**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-22 Іщенко Кіра Віталіївна* _____
(шифр, прізвище, ім'я, по батькові)

**Перевірив** *Ахаладзе І.Е.* _____
(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.

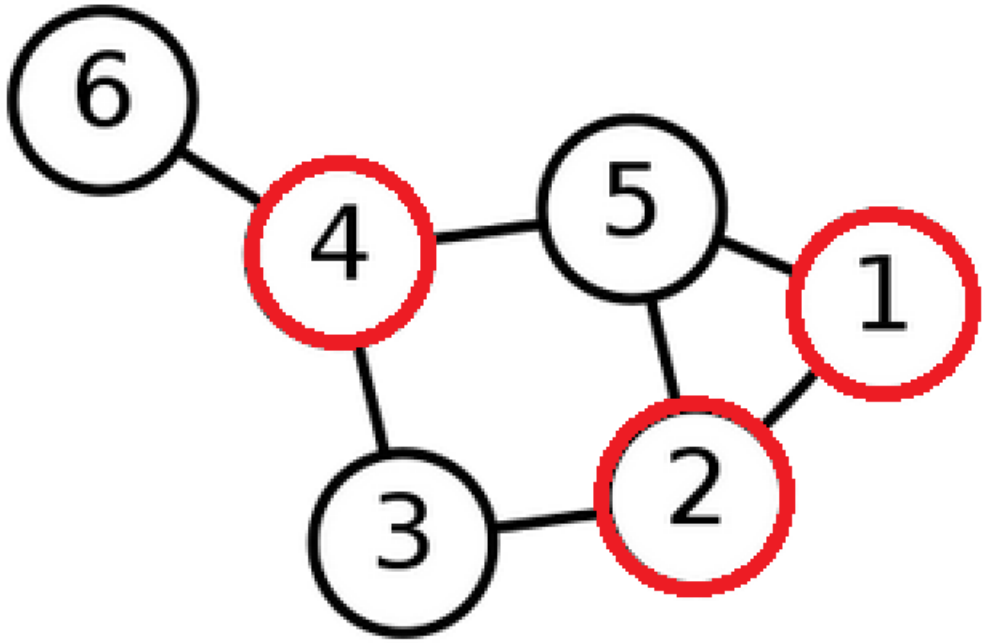
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

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

1. Генеруємо початкову популяцію розв’язків, обираючи досить велике число покритих вершин аби це з більшою ймовірністю був розв’язок.
2. Поки не досягнуто ліміту ітерацій:
    a. Пошук найкращого і рандомного батька, які не мають бути рівними
    b. Проведення схрещення батьків з використанням рівномірного оператора схрещування
    c. Повторити тричі:
        i. Провести мутацію
        ii. Якщо мутація не вдала:
            1. Повернутись до значення до мутації
        iii. Провести локальне покращення і перевірити чи воно дає розв’язок
        iv. Провести локальне покращення і перевірити чи воно дає розв’язок
        v. Додати рішення до популяції
        vi. Видалити найгіршу особину популяції
3. Повернути найкращий розв’язок

