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) *ІП-12 Сімчук Андрій Володимирович* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олексій Олександрович* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 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.

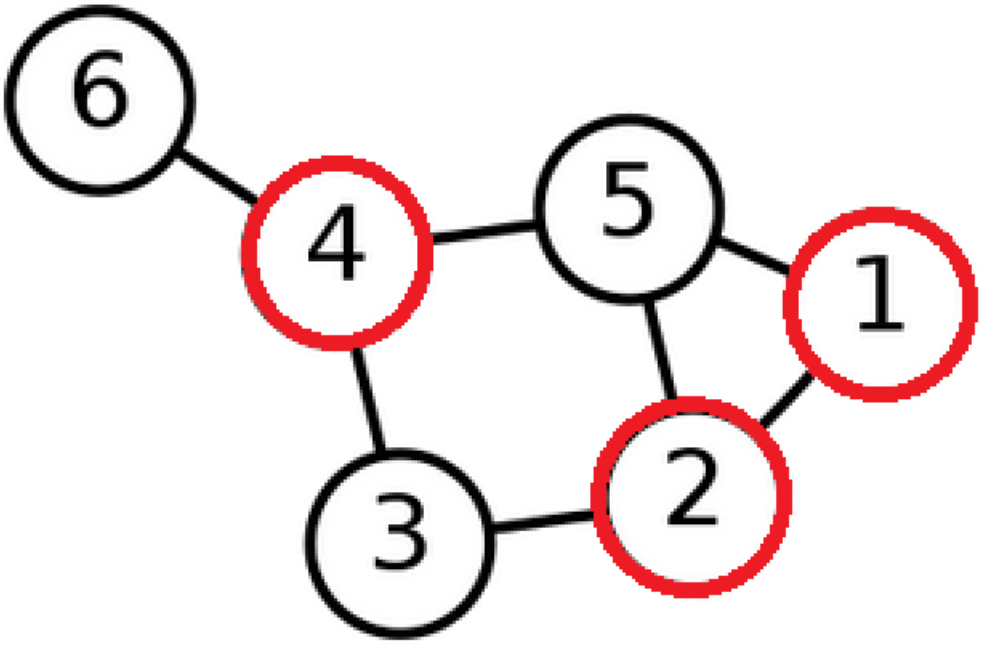
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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 покриття полягає 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 M 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

3 ВИКОНАННЯ

3.1 Покроковий алгоритм

```
function find_shortest_path (start, destination) do
      for (i in range (n_iterations)) do
            all_paths = gen_all_paths (start, destination)

            spread_pheromone (all_paths, n_best)

            shortest_path = min (all_paths, their_distances)

            if (shortest_path < all_time_shortest_path) do
                  all_time_shortest_path = shortest_path
            end if

            pheromone = pheromone * decay
      end for
      return all_time_shortest_path
end function


function gen_all_paths (start, destination) do
      all_paths = empty list
      for (i in range (n_ants)) do
            path = gen_path (start, destination)
            add (path, gen_path_dist (path)) to all_paths
      end for
  return all_paths
end function


function gen_path (start, destination) do
      path = empty list
      visited = list
      add start to visited
      prev = start
      move = None

      for (i in range (length of distances - 1)) do
            if (move != destination) do
```

```
                move = pick_move (pheromone of prev, distances of
                prev, visited)
                add (prev, move) to path
                prev = move
                add move to visited
            end if
            else do
                return path
            end else
        end for
        return path
end function


function gen_path_dist (path) do
        total_dist = 0
        for (ele in path) do
            total_dist += distances of ele
        end for
        return total_dist
end function


function pick_move (pheromone_, dist) do
        pheromone = copy of pheromone_
        row = pheromone ** alpha * ((1.0 / dist) ** beta)
        norm_row = row / sum of row
        move = choice random move with probability norm_row
        return move
end function


function spread_pheromone (all_paths, n_best) do
        sorted_paths = sorted(all_paths)
        for (path, dist in sorted_paths[:n_best]) do
            k = 0
            for (move in path) do
                k += 1
                pheromone of move += 1.0 / distances of move
            end for
end function
```

