**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-21 Сергієнко Сергій* _____
(шифр, прізвище, ім'я, по батькові)

**Перевірив** *Головченко М.М.* _____
(прізвище, ім'я, по батькові)

Київ 2023

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.

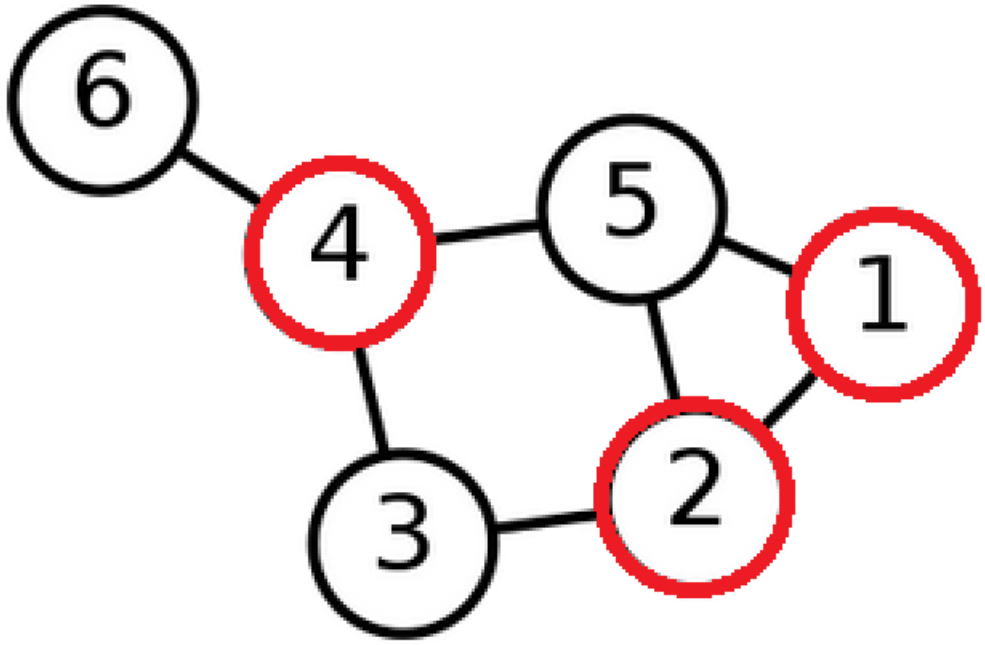
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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.<br>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.<br>**Розглядається симетричний, асиметричний та змішаний варіанти.**<br>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.<br>У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.<br>У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху.<br>Застосування:<br>– доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів);<br>– доставка води; |

| | |
|---|---|
| | – моніторинг об'єктів;<br>– поповнення банкоматів готівкою;<br>– 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.<br>Застосування:<br>– розкладу для освітніх установ;<br>– розкладу в спорті;<br>– планування зустрічей, зборів, інтерв'ю;<br>– розклади транспорту, в тому числі - авіатранспорту;<br>– 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.<br>Задача вершинного покриття полягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).<br>На вході: Граф G = (V, E).<br>Результат: множина C ⊆ V - найменше вершинне покриття графа G. |

| | |
|---|---|
| |  <br>Застосування:<br>– розміщення пунктів обслуговування;<br>– призначення екіпажів на транспорт;<br>– 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.<br>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).<br>Застосування:<br>– біоінформатика;<br>– 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.<br>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:**<br>- оператор схрещування (мінімум 3);<br>- мутація (мінімум 2);<br>- оператор локального покращення (мінімум 2). |
| 2 | **Мурашиний алгоритм**:<br>– α;<br>– β;<br>– ρ;<br>– Lmin;<br>– кількість мурах M і їх типи (елітні, тощо…);<br>– маршрути з однієї чи різних вершин. |
| 3 | **Бджолиний алгоритм:**<br>– кількість ділянок;<br>– 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

I. Генерація ділянок для пошуку нектару.
   a. Генерація одного початкового маршруту, який присвоюється усім ділянкам, кількість ділянок відповідає кількості робочих бджіл, одна робоча бджола прив’язана до однієї ділянки.

II. Оцінка корисності ділянок.
   a. Оцінка усіх початкових ділянок. Рахуємо довжину початкового шляху.

III. Відправка фуражирів на ділянки. Пошук в околицях джерел нектару.
   a. Спроба покращення шляхів локальним пошуком. Випадково вибирається проміжна вершина у початковому шляху, і намагаємося замінити наступні після неї вершини на інші.
   b. Якщо новий шлях коротше старого, міняємо старий на новий.
   c. Перевіряємо лічильник невдач покращень шляху спостерігачами. Якщо це значення перевищує граничне, фуражир перетворюється у розвідника і шукає нову ділянку, генеруючи новий шлях, що замінює старий.