## 3.2 Програмна реалізація алгоритму

### 3.2.1 Вихідний код

```
#include <vector>
#include <random>
#include <algorithm>
#include <set>
#include "Graph.h"
#include <iostream>
using namespace std;
struct  Solution
```

```
{
        vector<int> value;
        int vertexCounter;
};

class GeneticAlgorithm
{
        Graph& graph;
        vector<Solution> solutions;
        Solution child;
        int populationSize;
        int bestValue;
        set<pair<int, int>> edgesInSolution;
        vector <int> bestVertex;

        bool isVertexCover(Solution&);
        pair<int, int> findParent();
        void evenCross(pair<int, int> parents, Solution& child);
        void mutation(Solution& child);
        void deleteBad();
        void localImprovementOperator(Solution& initialSolution);
        float generate(int min, int max);
        void genratePopulation();

public:
        GeneticAlgorithm(Graph& graph, int sizePopulation);
        vector<Vertex> solve(int iteration);
};
vector<Vertex> GeneticAlgorithm::solve(int iteration)
{
   if (iteration == 0) {
      return vector<Vertex>();
   }
   genratePopulation();
   bestVertex = graph.findBestVertex();
   pair<int, int> parent;
   for (int i = 0; i < iteration; i++) {
```

```
        parent = findParent();
        Solution newChild;
        evenCross(parent, newChild);

        for (int j = 0; j < 3; j++) {
            child = newChild;
            mutation(newChild);

            if (!isVertexCover(newChild)) {
                newChild = child;
            }
            localImprovementOperator(newChild);
            localImprovementOperator(newChild);
            solutions.push_back(newChild);
            deleteBad();
        }

        if (i % 20 == 0) {
            cout <<"Iteration:"<<i << " Num of vertex: " << bestValue << endl;
        }
    }
    cout << "Iteration:" << iteration << " Num of vertex: " << bestValue << endl;

    vector<Vertex> solutionVector;
    int bestSolutionIndex = findParent().first;

    for (int i = 0; i < solutions[bestSolutionIndex].value.size();i++) {
        if (solutions[bestSolutionIndex].value[i] == 1) {
            solutionVector.push_back(graph.vertices[i]);
        }
    }
    return solutionVector;
}

float GeneticAlgorithm::generate(int min, int max)
{
    random_device rd;
    mt19937 gen(rd());
```

```
    uniform_int_distribution<> dis(min, max);
    return dis(gen);
}

void GeneticAlgorithm::genratePopulation()
{
    vector<Solution> initialPopulation;

    while (initialPopulation.size() < populationSize) {
        Solution solution;
        int counter = 0;
        for (int j = 0; j < graph.vertexCount; ++j) {
            if (j < graph.vertexCount - generate(0,7)) {
                solution.value.push_back(1);
                counter++;
            }
            else {
                solution.value.push_back(0);
            }
        }

        random_shuffle(solution.value.begin(), solution.value.end());

        if (isVertexCover(solution)) {
            solution.vertexCounter = counter;
            initialPopulation.push_back(solution);

        }
    }
    solutions = initialPopulation;
}

bool GeneticAlgorithm::isVertexCover(Solution& sol)
{
    edgesInSolution.clear();

    for (int i = 0; i < sol.value.size(); i++) {
        if (sol.value[i] == 0) {
```

```
            continue;
        }

        for (const auto& edge : graph.vertices[i].edgesVertex) {
            edgesInSolution.insert(edge);
        }
    }
    if (edgesInSolution.size() == graph.edges.size()) {
        return true;
    }
    else {
        return false;
    }
}


pair<int, int> GeneticAlgorithm::findParent()
{
    int bestParent = 0;
    int secondParent = 0;
    for (int i = 0; i < solutions.size(); i++) {
        if (solutions[i].vertexCounter <= bestValue) {
            bestValue = solutions[i].vertexCounter;
            bestParent = i;
        }
    }
    do {
        secondParent = generate(0, solutions.size() - 1);
    } while (secondParent == bestParent);

    return { bestParent, secondParent};
}

void GeneticAlgorithm::evenCross(pair<int, int> parents, Solution& child) {

    vector <int> parentA = solutions[parents.first].value;
    vector <int> parentB = solutions[parents.second].value;
```

```
    int num = 0;
    child.value.clear();
    for (int i = 0; i < solutions[parents.first].value.size(); i++) {

        double randomValue = generate(0.1, 1.0);

        if (randomValue < 0.2) {
            child.value.push_back(parentA[i]);
            if (parentA[i] == 1) {
                num++;
            }
        }
        else {
            child.value.push_back(parentB[i]);
            if (parentB[i] == 1) {
                num++;
            }
        }
    }
    child.vertexCounter = num;

}

void GeneticAlgorithm::mutation(Solution& child) {

    vector<int> mutatedSolution = child.value;
    for (int i = 0; i < 3; i++) {
        int firstValue = generate(0, child.value.size()-1);
        int secondValue = generate(0, child.value.size()-1);
        int x = child.value[firstValue];
        child.value[firstValue] = child.value[secondValue];
        child.value[secondValue] = x;
    }
}

void GeneticAlgorithm::deleteBad()
{
    int badValue = 0;
```

```
    int badParent = 0;
    for (int i = 0; i < solutions.size(); i++) {
        if (solutions[i].vertexCounter > badValue) {
            badValue = solutions[i].vertexCounter;
            badParent = i;
        }
    }
    solutions.erase(solutions.begin() + badParent);

}


void GeneticAlgorithm::localImprovementOperator(Solution& initialSolution) {

    int randomIndex = generate(0, bestVertex.size() - 1);

    int index = bestVertex[randomIndex];
    Solution currentSolution = initialSolution;

    if (currentSolution.value[index] == 0) {
        currentSolution.value[index] = 1;
        currentSolution.vertexCounter++;
    }

    if (isVertexCover(currentSolution)) {
        initialSolution = currentSolution;

    }
}
```