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

Файл «main.py»

```
from algorithm import AntColonyAlgorithm
import random as rn
import numpy as np


def create_random_graph(n, min_pow, max_pow, min_dist, max_dist):
    arr = []
    for i in range(n):
        a = []
        for j in range(n):
            if i == j:
                a.append(np.inf)
            else:
                a.append(10000000000000000000)
        power = rn.randint(min_pow, max_pow)
        k = 0
        while k != power:
            a[rn.randint(1, n - 1)] = rn.randint(min_dist, max_dist)
            k += 1
        arr.append(a)
    dist = np.array(arr)
    return dist


def main():
    n_ants = 20
    n_best = 5
    n_iterations = 50
    decay = 0.95
    alpha = 1
    beta = 1
    start = 5
    destination = 155
    min_pow = 1
    max_pow = 10
    min_dist = 5
    max_dist = 150
```

```
        distances = create_random_graph(300, min_pow, max_pow, min_dist,
max_dist)

        aco = AntColonyAlgorithm(distances, n_ants, n_best, n_iterations,
decay, alpha, beta)
        aco.find_shortest_path(start, destination)


    if __name__ == "__main__":
        main()
```

Файл «algorithm.py»

```
    import numpy as np
    from numpy.random import choice as np_choice


    class AntColonyAlgorithm:
        def __init__(self, distances, n_ants, n_best, n_iterations, decay,
alpha=1, beta=1):
            self.distances = distances
            self.n_ants = n_ants
            self.n_best = n_best
            self.n_iterations = n_iterations
            self.decay = decay
            self.alpha = alpha
            self.beta = beta
            self.pheromone = np.ones(distances.shape) / 10
            self.all_inds = range(len(distances))
            self.shortest_path = None
            self.all_time_shortest_path = ("placeholder", np.inf)

        def pick_move(self, pheromone_, dist, visited):
            pheromone = np.copy(pheromone_)
            # Make zero if the path has been visited
            # (can be used to make algorithm faster, but can cause not finding
path to destination)
            # pheromone[list(visited)] = 0

            # Ant makes a decision on what node to go using this formula
            row = pheromone ** self.alpha * ((1.0 / dist) ** self.beta)
            # Probability formula
            norm_row = row / row.sum()
```

```
            #  Move  randomly  using  probability  (select  path  to  go  using
probability)
            # p = probability
            # Get index of an element that has bigger probability
            move = np_choice(self.all_inds, 1, p=norm_row)[0]

            # Return path that randomly selected
            return move

        def spread_pheromone(self, all_paths, n_best):
            # sorted a path form small to big (with values to be shorted are
values on second column)
            sorted_paths = sorted(all_paths, key=lambda x: x[1])
            for path, dist in sorted_paths[:n_best]:
                k = 0
                for move in path:
                    k += 1
                    # print(move)
                    # ant deposits a pheromone on the way that its travelled
                    #  the  amount  of  pheromone  that  the  ant  deposit  is  (1  /
distances between 2 cities)
                    self.pheromone[move] += 1.0 / self.distances[move]
                print()
                print("Amount of moves: " + str(k))
                print("Distance: {1}\n{0}".format(path, dist))

        def gen_path_dist(self, path):
            total_dist = 0
            for ele in path:
                total_dist += self.distances[ele]
            return total_dist

        def gen_path(self, start, destination):
            path = []
            visited = set()
            visited.add(start)
            prev = start
            move = None
            for i in range(len(self.distances) - 1):
                if move != destination:
                    move        =        self.pick_move(self.pheromone[prev],
self.distances[prev], visited)
```

```
                    # Append path
                    path.append((prev, move))
                    # Change previous path to move path after append path
                    prev = move
                    # add path that has been moved to visited, so the path
that has visited can be made to zero
                    visited.add(move)
                else:
                    return path
            return path

        def gen_all_paths(self, start, destination):
            all_paths = []
            for i in range(self.n_ants):
                path = self.gen_path(start, destination)
                all_paths.append((path, self.gen_path_dist(path)))
            return all_paths

        def find_shortest_path(self, start, destination):
            for i in range(self.n_iterations):
                # Get all paths
                all_paths = self.gen_all_paths(start, destination)

                self.spread_pheromone(all_paths, self.n_best)

                # Get the minimal value in array all_paths in column 2
                # example :
                # a = [(0, 8), (3, 7), (2, 6), (1, 9), (4, 5)]
                # min(a, key=lambda x: x[1])
                # the result is : (4, 5)
                # x[1] means second column
                # x[0] means first column
                # shortest path = (path)

                # Get the shortest path in all paths, based on its distance
(x[1])
                self.shortest_path = min(all_paths, key=lambda x: x[1])

                # print("shortest path : ## {0}".format(self.shortest_path))

                if self.shortest_path[1] < self.all_time_shortest_path[1]:
                    self.all_time_shortest_path = self.shortest_path
```

```
                    self.pheromone = self.pheromone * self.decay

                k = 0
                for move in self.all_time_shortest_path[0]:
                    k += 1

                # return the shortest path that founded by the ants
                print("\nShortest
path\n==========================================================================
==========")
                print("Amount of moves: " + str(k))
                print("Distance:  {1}\n{0}".format(self.all_time_shortest_path[0],
self.all_time_shortest_path[1]))
                return self.all_time_shortest_path
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

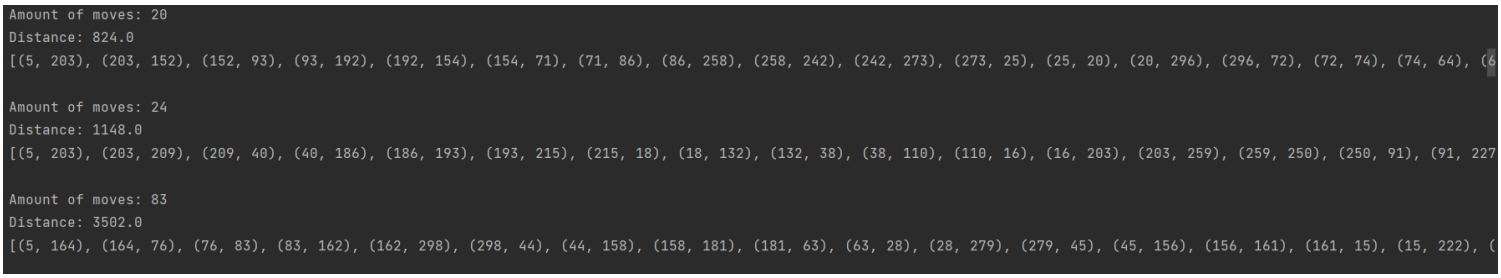


Рисунок 3.1 – Вивід програмою варіантів шляху з одної вершини в іншу

