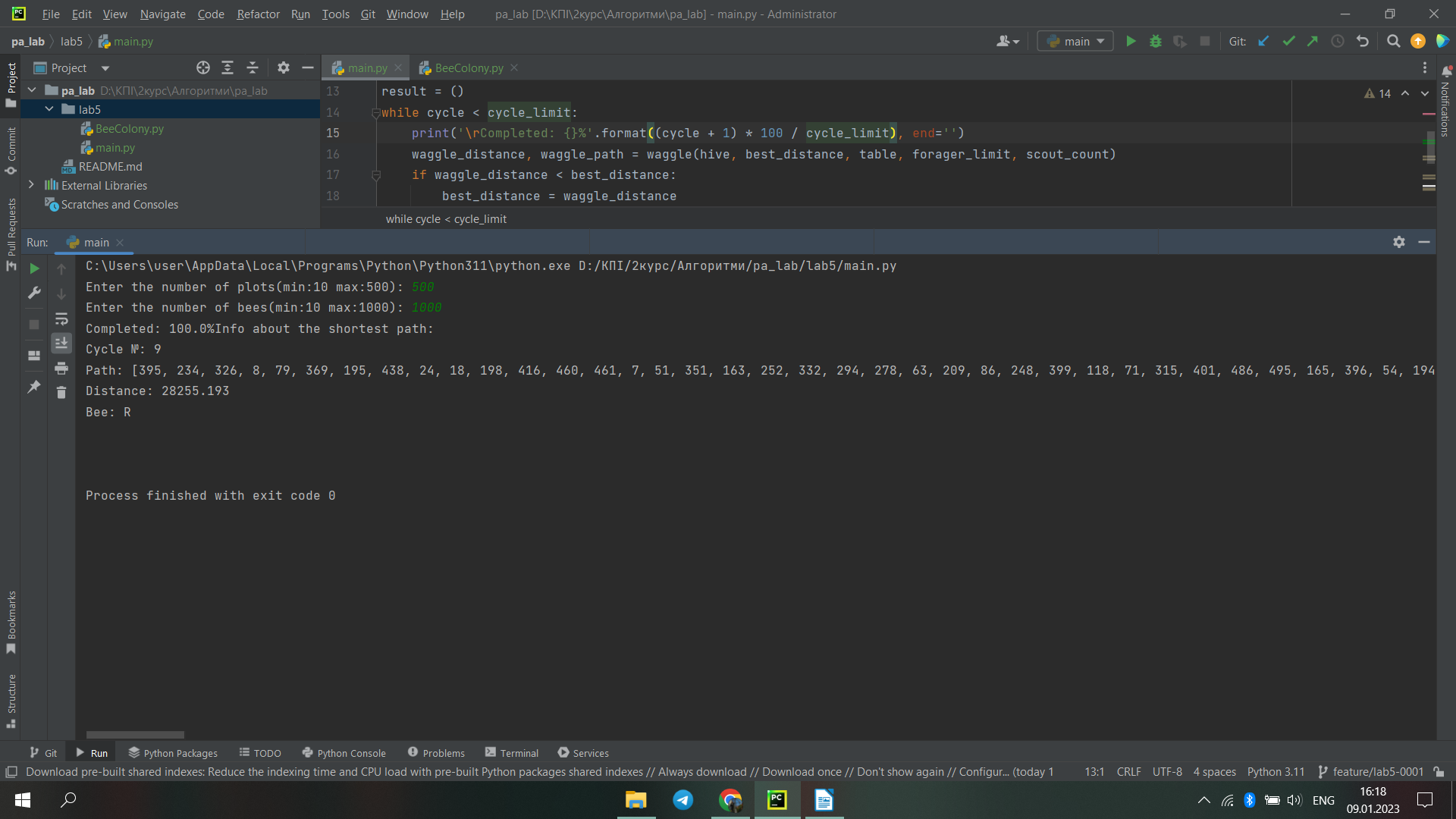


Рисунок 3.2 –



## Тестування алгоритму

Кількість фуражирів: 10

Кількість ділянок:12

|  |  |
| --- | --- |
| Кількість розвідників | Найменша відстань |
| 1 | 340.316 |
| 2 | 380.225 |
| 3 | 360.24 |
| 4 | 406.296 |
| 5 | 413.131 |
| 6 | 333 |
| 7 | 394.755 |
| 8 | 340 |
| 9 | 340 |
| 10 | 420 |

Рисунок 3.1 — Графік залежності цільової функції від кількості розвідників (Кількість фуражирів: 10)

Кількість фуражирів: 100

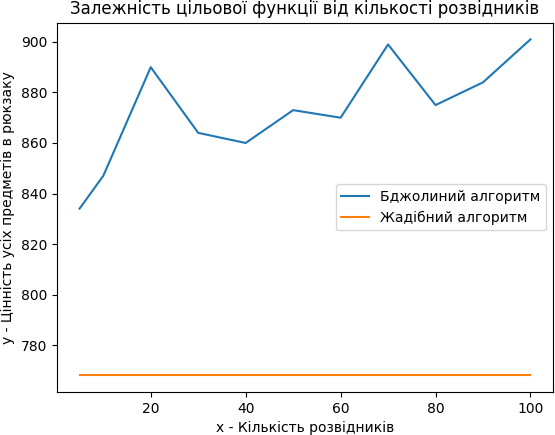
|  |  |
| --- | --- |
| Кількість розвідників | Цінність усіх предметів в рюкзаку |
| 10 | 333 |
| 20 | 347 |
| 30 | 356 |
| 40 | 362 |
| 50 | 333 |
| 60 | 345 |

в

## Рисунок 3.2 — Графік залежності цільової функції від кількості розвідників (Кількість фуражирів: 100)

Кількість фуражирів: 100

|  |  |
| --- | --- |
| Кількість розвідників | Цінність усіх предметів в рюкзаку |
| 5 | 834 |
| 10 | 847 |
| 20 | 890 |
| 30 | 864 |
| 40 | 860 |
| 50 | 873 |
| 60 | 870 |
| 70 | 899 |
| 80 | 875 |
| 90 | 884 |
| 100 | 901 |



## Рисунок 3.3 — Графік залежності цільової функції від кількості розвідників (Кількість фуражирів: 100)

Висновок

В рамках даної лабораторної роботи я вивчила основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач. Опрацювала методологію підбору прийнятних параметрів алгоритму, формалізувала алгоритм вирішення задачі відповідно загальної методології.

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%.