### 3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

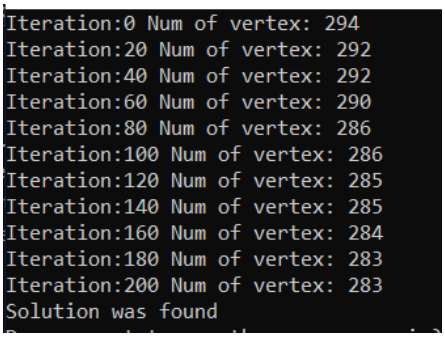



Рисунок 3.1 – Приклад роботи програми при значеннях популяції 200, кількість ітерацій 200

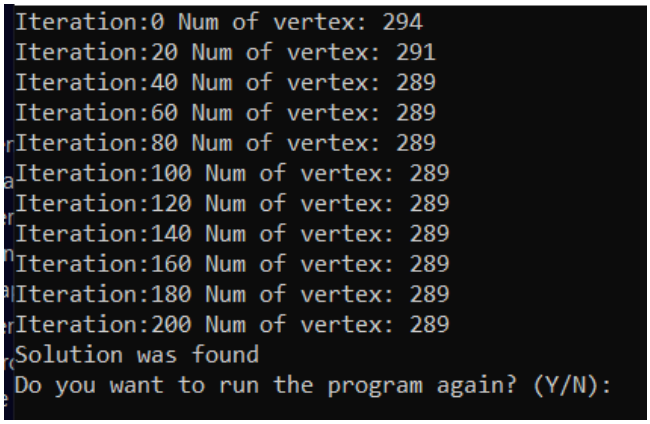



Рисунок 3.2 – Приклад роботи програми при значеннях популяції 50, кількість ітерацій 200

## 3.3 Тестування алгоритму

Почнемо з визначення параметру – чисельності популяції. Зафіксуємо кількість ітерацій на значення 800, а ймовірність схрещування 0,5.

Для значення чисельність популяції 500 маємо

| | |
|---|---|
| 0 | 294 |
| 100 | 286 |
| 200 | 284 |
| 300 | 277 |
| 400 | 276 |
| 500 | 273 |
| 600 | 272 |
| 700 | 271 |
| 800 | 271 |

Для значення чисельність популяції 300 маємо

| | |
|---|---|
| 0 | 294 |
| 100 | 286 |
| 200 | 286 |
| 300 | 283 |
| 400 | 282 |
| 500 | 278 |
| 600 | 278 |
| 700 | 278 |
| 800 | 278 |

Для значення чисельність популяції 50 маємо

| | |
|---|---|
| 0 | 294 |
| 100 | 286 |
| 200 | 281 |

| | |
|---|---|
| 300 | 277 |
| 400 | 276 |
| 500 | 276 |
| 600 | 276 |
| 700 | 276 |
| 800 | 276 |

