**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 Орищенко Я.О.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

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

Створити порожній список найкращих клік

Для заданої кількості ітерацій:

Поки не оглянуто всі вершини:

Відправити розвідників на ділянки (вершини графу)

Обчислити нектар

Для кожного розвідника:

Поділитись результатами пошуку із спостерігачами

Обрати ділянку

Надіслати на ділянку спосерігачів

Для кожної сусідньої вершини ділянки:

Перевірити, чи входить сусід в кліку

Повернути обчислену кліку

Додати найбільшу кліку в список

Повернути список найбільших клік

## Програмна реалізація алгоритму

### Вихідний код

**main.py**

import random

import time

from bee\_algorithm import BeeColony

def create\_graph(size, min\_max):

min, max = min\_max

graph = {key: set() for key in range(size)}

for key in graph:

r = random.randint(min, max) - len(graph[key])

for i in range(r):

elem = random.randint(0, size - 1)

if elem != key:

graph[key].add(elem)

for k in graph[key]:

graph[k].add(key)

return graph

def main():

graph = create\_graph(300, (2, 30))

scouts = 5

onlookers = 40

iterations = 2

bc = BeeColony(graph, scouts, onlookers, iterations)

start = time.time()

cliques = bc.abc\_algorithm()

end = time.time()

print(f"best results of {iterations} iterations:")

for clique in cliques:

print(clique)

print("time:", end - start)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**bee\_algorithm.py**

import random

class BeeColony:

def \_\_init\_\_(self, graph: dict, scouts\_amount: int, onlookers\_amount: int, iterations: int):

self.graph = graph

self.not\_visited = set(self.graph.keys())

self.not\_checked = set(self.graph.keys())

self.scouts\_amount = scouts\_amount

self.scouts = [None for \_ in range(self.scouts\_amount)]

self.onlookers\_amount = onlookers\_amount

self.iterations = iterations

self.cliques = []

def abc\_algorithm(self) -> list:

max\_cliques = []

for i in range(self.iterations):

while self.not\_checked:

self.scouts = [None for \_ in range(self.scouts\_amount)]

self.\_send\_scouts()

for scout in self.scouts:

vertex = self.\_choose\_by\_nectar(self.\_fitness())

self.\_send\_onlookers(vertex)

max\_cliques.append(sorted(self.cliques, key=lambda clique: len(clique), reverse=True)[0])

self.\_refresh()

return max\_cliques

def \_send\_scouts(self):

for i, scout in enumerate(self.scouts):

if self.not\_visited:

vertex = random.choice(list(self.not\_visited))

self.scouts[i] = vertex

else:

break

def \_send\_onlookers(self, vertex: int):

clique = self.\_find\_clique(vertex)

if clique not in self.cliques and len(clique) > 2:

self.cliques.append(clique)

def \_fitness(self) -> list:

nectar\_list = []

for scout in self.scouts:

neighbors\_amount = len(self.graph[scout])

nectar\_list += [scout for \_ in range(neighbors\_amount)]

return nectar\_list

def \_choose\_by\_nectar(self, nectar\_list: list) -> int:

vertex = random.choice(nectar\_list)

i = self.scouts.index(vertex)

self.scouts.pop(i)

return vertex

def \_find\_clique(self, vertex: int) -> list:

available\_bees = self.onlookers\_amount

if vertex in self.not\_checked:

self.not\_checked.remove(vertex)

self.not\_visited.remove(vertex)

clique = [vertex]

for neighbor in self.graph[vertex]:

if available\_bees <= 0:

break

available\_bees -= 1

if neighbor in self.not\_checked:

self.not\_checked.remove(neighbor)

in\_clique = True

for clique\_vertex in clique:

if neighbor not in self.graph[clique\_vertex]:

in\_clique = False

break

if in\_clique:

clique.append(neighbor)

return clique

def \_refresh(self):

self.not\_visited = set(self.graph.keys())

self.not\_checked = set(self.graph.keys())

self.scouts = [None for \_ in range(self.scouts\_amount)]

self.cliques = []

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

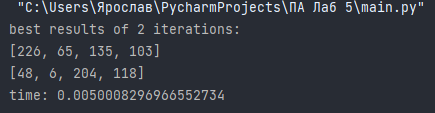


Рисунок 3.1 – результат роботи програми для 2 ітерацій

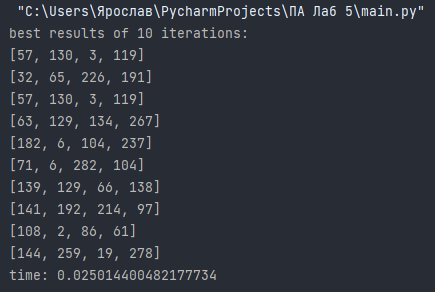


Рисунок 3.2 – результат роботи програми для 10 ітерацій

## Тестування алгоритму

Протестуємо, при якій кількості розвідників та фуражирів алгоритм працюватиме найбільш ефективно.

Для одного і того самого графу, поступово збільшуватимемо кількість спостерігачів, від 1 до 100 із кроком 5, і перевіримо, при якій кількості алгоритм працюватиме найшвидше при 10 ітераціях, та видаватиме найкращий результат.

Як можна побачити, в проміжку від 1 до 40 час кількість розв’язків стрімко збільшується, а час виконання стрімко зменшується. Після 40, результати залишаються приблизно рівними, відрізняючись в межах деякої похибки. Зафіксуємо оптимальну кількість спостерігачів – 40.

Далі протестуємо залежність часу виконання 10 ітерацій алгоритму від кількості розвідників, поступово збільшуючи кількість розвідників від 1 до 50.

Можна побачити лінійну залежність кількості часу від кількості розвідників. Оберемо опитмальну кількість розвідників 5, таким чином алгоритм не буде жадібним, і виконуватиметься за оптимальний час.

Висновок

В рамках даної лабораторної роботи було вивчено основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач. Було розроблено рішення обчислювальної задачі про Кліку, що базується на ABC варіації алгоритму бджолиної колонії. Було досліджено та підібрано оптимальні параметри алгоритму, такі як кількість розвідників та кількість спостерігачів для заданого графу.

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%.