

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

09.03.01 Информатика и вычислительная техника
код и наименование направления подготовки

ОТЧЕТ
по учебной практике

по направлению 09.03.01 «Информатика и вычислительная техника»,
направленность (профиль) – «Электронно-вычислительные машины, комплексы, системы и
сети», квалификация – бакалавр,
программа академического бакалавриата,
форма обучения – очная, год начала подготовки (по учебному плану) – 2016

Выполнил:
студент гр. ИВ-621
«06» июля 2018 г.

/Сенченко А.П./

Оценка «_____»

Руководитель практики
от университета
к.т.н., доцент Кафедры ВС
«07» июля 2018 г.

/Молдованова О.В./

Новосибирск 2018

ПЛАН-ГРАФИК ПРОВЕДЕНИЯ ПРОИЗВОДСТВЕННОЙ ПРАКТИКИ

Тип практики: учебная практика по получению первичных профессиональных

Способ проведения практики: стационарная

Форма проведения практики: дискретно по периодам проведения практики

Тема: Алгоритмы раскраски графов.

Содержание практики

Наименование видов деятельности	Дата (начало – окончание)
Общее ознакомление со структурным подразделением СибГУТИ, вводный инструктаж по технике безопасности	05.02.18-17.02.18
Выдача задания на практику, определение конкретной индивидуальной темы, формирование плана работ	19.02.18-24.02.18
Работа с библиотечными фондами структурного подразделения или предприятия, сбор и анализ материалов по теме практики	26.02.18-24.03.18
Выполнение работ в соответствии с составленным планом	26.03.18-02.06.18
Анализ полученных результатов и произведенной работы, составление отчета по практике	02.07.18-07.07.18

Согласовано:

Руководитель практики
от университета
к.т.н., доцент Кафедры ВС

_____ / Молдованова О.В./

Оглавление

1. ЗАДАНИЕ НА ПРАКТИКУ	4
2. ВВЕДЕНИЕ.....	4
2.1. Практическое применение раскраски графов	4
2.2. Краткая теория использованных алгоритмов.....	4
2.2.1. Закрашивание двумя цветами	4
2.2.2. Закрашивание тремя цветами.....	4
2.2.3. Закрашивание четырьмя цветами	5
2.2.4. Другие алгоритмы закрашивания карт	6
3. ОСНОВНАЯ ЧАСТЬ.....	6
4. ЗАКЛЮЧЕНИЕ	14
5. СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ.....	14
6. ЛИСТИНГ	14

1. ЗАДАНИЕ НА ПРАКТИКУ

Реализовать алгоритмы раскраски графа в 2, 3, 4, 5 и неопределенное количество цветов (100% раскрашивание).

2. ВВЕДЕНИЕ

2.1. Практическое применение раскраски графов

Закрашивание графа используется во многих отраслях, например:

- Вычисление производных
- Распределение частот
- Составление расписаний
- Распределение регистров в микропроцессорах
- Распараллеливание численных методов

2.2. Краткая теория использованных алгоритмов

Цель подобного алгоритма — раскрашивание карты таким образом, чтобы на ней не было двух смежных участков одинакового цвета. Конечно, это можно сделать, используя разные цвета для каждого участка. Вопрос состоит в том, с помощью какого минимального количества цветов реально раскрасить конкретную карту.

2.2.1. Закрашивание двумя цветами

В раскрашивании такого рода карт нет ничего сложного. Выберите любой участок и назначьте ему один из двух цветов. Затем задайте всем его соседям другой цвет и рекурсивно обойдите их, раскрашивая смежные участки. Если вы столкнетесь с тем, что соседний узел уже закрашен тем же цветом, каким заполнен и текущий, это будет означать следующее: карту нельзя сделать двухцветной.

2.2.2. Закрашивание тремя цветами

Определение того, является ли карта трехцветной, — крайне сложная задача. Фактически еще не изобретено такого алгоритма, который мог бы ее решить за полиномиальное время.

Одно из довольно очевидных решений заключается в применении к узлам каждого из трех цветов и поиске подходящей комбинации. Если сеть содержит N узлов, время поиска будет равно $O(3^N)$; это довольно медленно, если N большое.

В данном случае вы можете убрать из сети узел, у которого меньше трех соседей, раскрасить уменьшенную сеть и затем вернуть его обратно, назначив ему цвет, не примененный смежными узлами.

Удаление узла из сети уменьшает количество соседей для оставшихся узлов. Если повезет, сеть будет уменьшаться, пока в ней не останется всего один узел. Выбрав для него цвет, происходит постепенное возвращение удаленных узлов, раскрашивая каждый из них.