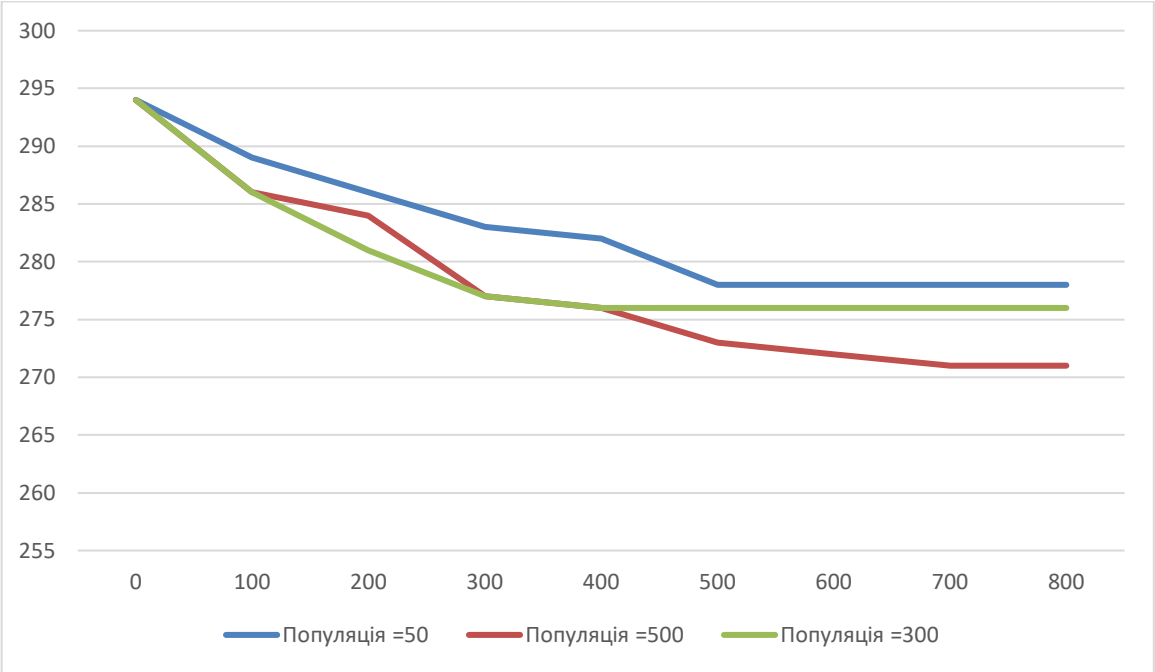



Рисунок 3.3.1 – Графік залежності кількості вершин у розв'язку від кількості ітерацій для популяції чисельністю 500

З цієї частини можемо зробити висновок, що чим більше популяція тим оптимальніше рішення може знайти алгоритм.

Продовжимо з параметром ймовірність схрещування. Зафіксуємо кількість ітерацій на значення 800, а чисельність популяції 500.

Для першого візьмемо ймовірність схрещування з першим батьком 0.8, а ймовірність схрещування з другим батьком 0.2

| | |
|---|---|
| 0 | 294 |
| 100 | 288 |

| | |
|---|---|
| 200 | 281 |
| 300 | 279 |
| 400 | 275 |
| 500 | 271 |
| 600 | 269 |
| 700 | 269 |
| 800 | 269 |

Тепер змінимо місцями параметри, виходить будемо надавати перевагу рандомному батьку, а не кращому.

| | |
|---|---|
| 0 | 294 |
| 100 | 286 |
| 200 | 279 |
| 300 | 275 |
| 400 | 272 |
| 500 | 270 |
| 600 | 268 |
| 700 | 268 |
| 800 | 268 |

Порівнюючи з попереднім значення не дуже відрізняються. Проте використання другого параметру дещо покращує роботу програми.

Оператор локального покращення залежить від рандомних величн і кількості вершин, що мають багато ребер, тому його неможливо модифікувати.

# ВИСНОВОК

В рамках даної лабораторної роботи розроблено генетичний алгоритм для розв’язання задачі вершинного покриття та обрано найкращі параметри для нього. Досить сильно на продуктивність впливає кількість особин у популяції, що логічно адже це дає більшу варіативність і простір для пошуку рішень. Для мутації я обрала рівномірний оператор схрещування і спробувала його модифікувати таким чином, що перевага від час вибору гену надається рандомно обраному батьку, а не кращому. Це дозволило трохи покращити отримане значення.

## 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