IV. Оновлення оцінок корисності ділянок.

V. Відправка бджіл-спостерігачів.
   a. Фуражири повертаються у вулик і передають інформацію про кількість нектару на ділянках спостерігачам. Спостерігачі, користуючись пропорційною селекцією, обирають перспективні ділянки для дослідження.
   b. Спостерігачі вилітають на ділянки та намагаються покращити шляхи локальним пошуком.
   c. Якщо покращення вдале, міняємо старий шлях на новий. Якщо покращення невдале, додаємо 1 до лічильника невдач покращень шляху.

VI. Якщо умова зупинки не виконується, то п. II.

VII. Кінець роботи алгоритму.

## 3.2 Програмна реалізація алгоритму

### 3.2.1 Вихідний код

```
from bee_colony import *
from helpers import *
VERTEX_AMOUNT = 300
LENGTH_LO = 5
LENGTH_HI = 150
VERTEX_LO = 1
VERTEX_HI = 10
ITERATIONS_NUM = 30


def main():
    while True:
        onlookers = input("Amount of onlookers: ")
        if validate_positive_integer(onlookers):
            break
    onlookers = int(onlookers)
    while True:
        employed = input("Amount of employed: ")
        if validate_positive_integer(employed):
            break
    employed = int(employed)
    graph = Graph(VERTEX_AMOUNT, VERTEX_LO, VERTEX_HI, LENGTH_LO,
LENGTH_HI)
    ba = BeesAlgorithm(onlookers, employed, graph, 0, 299, ITERATIONS_NUM)
    ba.solve()
    best_path = ba.get_best_path()
    print("PATH (vertices):", '->'.join(map(str, best_path[0][0])))
    print(f"total length: {best_path[0][1]} | PATH (weights):",
'+'.join(map(str, best_path[1])))


if __name__ == "__main__":
    main()

def validate_positive_integer(n):
    try:
```

```
                value = int(n)
                if value > 0:
                    return True
                else:
                    return False
            except ValueError:
                return False

        from graph import *
        FAIL_SEARCH_COUNT = 10


        from graph import *
        FAIL_SEARCH_COUNT = 10


        class BeesAlgorithm:
            def __init__(self, onlookers, employed, graph, start_vertex,
end_vertex, iterations_num):
                self.onlookers_num = onlookers
                self.employed_num = employed
                self.graph = graph
                self.size = self.graph.size
                self.path_counters = [[None, 0] for _ in range(self.employed_num)]
# зберігається к-сть невдалих пошуків
                self.paths = [[[], 0] for _ in range(self.employed_num)]
                self.record_fitness = 0
                self.start_vertex = start_vertex
                self.end_vertex = end_vertex
                self.record_fitness = 0
                self.iterations_num = iterations_num
                self.records = []

            def calculate_fitness(self, path):
                vertices = path[0]
                path[1] = sum(self.graph.adj_matrix[vertices[i]][vertices[i + 1]]
for i in range(len(vertices) - 1))

            def local_search(self, start_vertex, end_vertex):
                path = [start_vertex]
                current_vertex = start_vertex
                steps_taken = 0
```

```
            while current_vertex != end_vertex:
                possible_next_vertices = [
                    [neighbor,
self.graph.adj_matrix[current_vertex][neighbor]]
                    for neighbor in range(self.graph.size)
                    if self.graph.adj_matrix[current_vertex][neighbor]
                ]

                current_vertex = roulette_wheel_choice(possible_next_vertices)
                path.append(current_vertex)
                steps_taken += 1

                if steps_taken > self.size - 1:  # шлях більше кількості вершин
                    return [-1]
            path = remove_duplicates(path)

            return path

        def solve(self):
            self.initialize_sources()
            print(f"i = 0 | Init path = {'->'.join(map(str, self.paths[0][0]))},
its length = {self.paths[0][1]}")

            for i in range(self.iterations_num):
                self.records.append(self.paths[self.record_fitness][1])
                self.send_employed()
                self.record_fitness = min(range(self.employed_num), key=lambda
x: self.paths[x][1])
                self.send_onlookers()
                print(f"i = {i+1} | Best path = {'->'.join(map(str,
self.paths[self.record_fitness][0]))}, its length =
{self.paths[self.record_fitness][1]}")

        def send_employed(self):
            for i in range(self.employed_num):
                pivot = random.randint(0, len(self.paths[i][0]) - 1)
                old_part_path = self.paths[i][0][:pivot]
                new_part_path = self.local_search(self.paths[i][0][pivot],
self.paths[i][0][-1])

                if new_part_path[0] == -1:  # пошук в околицях не вдався
                    continue
```

```
                new_path = [old_part_path + new_part_path, 0]
                self.calculate_fitness(new_path)

                if new_path[1] < self.paths[i][1]:  # новий шлях краще старого