2.2.3. Закрашивание четырьмя цветами

Согласно теореме о четырех красках, любую карту можно закрасить четырьмя цветами. Впервые эта теорема была предложена Фрэнсисом Гутри в 1852 г. Она активно изучалась на протяжении 124 лет, пока в 1974 г. ее, наконец, не доказали Кеннет Appel и Вольфганг Хакен. К сожалению, их доказательство включает исчерпывающее исследование набора из 1936 специально подобранных карт, поэтому оно не предлагает надежного метода для раскрашивания любой карты четырьмя цветами.

Теорема о четырех красках исходит из того, что сеть является планарной. Это означает, что ее можно нарисовать на плоскости без пересечения звеньев. Звенья должны иметь вид прямых линий; они могут отклоняться и перемещаться по всей поверхности, но только не пересекаться.

Если сеть не планарная, нет никакой гарантии того, что ее получится закрасить четырьмя цветами. Например, вы можете создать 10 узлов с 90 звеньями, соединяющими каждую из пар. Поскольку все узлы связаны между собой, для закрашивания сети понадобится 10 цветов. Однако сеть, созданная из обычной карты, является планарной.

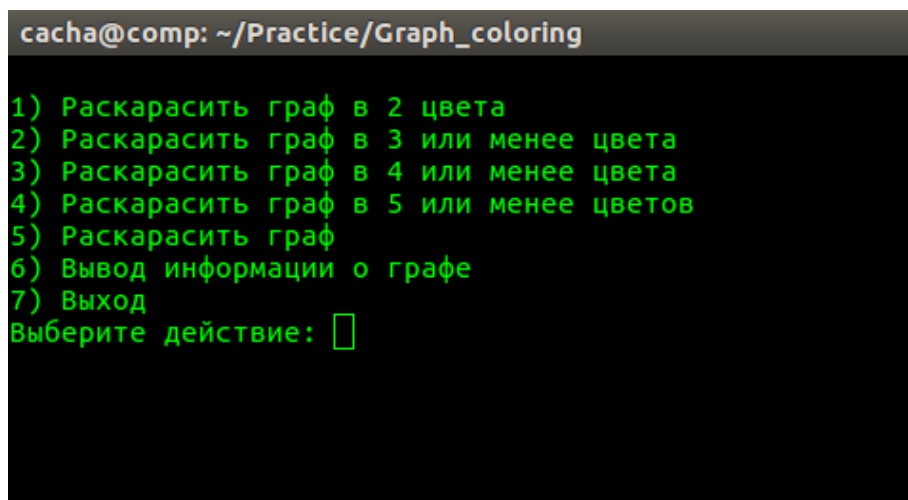
Для закрашивания используется алгоритм похожий на предыдущий, единственным отличием является только то, что из сети удаляются узлы имеющие степень меньше 4.

2.2.4. Другие алгоритмы закрашивания карт

Стратегия восхождения на вершину может циклически перебирать узлы сети, выдавая каждому из них первый цвет, который не используется ни одним из его соседей. Этот алгоритм не всегда позволяет закрасить сеть минимальным количеством цветов, однако при этом является чрезвычайно простым и быстрым. Он работает даже для не планарных сетей и в некоторых ситуациях дает возможность использовать для закрашивания четыре цвета.

3. ОСНОВНАЯ ЧАСТЬ

Меню программы. После выбора алгоритма закрашивания и успешного его выполнения запускается функция формирования файла с расширением “gv”, затем на основе этого файла создается изображение “graph.png”.

A screenshot of a terminal window with a dark background. The title bar at the top reads 'cacha@comp: ~/Practice/Graph_coloring'. The terminal displays a menu with seven options in green text: '1) Раскрасить граф в 2 цвета', '2) Раскрасить граф в 3 или менее цвета', '3) Раскрасить граф в 4 или менее цвета', '4) Раскрасить граф в 5 или менее цветов', '5) Раскрасить граф', '6) Вывод информации о графе', and '7) Выход'. Below the menu, it says 'Выберите действие: ' followed by a small square cursor icon.

```
cacha@comp: ~/Practice/Graph_coloring
1) Раскрасить граф в 2 цвета
2) Раскрасить граф в 3 или менее цвета
3) Раскрасить граф в 4 или менее цвета
4) Раскрасить граф в 5 или менее цветов
5) Раскрасить граф
6) Вывод информации о графе
7) Выход
Выберите действие: □
```