```
Shortest path
================================================================================
Amount of moves: 8
Distance: 200.0
[(5, 28), (28, 189), (189, 271), (271, 57), (57, 119), (119, 74), (74, 225), (225, 155)]

Process finished with exit code 0
```

Рисунок 3.2 – Вивід програмою найкоротшого шляху з однієї вершини в іншу

3.3 Тестування алгоритму

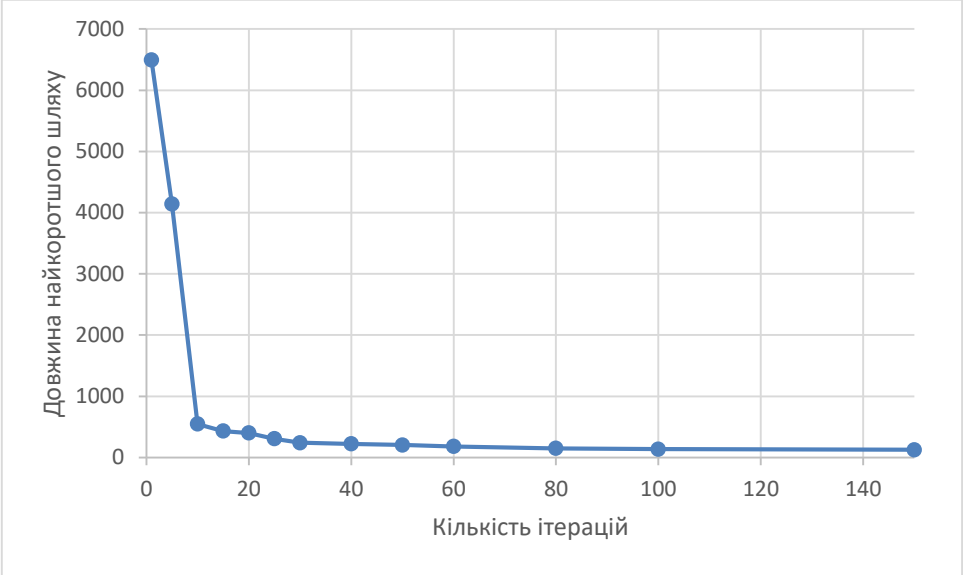


Рисунок 3.3 – Графік залежності довжини найкоротшого шляху між двома вершинами від кількості ітерацій (для однієї задачі)

Рисунок 3.3 – Графік залежності довжини найкоротшого шляху між двома вершинами від кількості ітерацій (для випадково згенерованих задач)

ВИСНОВОК

В рамках даної лабораторної роботи було вивчено основні підходи розробки метаевристичних алгоритмів для типових прикладних задач (Задача про найкоротший шлях + Мурашиний алгоритм). Було опрацьовано методологію підбору прийнятних параметрів алгоритму і визначено оптимальні параметри для заданої задачі:

- $\alpha = 1$;
- $\beta = 1$;
- $\rho = 0.95$;
- $L_{min} = 1$;
- кількість мурах $M = 20$;

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%.