                    self.paths[i] = new_path

                if    self.path_counters[i][1]    >    FAIL_SEARCH_COUNT    and
self.record_fitness != i:
                    new_path = None
                    while new_path is None or new_path[0][0] == -1:
                        new_path    =    [self.local_search(self.start_vertex,
self.end_vertex), 0]

                    self.calculate_fitness(new_path)
                    self.paths[i] = new_path
                    self.path_counters[i] = [None, 0]

        def send_onlookers(self):
            proportional_values  =  [[i,  1  /  self.paths[i][1]]  for  i  in
range(self.employed_num)]
            for _ in range(self.onlookers_num):
                chosen_path = roulette_wheel_choice(proportional_values)
                pivot = random.randint(0, len(self.paths[chosen_path][0]) - 1)
                old_part_path = self.paths[chosen_path][0][:pivot]
                new_part_path                                                  =
self.local_search(self.paths[chosen_path][0][pivot],
self.paths[chosen_path][0][-1])

                if new_part_path[0] == -1:  # пошук в околицях не вдався
                    self.path_counters[chosen_path]          =          [None,
self.path_counters[chosen_path][1] + 1]
                    continue

                new_path = [old_part_path + new_part_path, 0]
                self.calculate_fitness(new_path)

                if new_path[1] < self.paths[chosen_path][1]:  # новий шлях краще
старого
                    self.paths[chosen_path] = new_path
                    self.path_counters[chosen_path] = [None, 0]

        def initialize_sources(self):
            init_path = None
```

```
            while init_path is None or init_path[0][0] == -1:
                init_path       =       [self.local_search(self.start_vertex,
self.end_vertex), 0]
                self.calculate_fitness(init_path)

            for i in range(self.employed_num):
                self.paths[i] = init_path

        def get_best_path(self):
            weights = []
            for i in range(len(self.paths[self.record_fitness][0]) - 1):
                cur_element = self.paths[self.record_fitness][0][i]
                next_element = self.paths[self.record_fitness][0][i+1]

weights.append(self.graph.adj_matrix[cur_element][next_element])
            return self.paths[self.record_fitness], weights


    def roulette_wheel_choice(values):
        weights = [value[1] for value in values]
        chosen_value = random.choices(values, weights=weights, k=1)[0]
        return chosen_value[0]


    def remove_duplicates(lst):
        seen = set()
        seen_index = {}
        for i, value in enumerate(lst):
            if value in seen:
                start_index = seen_index[value]
                end_index = i
                lst = lst[:start_index] + lst[end_index:]
                break
            seen.add(value)
            seen_index[value] = i
        return lst


    import random

    class Graph:
        def __init__(self, size, vertex_lo, vertex_hi, length_lo, length_hi):
            self.adj_matrix = [[0] * size for _ in range(size)]
```

```
            self.size = size
            self.vertex_lo = vertex_lo
            self.vertex_hi = vertex_hi
            self.length_lo = length_lo
            self.length_hi = length_hi
            self.generate()

        def generate(self):
            for row_index in range(self.size - 1):
                taken = sum(1 for value in
self.adj_matrix[row_index][:row_index] if value != 0)
                bound = random.randint(self.vertex_lo, self.vertex_hi) - taken
                for i in range(bound):
                    pos_value = random.randint(row_index + 1, self.size - 1)
                    taken_y = sum(self.adj_matrix[pos_value][:row_index])
                    if taken_y >= self.vertex_hi:
                        continue
                    self.adj_matrix[row_index][pos_value] = 1
                    self.adj_matrix[pos_value][row_index] = 1

            for row_index in range(self.size):
                for col_index in range(row_index):
                    if self.adj_matrix[row_index][col_index]:
                        length_value = random.randint(self.length_lo,
self.length_hi)
                        self.adj_matrix[row_index][col_index] =
self.adj_matrix[col_index][row_index] = length_value
            self.is_valid_matrix()

        def is_valid_matrix(self):
            for i in range(self.size):
                non_zero_count = sum(1 for value in self.adj_matrix[i] if value
!= 0)
                if non_zero_count < self.vertex_lo or non_zero_count >
self.vertex_hi:
                    self.adj_matrix = [[0] * self.size for _ in
range(self.size)]
                    self.generate()

        def print_matrix(self):
            for row in self.adj_matrix:
                print(', '.join(map(str, row)))
```