Рисунок 1

Вывод программы, если алгоритм не может раскрасить граф. Цифра обозначает в какое количество цветов не удалось раскрасить граф. Такая ошибка может случиться только у алгоритмов, которые раскрашивают граф в определенное количество цветов.

```
cache@comp: ~/Practice/Graph_coloring
Error, 2 color

1) Раскрасить граф в 2 цвета
2) Раскрасить граф в 3 или менее цвета
3) Раскрасить граф в 4 или менее цвета
4) Раскрасить граф в 5 или менее цветов
5) Раскрасить граф
6) Вывод информации о графе
7) Выход
Выберите действие: 
```

Рисунок 2

Вывод информации о графе, 6-ая команда. В данном случае граф раскрашен, иначе в столбце “Color” стояло бы значение “-1” у всех узлов. Номера цветов: 0 – red, 1 – lawngreen, 2 – deeppink, 3 – cyan, 4 – indigo, 5 – purple.

```
cache@comp: ~/Practice/Graph_coloring

Number node = 8
Index  Node      Next      Parent      Color  NumCont Status
0      0x8f2a040    0x8f2a060    (nil)        1      3      1
>ind: 1 color: 0
>ind: 3 color: 3
>ind: 7 color: 0
1      0x8f2a060    0x8f2a080    0x8f2a040    0      3      1
>ind: 0 color: 1
>ind: 3 color: 3
>ind: 4 color: 2
2      0x8f2a080    0x8f2a0a0    0x8f2a060    1      3      1
>ind: 3 color: 3
>ind: 5 color: 0
>ind: 7 color: 0
3      0x8f2a0a0    0x8f2a0c0    0x8f2a080    3      6      1
>ind: 0 color: 1
>ind: 1 color: 0
>ind: 2 color: 1
>ind: 4 color: 2
>ind: 5 color: 0
>ind: 6 color: 1
4      0x8f2a0c0    0x8f2a0e0    0x8f2a0a0    2      4      1
>ind: 1 color: 0
>ind: 3 color: 3
>ind: 6 color: 1
>ind: 7 color: 0
5      0x8f2a0e0    0x8f2a100    0x8f2a0c0    0      3      1
>ind: 2 color: 1
>ind: 3 color: 3
>ind: 6 color: 1
6      0x8f2a100    0x8f2a120    0x8f2a0e0    1      4      1
>ind: 3 color: 3
>ind: 4 color: 2
>ind: 5 color: 0
>ind: 7 color: 0
7      0x8f2a120    (nil)        0x8f2a100    0      4      1
>ind: 0 color: 1
>ind: 2 color: 1
>ind: 4 color: 2
>ind: 6 color: 1
```

Рисунок 3

Изначально графы находятся в файле Matrix.txt в виде матрицы смежности.

Пример раскраски графа в 2 цвета. Алгоритмы раскрашивания в 3, 4, 5 цветов раскрашивают граф иначе. Последний алгоритм закрашивает граф также как и первый.

Matrix.txt

```
0 1 0 1 0 0 0 1
1 0 0 0 1 0 0 0
0 0 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 0 0 1
0 0 1 0 0 0 1 0
0 0 0 1 0 1 0 1
1 0 1 0 1 0 1 0
```

