**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 № 3 з дисципліни

«Проектування алгоритмів»

„**Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.1**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-01 Галько Міла*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2021

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи формалізації метаеврестичних алгоритмів і вирішення типових задач з їхньою допомогою.

# Завдання

Згідно варіанту, розробити алгоритм вирішення задачі і виконати його програмну реалізацію на будь-якій мові програмування.

Зафіксувати якість отриманого розв'язку (значення цільової функції) після кожних 20 ітерацій до 1000 і побудувати графік залежності якості розв'язку від числа ітерацій.

Зробити узагальнений висновок.

|  |  |
| --- | --- |
| **№** | **Задача і алгоритм** |
| 6 | Задача розфарбовування графу (250 вершин, степінь вершини не більше 25, але не менше 2), бджолиний алгоритм ABC (число бджіл 35 із них 3 розвідники). |

# Виконання

## Програмна реалізація алгоритму

### Вихідний код

|  |
| --- |
| #include "ABCAlgorithm.hpp"  **int** main() {  srand(**static\_cast**<**unsigned** **int**>(time(**nullptr**)));  vector<vector<**bool**>> graph (3);  GraphGeneration(graph);  ABC(graph);  **return** 0;  } |
| #pragma once  #include <iostream>  #include <vector>  #include <time.h>  #include <stdlib.h>  #include <stdio.h>  **using** **namespace**::std; |
| #pragma once  #include "libs.hpp"  **class** Vertex {  **int** color;  **int** nectar;    **public**:  Vertex() {}  Vertex(**int** degree): color(0), nectar(degree) {}  **int** getNectar() {**return** nectar;}  **int** getColor() {**return** color;}  **void** setColor(**int** color) {**this**->color = color;}  **void** setNectar(**int** nectar) {**this**->nectar = nectar;}  };  **void** GraphGeneration(vector<vector<**bool**>> &graph);  //ABC algorithm  **void** ABC(vector<vector<**bool**>> graph);  **int** coloring(vector<Vertex> vertexes, vector<vector<**bool**>> graph);  //main steps in ABC  **void** setEmployedVertexes(vector<**int**> &employedVertexes, vector<Vertex> vertexes);  vector<**int**> onlookersAllocation(vector<**int**> employedVertexes, vector<Vertex> vertexes);  **void** onlookersColoring(vector<**int**> employedVertexes, vector<Vertex> &vertexes, vector<vector<**bool**>> graph, vector<**int**> onlookersOnVertex);  //additional functions  **int** getChromaticNumber(vector<Vertex> vertexes);  **bool** graphColored(vector<Vertex> vertexes);  **void** recountNectar(**int** vertex, vector<vector<**bool**>> graph, vector<Vertex> &vertexes); |
| #include "ABCAlgorithm.hpp"  **const** **int** VERTEXES = 250;  **const** **int** MIN\_VERTEX = 2, MAX\_VERTEX = 25;  **const** **int** ITERATIONS = 1000;  **int** EMPLOYEDBEES; // get info about nectar (chose vertexes)  **const** **int** CONST\_EMPLOYEDBEES = 3;  **const** **int** ONLOOKERBEES = 32; // pick nectar (color vertexes)  **void** GraphGeneration(vector<vector<**bool**>> &graph) {  graph.resize(VERTEXES);  **for** (**int** i = 0; i < VERTEXES; i++) {graph[i].resize(VERTEXES);}  **for** (**int** i = 0; i < VERTEXES; i++) {  **int** degree = 0;  **for** (**int** j = 0; j < VERTEXES; j++) {graph[i][j] ? degree++ : degree += 0;}  degree = rand() % (MAX\_VERTEX - 1) + MIN\_VERTEX - degree;    **for** (**int** j = 0; j < degree; j++) {  **int** randVertex;  **do** {randVertex = rand() % VERTEXES;} **while** (i == randVertex);  graph[i][randVertex] = **true**;  graph[randVertex][i] = **true**;  }  }  //GRAPH COUT (OPTIONAL)  /\*  for (auto line : graph) {  for (auto vertex: line) {cout << vertex << " ";}  cout << endl;  }  cout << endl;  \*/  }  // ITERATION PROCCESS  **void** ABC(vector<vector<**bool**>> graph) {  // creating vertexes with it's degree info  vector<Vertex> vertexes (VERTEXES);  **for** (**int** i = 0; i < VERTEXES; i++) {  **int** degree = 0;  **for** (**int** j = 0; j < VERTEXES; j++) {{graph[i][j] ? degree++ : degree += 0;}}  vertexes[i] = Vertex(degree);  }    // iteration process  **int** chromaticNum = coloring(vertexes, graph);  cout << "1) First chromatic number: " << chromaticNum << endl;  **int** bestChromaticNum = chromaticNum;  **for** (**int** i = 2; i <= ITERATIONS; i++) {  chromaticNum = coloring(vertexes, graph);  bestChromaticNum = bestChromaticNum > chromaticNum ? chromaticNum : bestChromaticNum;  **if** (i % 20 == 0) {cout << i <<") Best chromatic number: " << bestChromaticNum << endl;}  }  }  // MAIN PROCCESS  **int** coloring(vector<Vertex> vertexes, vector<vector<**bool**>> graph) {  **do** {  EMPLOYEDBEES = CONST\_EMPLOYEDBEES;  // FIRST STEP (EMPLOYED BEES) ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  vector<**int**> employedVertexes (0);  setEmployedVertexes(employedVertexes, vertexes);    // SECOND STEP (ONLOOKER BEES' ALLOCATION) ~~~~~~~~~~~~~~~~~~~~  vector<**int**> onlookersOnVertex = onlookersAllocation(employedVertexes, vertexes);    // THIRD STEP (ONLOOKER BEES' COLORING) ~~~~~~~~~~~~~~~~~~~~~~~  onlookersColoring(employedVertexes, vertexes, graph, onlookersOnVertex);  } **while** (!graphColored(vertexes));    **int** chromaticNumber = getChromaticNumber(vertexes);    //COLORS COUT (OPTIONAL)  /\*  cout << "Colors: ";  for (auto v: vertexes) {cout << v.getColor() << " ";}  cout << endl << endl; \*/    **return** chromaticNumber;  }  // FIRST STEP - SET ALL EMPLOYED BEES ON VERTEXES  **void** setEmployedVertexes(vector<**int**> &employedVertexes, vector<Vertex> vertexes) {  // find vertex for all employed bees  **int** hasNectar = 0;  **for** (**auto** v: vertexes) {**if** (v.getNectar()) hasNectar++;}  **int** enoughBees = EMPLOYEDBEES - hasNectar >= 0 ? hasNectar : EMPLOYEDBEES;    **for** (**int** i = 0; i < enoughBees; i++) {  **int** v;  // check if vertex is already added to employedVertexes or v's nectar == 0  **bool** alreadyAdded = **false**;  **do** {  alreadyAdded = **false**;  v = rand() % VERTEXES;  **for** (**auto** elem: employedVertexes) {  **if** (elem == v || vertexes[v].getNectar() == 0) {  alreadyAdded = **true**;  **break**;  }  }  } **while** (alreadyAdded);  employedVertexes.push\_back(v);  }  //EMPLOYED VERTEXES COUT (OPTIONAL)  /\*  cout << "Employed vertexes: ";  for (auto elem: employedVertexes) {cout << elem << " ";}  cout << endl; \*/  }  // SECOND STEP - ALLOCATE ALL ONLOOKER BEES ON CHOSEN VERTEXES  vector<**int**> onlookersAllocation(vector<**int**> employedVertexes, vector<Vertex> vertexes) {  **int** allNectar = 0;  **for** (**int** i = 0; i < employedVertexes.size(); i++) {  allNectar += vertexes[employedVertexes[i]].getNectar();  }  vector<**int**> beesProportion (0);  **int** startPoint = 0;  **for** (**int** v = 0; v < employedVertexes.size(); v++) {  **int** nectar = vertexes[employedVertexes[v]].getNectar();  **for** (**int** i = startPoint; i < nectar + startPoint; i++) {beesProportion.push\_back(v);}  }    // bees' allocation  vector<**int**> onlookersOnVertex (EMPLOYEDBEES);  **if** (ONLOOKERBEES - allNectar < 0) {  **for** (**int** bee = 0; bee < ONLOOKERBEES; bee++) {  **do** {  **int** pos = rand() % allNectar;  **if** (beesProportion[pos] != -1) {  onlookersOnVertex[beesProportion[pos]]++; // checking proportion range to get vertex number  beesProportion[pos] = -1;  **break**;  }  } **while** (**true**);  }  }  **else** {  **for** (**int** v = 0; v < employedVertexes.size(); v++) {  **int** nectar = vertexes[employedVertexes[v]].getNectar();  onlookersOnVertex[v] = nectar;  }  }    //(OPTIONAL)  /\*  cout << "Bees num in every vertex: ";  for (auto elem: onlookersOnVertex) {cout << elem << " ";}  cout << endl; \*/    **return** onlookersOnVertex;  }  // THIRD STEP - COLOR ADJACENT VERTEXES BY ONLOOKER BEES  **void** onlookersColoring(vector<**int**> employedVertexes, vector<Vertex> &vertexes, vector<vector<**bool**>> graph, vector<**int**> onlookersOnVertex) {  **for** (**int** v = 0; v < employedVertexes.size(); v++) {  **int** vertex = employedVertexes[v];  **int** onlookersCount = 0;    // firstly check uncolored vertexes ..  **for** (**int** adjacent = 0; adjacent < VERTEXES; adjacent++) {  **if** (graph[vertex][adjacent] && vertexes[adjacent].getColor() == 0) {  **int** color = 1;  **for** (**int** i = 0; i < VERTEXES; i++) {  **if** (graph[adjacent][i] && vertexes[i].getColor() == color) {  i = -1;  color++;  }  }  vertexes[adjacent].setColor(color);  onlookersCount++;  **if** (onlookersCount == onlookersOnVertex[v]) {**break**;}  }  }  // ..secondly check colored vertexes  **if** (onlookersCount < onlookersOnVertex[v]) {  vector<**int**> checkedVertexes (0);  **int** adjacent;  **do** {  **do** {adjacent = rand() % VERTEXES;} **while** (graph[v][adjacent] == 0);  **int** color = 1;  **for** (**int** i = 0; i < VERTEXES; i++) {  **if** (graph[adjacent][i] && vertexes[i].getColor() == color) {  i = -1;  color++;  }  }  vertexes[adjacent].setColor(color);  onlookersCount++;  } **while** (onlookersCount != onlookersOnVertex[v]);  }  recountNectar(vertex, graph, vertexes);  **if** (vertexes[vertex].getNectar() == 0) {  **int** color = 1;  **for** (**int** adjacent = 0; adjacent < VERTEXES; adjacent++) {  **if** (graph[vertex][adjacent] && vertexes[adjacent].getColor() == color) {  adjacent = -1;  color++;  }  }  vertexes[vertex].setColor(color);  EMPLOYEDBEES++;  }  }  }  // ADDITIONAL FUNCTIONS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  **int** getChromaticNumber(vector<Vertex> vertexes) {  **int** chromaticNumber = 2;  **for** (**auto** vertex: vertexes) {**if** (chromaticNumber < vertex.getColor()) chromaticNumber = vertex.getColor();}  **return** chromaticNumber;  }  **bool** graphColored(vector<Vertex> vertexes) {  **for** (**auto** vertex: vertexes) { **if**(vertex.getColor() == 0) **return** **false**;}  **return** **true**;  }  **void** recountNectar(**int** vertex, vector<vector<**bool**>> graph, vector<Vertex> &vertexes) {  **int** nectar = 0;  **for** (**int** adjacent = 0; adjacent < VERTEXES; adjacent++) {  **if** (graph[vertex][adjacent] && vertexes[adjacent].getColor() == 0) {  nectar++;  }  }  vertexes[vertex].setNectar(nectar);  } |