### 3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

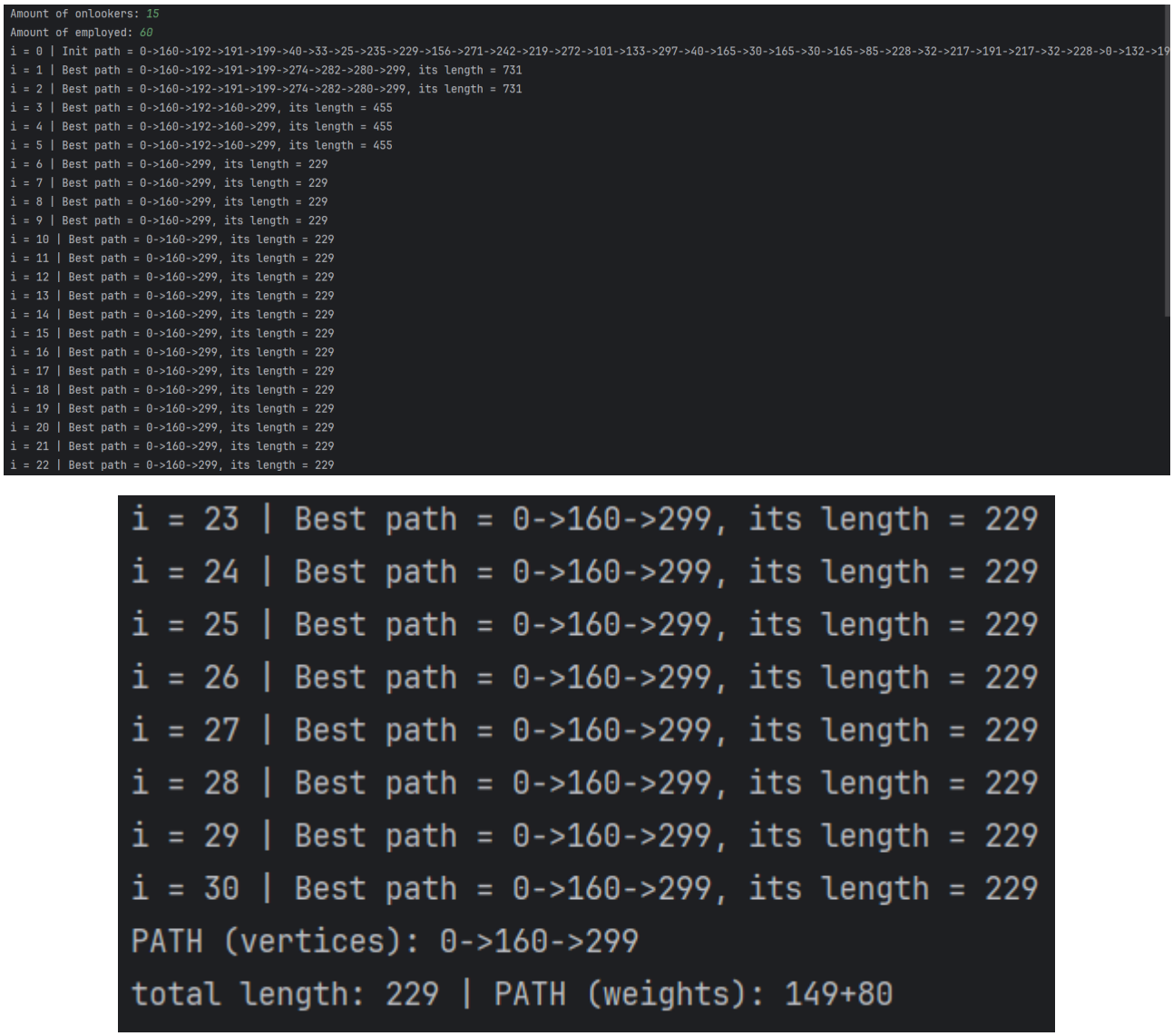



Рисунок 3.1 – Приклад роботи програми з 15 спостерігачами та 60 фуражирами

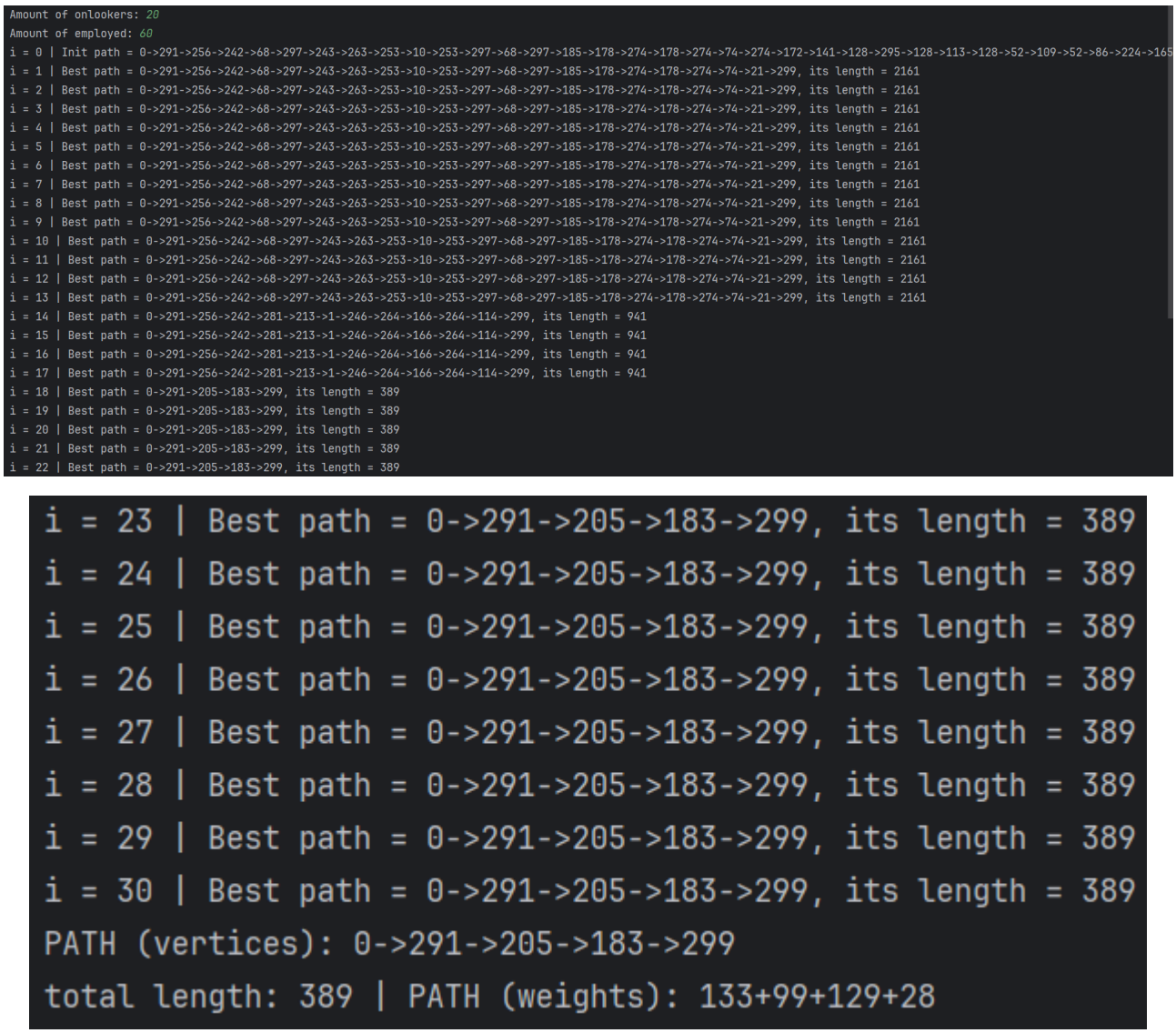



Рисунок 3.2 – Приклад роботи програми з 20 спостерігачами та 60 фуражирами

## 3.3 Тестування алгоритму

Усі випробування я проводжу на одному графі і одних вершинах.

Спочатку я фіксую число спостерігачів і змінюю число фуражирів. Кількість фуражирів обмежена 70-ма, оскільки при більших кількостях алгоритм стає занадто повільним. На рис. 3.3 можемо спостерігати залежність цільової функції від числа ітерацій при 10 спостерігачах та змінній кількості фуражирів.