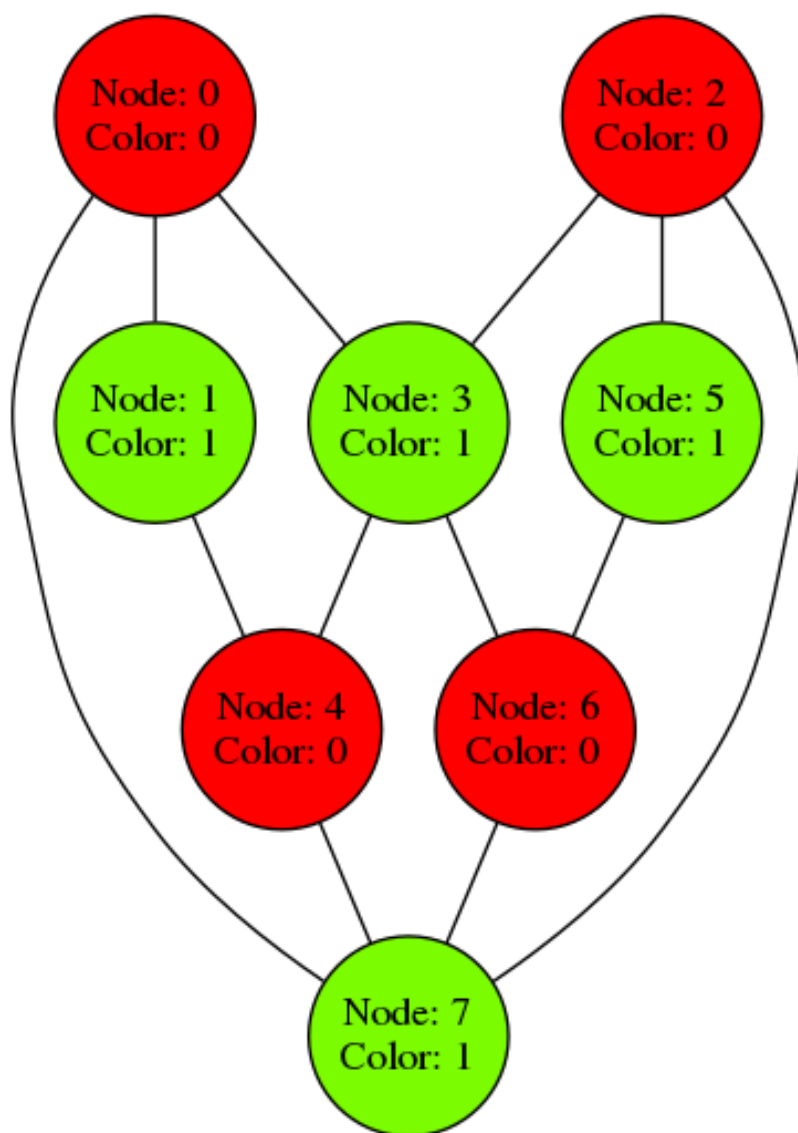


Рисунок 4

Пример раскраски графа в 3 цвета.

Matrix.txt

```
0 1 0 1 0 0 0 1
1 0 0 0 1 0 0 0
0 0 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 0 1 1
0 0 1 0 0 0 1 0
0 0 0 1 1 1 0 1
1 0 1 0 1 0 1 0
```