### Приклади роботи

На рисунку 3.1 показаний приклад роботи програми для 100 ітерацій при стартових параметрах зазначених у 6-ому варінті.

На рисунку 3.2 показаний приклад роботи програми для 15 вершин; степінь вершини: 2-10; число бджіл-розвідників: 3, робочих: 10; 10 ітерацій.

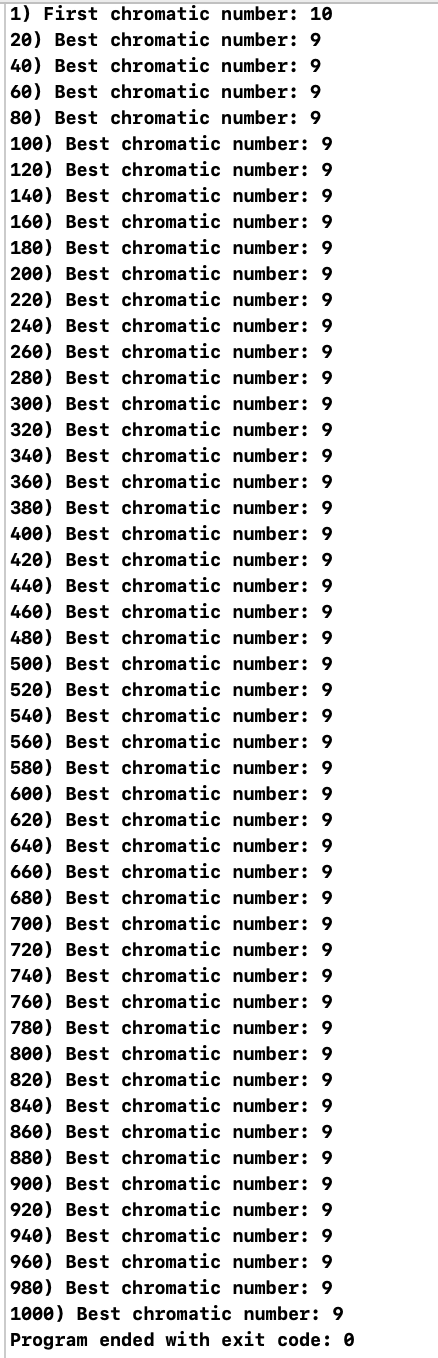


Рисунок 3.1

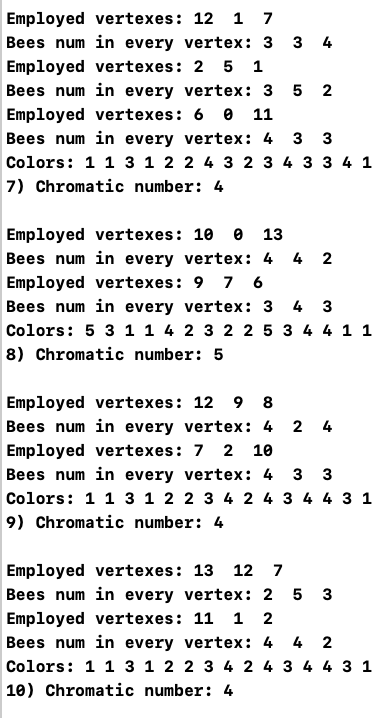
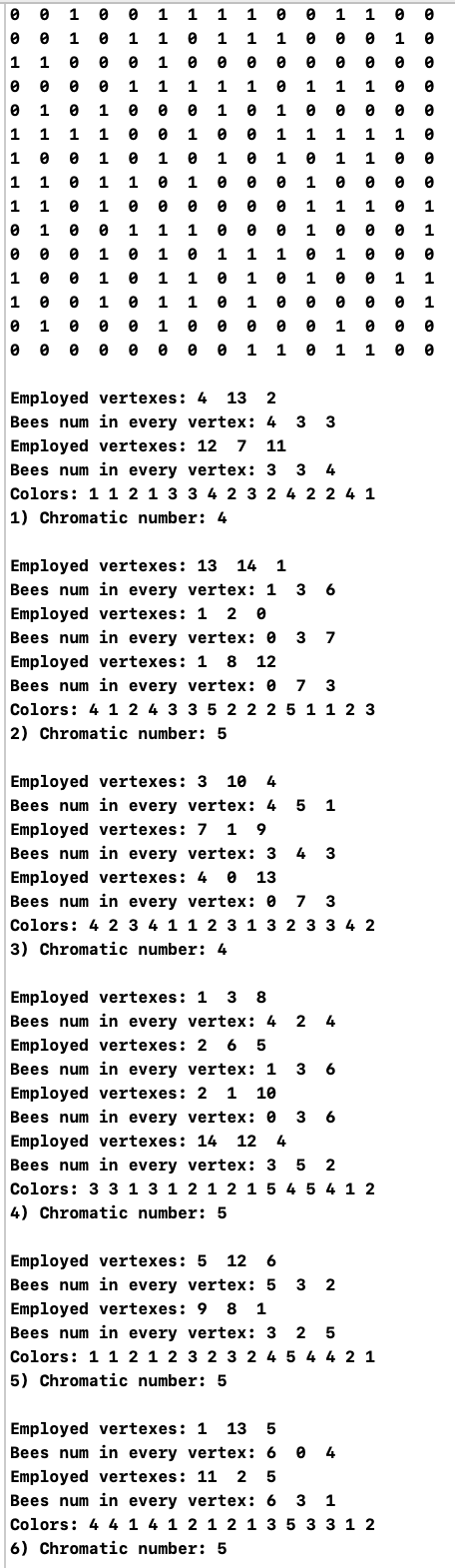


Рисунок 3.2 –

## Тестування алгоритму

### Значення цільової функції зі збільшенням кількості ітерацій

У списку наведено значення цільової функції зі збільшенням кількості ітерацій.