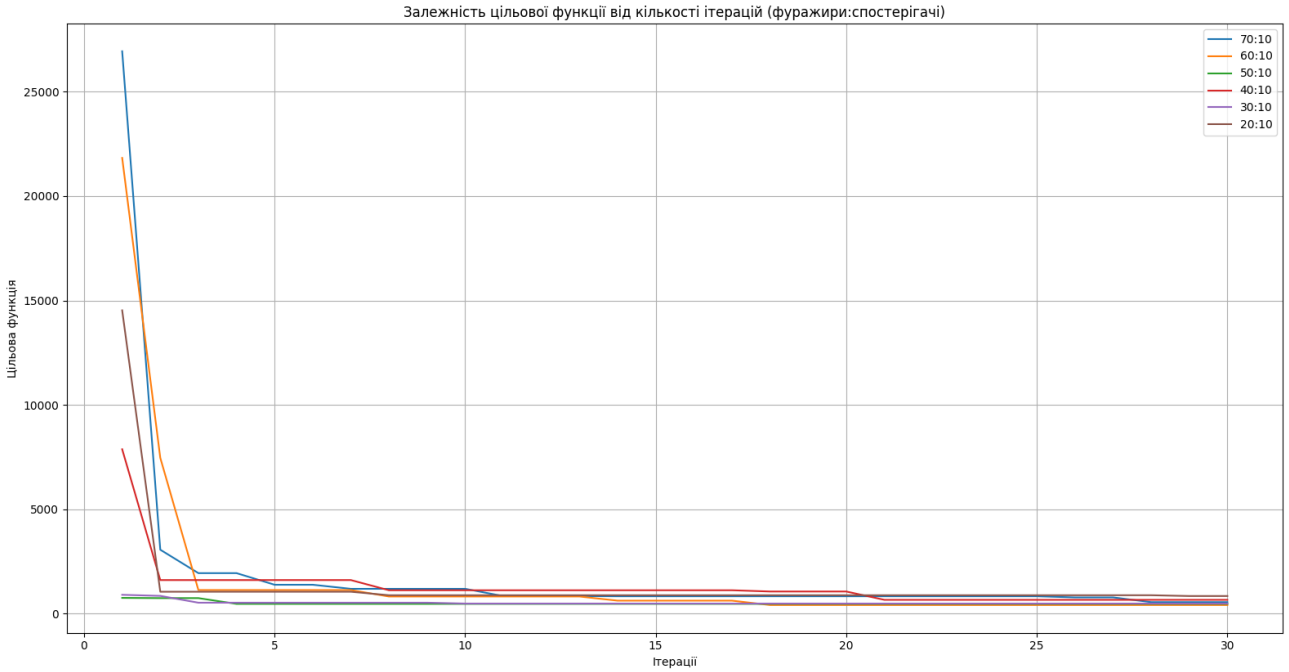



Рисунок 3.3 – Графік залежності цільової функції при 10 спостерігачах

Варіант з 30-ма фуражирами найефективніше знаходить рішення. Запам’ятаємо це. Змінимо кількість спостерігачів до 20 штук (рис. 3.4).

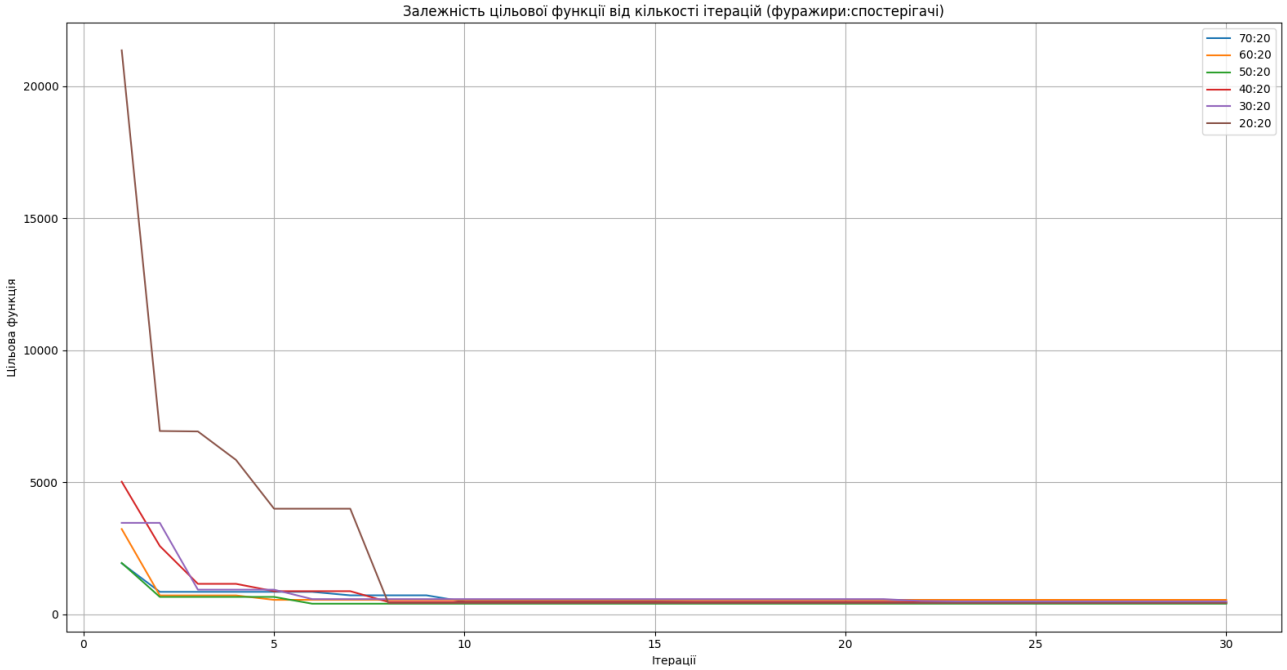



Рисунок 3.4 – Графік залежності цільової функції при 20 спостерігачах

Найкраще розв'язок знайшли варіанти з 50-ма та 60-ма фуражирами. Збільшимо кількість спостерігачів до 30 штук (рис. 3.5).