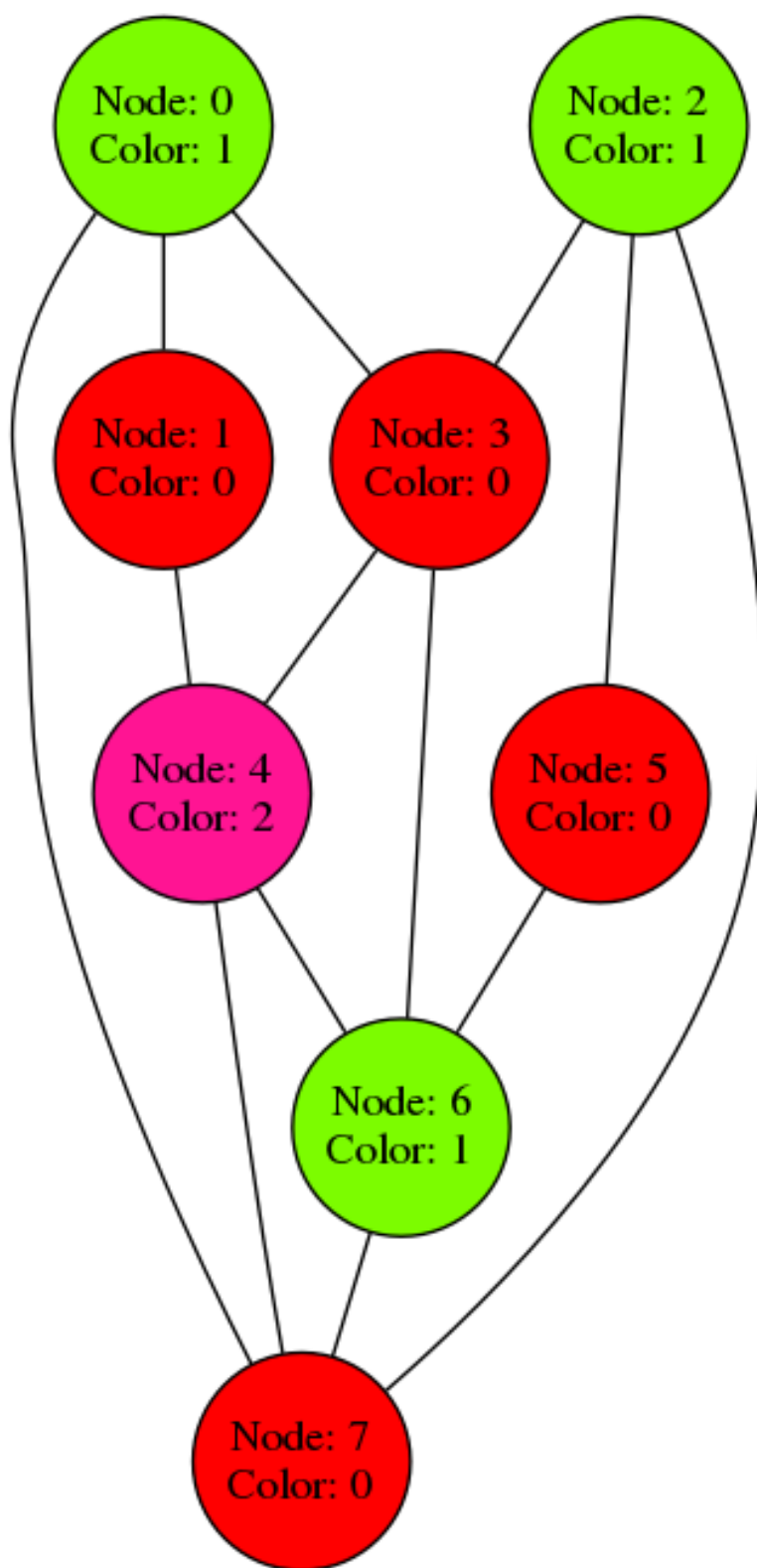


Рисунок 5

Пример раскраски графа в 4 цвета.

Matrix.txt

```
0 1 0 1 0 0 0 1
1 0 0 1 1 0 0 0
0 0 0 1 0 1 0 1
1 1 1 0 1 1 1 0
0 1 0 1 0 0 1 1
0 0 1 1 0 0 1 0
0 0 0 1 1 1 0 1
1 0 1 0 1 0 1 0
```

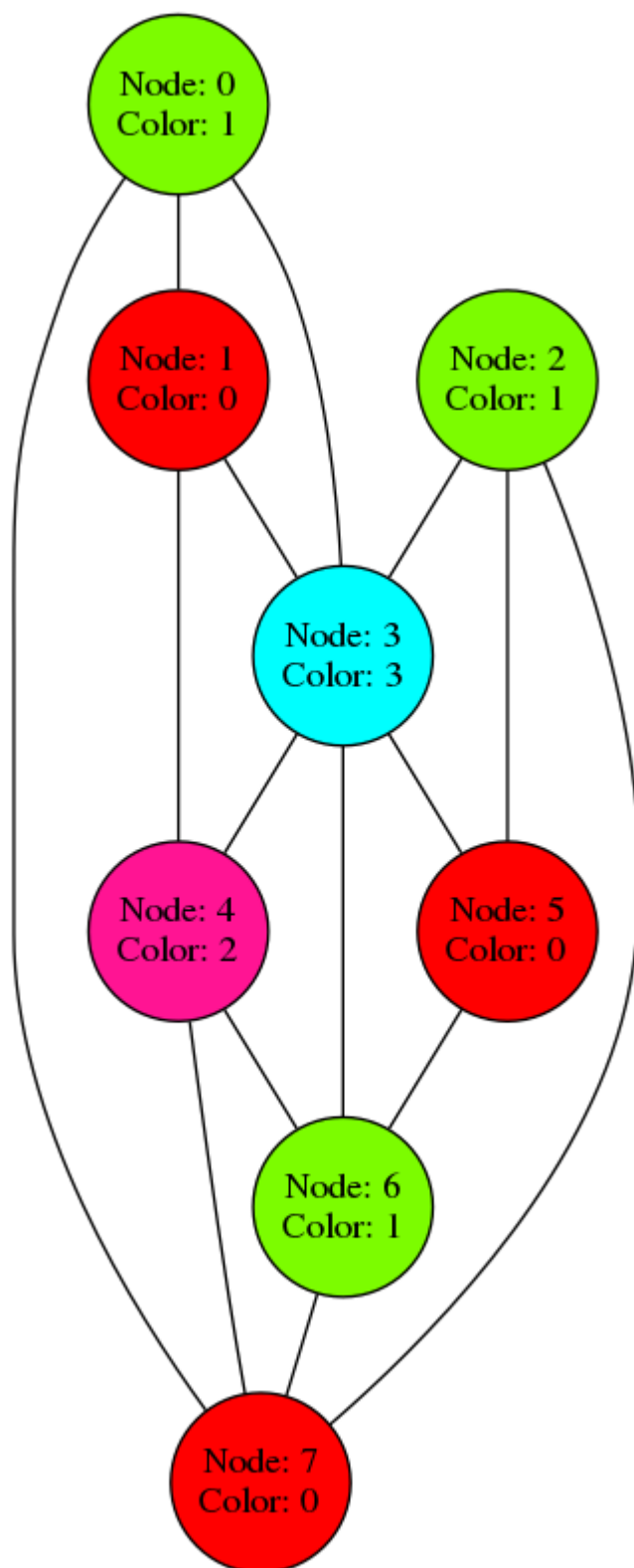


Рисунок 6

Пример раскраски графа в 5 цветов.