**1) First chromatic number: 10**

**20) Best chromatic number: 10**

**40) Best chromatic number: 9**

**60) Best chromatic number: 9**

**80) Best chromatic number: 9**

**100) Best chromatic number: 9**

**120) Best chromatic number: 9**

**140) Best chromatic number: 9**

**160) Best chromatic number: 9**

**180) Best chromatic number: 9**

**200) Best chromatic number: 9**

**220) Best chromatic number: 9**

**240) Best chromatic number: 9**

**260) Best chromatic number: 9**

**280) Best chromatic number: 9**

**300) Best chromatic number: 9**

**320) Best chromatic number: 9**

**340) Best chromatic number: 9**

**360) Best chromatic number: 9**

**380) Best chromatic number: 9**

**400) Best chromatic number: 9**

**420) Best chromatic number: 9**

**440) Best chromatic number: 9**

**460) Best chromatic number: 9**

**480) Best chromatic number: 9**

**500) Best chromatic number: 9**

**520) Best chromatic number: 9**

**540) Best chromatic number: 9**

**560) Best chromatic number: 9**

**580) Best chromatic number: 9**

**600) Best chromatic number: 9**

**620) Best chromatic number: 9**

**640) Best chromatic number: 9**

**660) Best chromatic number: 9**

**680) Best chromatic number: 9**

**700) Best chromatic number: 9**

**720) Best chromatic number: 9**

**740) Best chromatic number: 9**

**760) Best chromatic number: 9**

**780) Best chromatic number: 9**

**800) Best chromatic number: 9**

**820) Best chromatic number: 9**

**840) Best chromatic number: 9**

**860) Best chromatic number: 9**

**880) Best chromatic number: 9**

**900) Best chromatic number: 9**

**920) Best chromatic number: 9**

**940) Best chromatic number: 9**

**960) Best chromatic number: 9**

**980) Best chromatic number: 9**

**1000) Best chromatic number: 9**

### Графіки залежності розв'язку від числа ітерацій

На рисунку 3.3 наведений графік, який показує якість отриманого розв'язку.

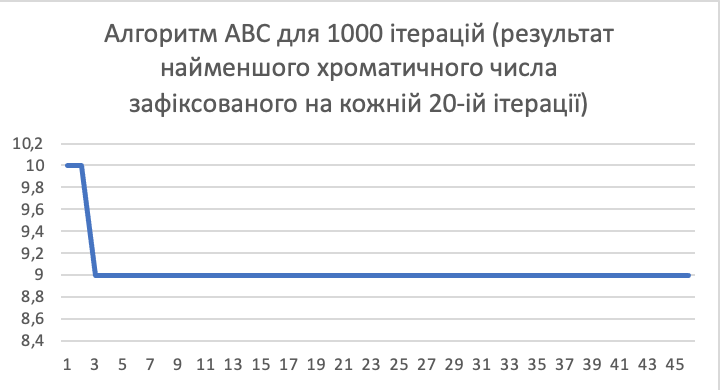


Рисунок 3.3 – Графіки залежності розв'язку від числа ітерацій

Висновок

В рамках даної лабораторної роботи я вивчила основні підходи формалізації метаеврестичних алгоритмів і вирішила за їх допомогою задачу розфарбування графу бджолиним алгоритмом ABC. Дослідивши усі етапи даного алгоритму, я розробила програмну реалізацію на мові С++.

Також мною була досліджена залежність якості розв’язку від числа ітерацій (1000 ітерацій). В ході тестування фіксувався найменше хроматичне число графу на кожній 20-ій ітерації. Як висновок можна зазначити, що хроматичне число зменшується в ході усіх ітерацій, але дане число не є надто малим порівняно з хроматичним числом 1-ої ітерації. Наочно дану закономірність можна помітити на графіках залежності. Також найчастіше число після першої ітерації співпадає із найкращим результатом тестування. Отже, можна сказати, що використання алгоритму ABC є резонним для розфарбування графу, адже одразу дає непогані результати і немає великої потреби у великій кількості ітерацій.

Критерії оцінювання

При здачі лабораторної роботи до 5.11.2021 включно максимальний бал дорівнює – 5. Після 5.11.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* програмна реалізація алгоритму – 75%;
* тестування алгоритму– 20%;
* висновок – 5%.