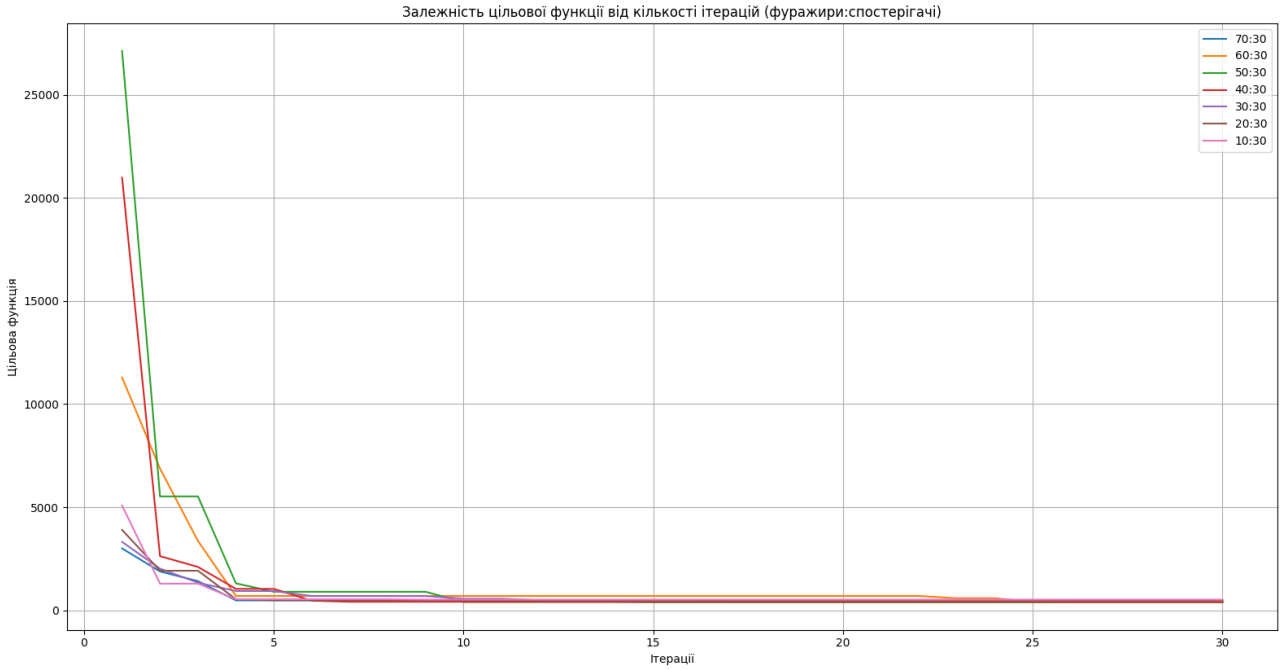



Рисунок 3.5 – Графік залежності цільової функції при 30 спостерігачах

Помічаємо, що найкраще розв'язок знаходять варіації з 50-ма та 60-ма фуражирами. Можемо зробити висновок, що кількість спостерігачів повинна складати від третини до половини кількості фуражирів. Зафіксуємо число

фуражирів на 60 штук, оскільки це значення найбільш ефективне у плані часозатратності та оптимальності розв'язку та спробуємо знайти більш чіткі значення оптимальної кількості спостерігачів (рис. 3.6).