Matrix.txt

```
0 1 1 1 1 0 0 1
1 0 1 1 1 0 0 0
1 1 0 1 0 1 0 1
1 1 1 0 1 1 1 1
1 1 0 1 0 1 1 1
0 0 1 1 1 0 1 1
0 0 0 1 1 1 0 1
1 0 1 1 1 1 1 0
```

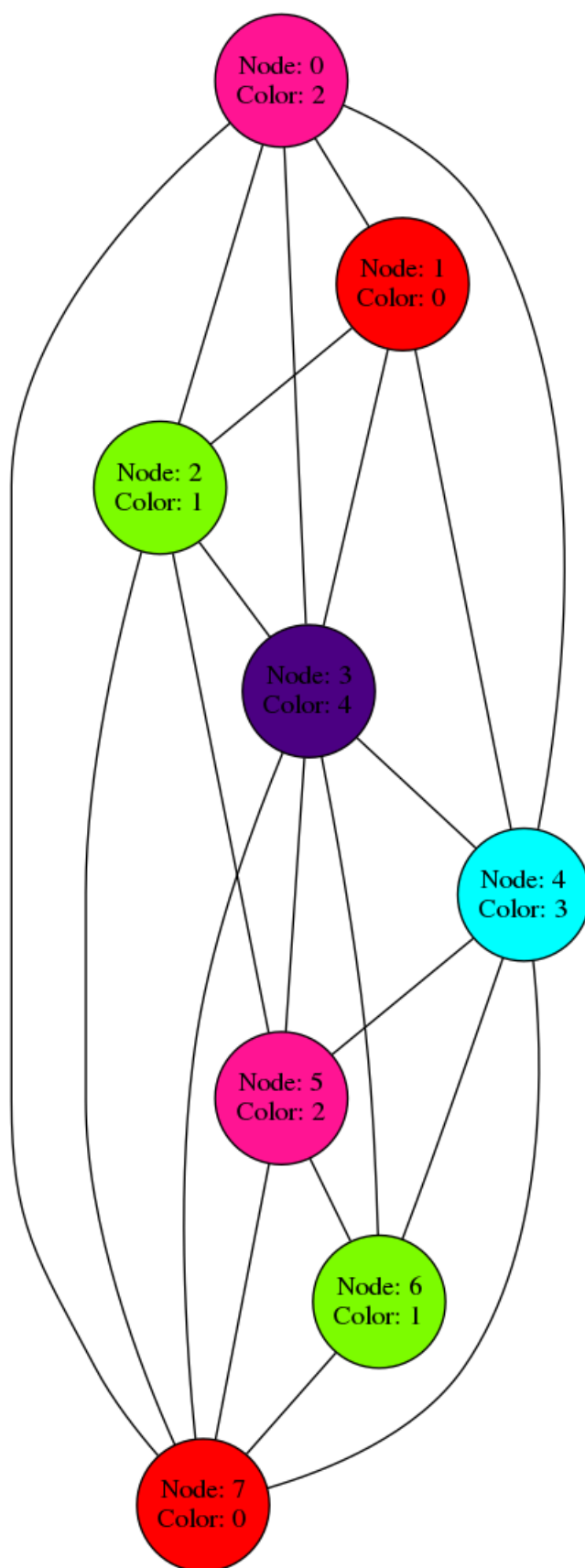


Рисунок 7

Пример раскраски графа в столько цветов, сколько необходимо его закрасить.