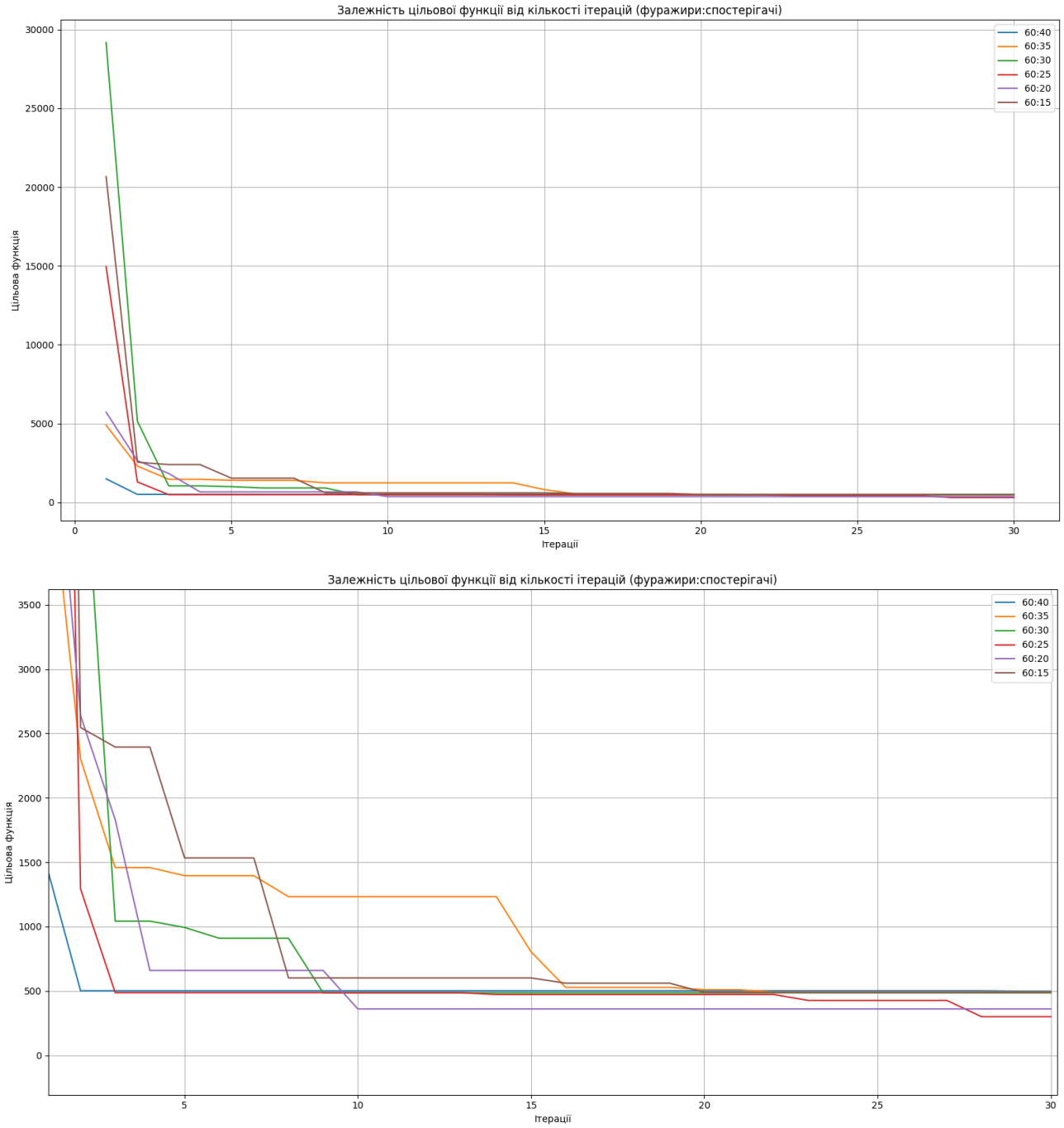



Рисунок 3.5 – Графік залежності цільової функції при 60 фуражирах

Пік ефективності помітний при 20 та 25 спостерігачах. Підсумовуючи, можна сказати, що найкращими в плані часу та оптимальності розв’язку є вхідні значення кількості фуражирів та спостерігачів, які відносяться як три до одного.

Загалом, чим більше бджіл, тим краще алгоритм знаходить оптимальне рішення, однак він стає занадто повільним, тому 60 фуражирів та 20 спостерігачів є золотою серединою.

# ВИСНОВОК

В рамках даної лабораторної роботи я вивчив основні підходи розробки метаевристичних алгоритмів для типових прикладних задач. Результатом моєї роботи стала програмна реалізація рішення задачі про найкоротший шлях за допомогою бджолиного алгоритму. Я опрацював методологію підбору прийнятних параметрів алгоритму та визначив, що для мого алгоритму прийнятними вхідними значеннями є значення кількості фуражирів та спостерігачів, які відносяться як три до одного.

Пошук в околицях ділянки (шляху) я реалізував як випадковий вибір суміжного ребра, заснований на пропорційній селекції. Чим менша вага ребра, тим більший шанс у нього потрапити у шлях, однак є і шанс потрапити в шлях для ребр з високою вагою. Довжина шляху обмежена кількістю вершин.

Упродовж тестування отримані результати вказують на те, що для найкращої ефективності алгоритму кількість робочих бджіл має бути втричі більша за кількість спостерігачів. Кількість робочих бджіл для 30 ітерацій алгоритму варто обмежувати 60-70 штуками. Чим більше буде бджіл, тим більше алгоритм витратить часу, однак тим більший шанс, що він знайде оптимальне чи максимально близьке до оптимального рішення. Якщо ставити кількість робочих бджіл більше 100-150 штук, то алгоритм майже завжди знаходить оптимальне рішення, але цей процес займає досить багато часу.

Кількість ітерацій для заданого в умові графа та знайдених при тестуванні вхідних параметрів варто ставити в межах від 25 до 35 ітерацій, адже ще до 25 ітерацій рішення, як правило, значно покращується, а після може ще покращитися на кілька десятків пунктів.

# КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 24.12.2023 включно максимальний бал дорівнює – 5. Після 24.12.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

- покроковий алгоритм – 10%;
- програмна реалізація алгоритму – 45%;
- робота з гіт – 20%;
- тестування алгоритму– 20%;
- висновок – 5%.

+1 додатковий бал можна отримати за виконання та захист роботи до 17.12.2023