Matrix.txt

```

0 1 1 1 1 0 0 1
1 0 1 1 1 0 0 1
1 1 0 1 1 1 0 1
1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1
0 0 1 1 1 0 1 1
0 0 0 1 1 1 0 1
1 1 1 1 1 1 1 0

```

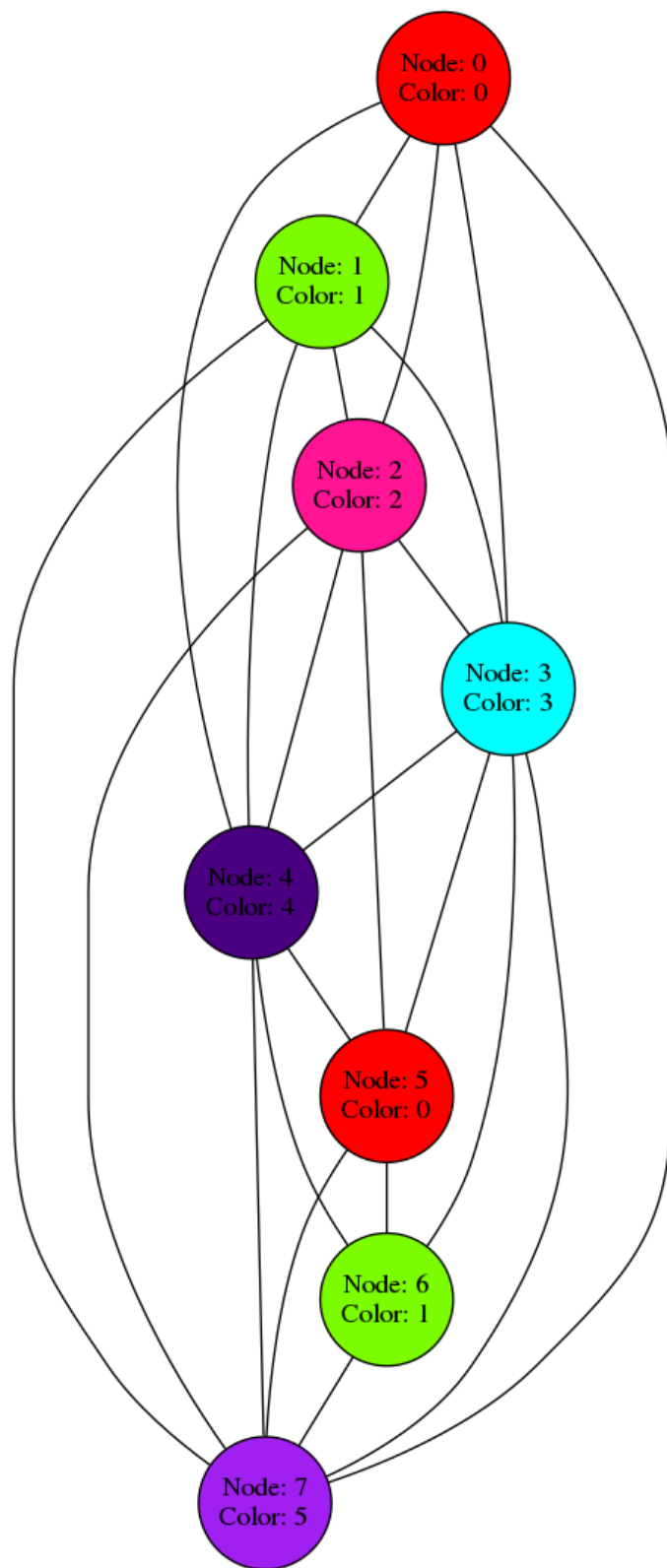


Рисунок 8

Данный граф уже был приведен в пример 4 цветной раскраски, различие в алгоритме закрашивания, результат получился немного другим.

Matrix.txt

```
0 1 0 1 0 0 0 1
1 0 0 1 1 0 0 0
0 0 0 1 0 1 0 1
1 1 1 0 1 1 1 0
0 1 0 1 0 0 1 1
0 0 1 1 0 0 1 0
0 0 0 1 1 1 0 1
1 0 1 0 1 0 1 0
```

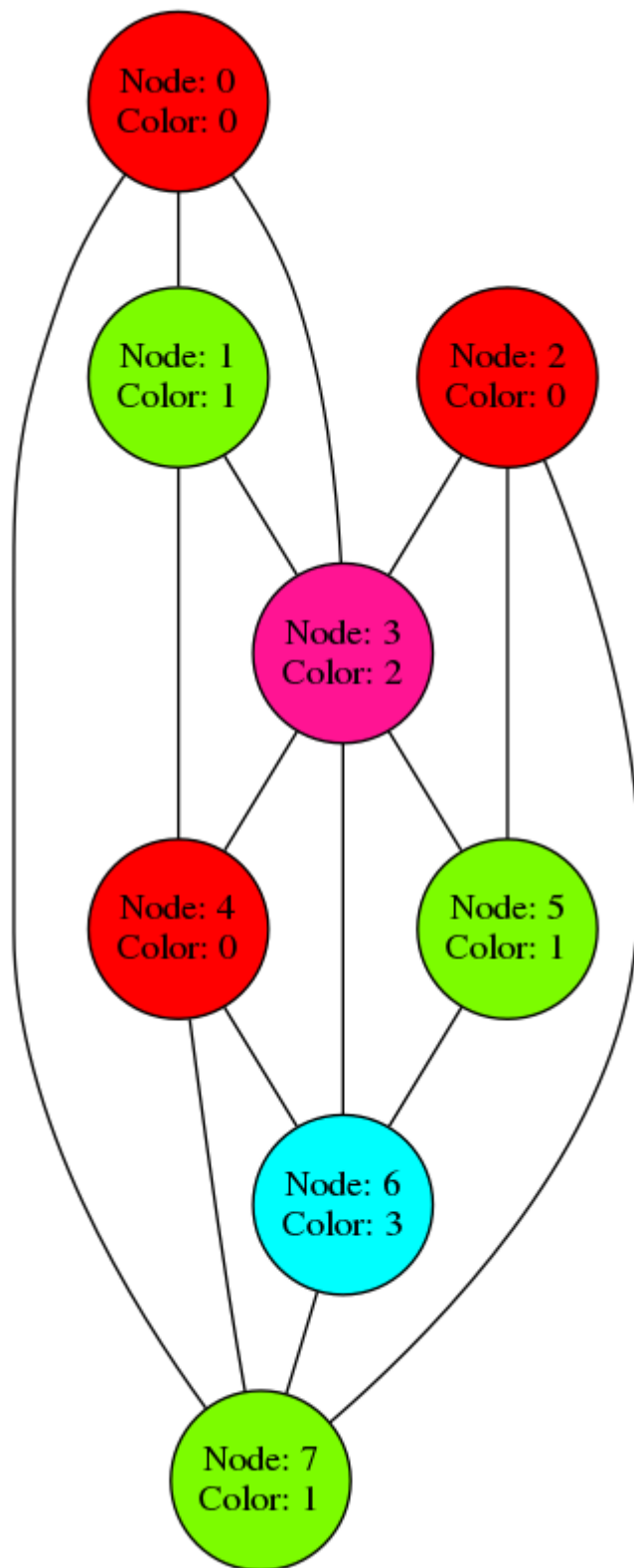


Рисунок 9

4. ЗАКЛЮЧЕНИЕ

В ходе работы были реализованы алгоритмы закрашивания графа. В процессе реализации стало понятно, что окрашивание в 2 и неопределенное количество цветов были довольно простыми, в отличие от трех других. Из экспериментов можно сделать вывод, что если вам нужно раскрасить любой достаточно большой граф, не зависимо сколько цветов будет использовано, то лучше использовать последний алгоритм. В дальнейшем можно заняться оптимизацией и распараллеливанием алгоритмов.

5. СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

- 1) Кнут Д. У. Искусство программирования: В 3 т.: Пер. с англ. Т.1: Основные алгоритмы, 2002. 712 с.
- 2) Род Стивенс. Алгоритмы. Теория и практическое применение, 2016. 544 с.
- 3) Graphviz – Graph Visualization Software. [Электронный ресурс]. URL: <https://www.graphviz.org>
- 4) Википедия // Практическое применение раскраски графов. [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Практическое_применение_раскраски_графов

6. ЛИСТИНГ

main.c

```
#include "GraphColoring.h"

int main()
{
    int line = 0;
    int column = 0;
    int *Matrix = NULL;
    struct Graph *graph;

    GetMatrix(&Matrix, &line, &column);

    graph = CreateGraph(Matrix, line, column);
```

```

    ColoringGraph(graph);

    free(Matrix);

    return 0;
}

```

Graph.h

```

#ifndef GRAPH_H
#define GRAPH_H

#include <stdio.h>
#include <stdlib.h>

typedef struct Graph
{
    struct Node *Head;
    int number;
} Graph;

typedef struct Node
{
    struct Node *next;
    struct Node *parent;
    struct HashT *Contact;
    int index;
    int color;
    int number;
    int status;
} Node;

typedef struct HashT
{
    struct Node *node;
    struct HashT *next;
    struct HashT *parent;
} HashT;

void GetMatrix(int **Matrix, int *line, int *column);
int GetIndex(int line, int i, int j);
void PrintMatrix(int *Matrix, int line, int column);
Graph *CreateGraph(int *Matrix, int line, int column);
Node *CreateNode(int index, int num_contact);
HashT *CreateNodeHashT();
Graph *DeleteNodeSave(Graph *graph, Node *node);
void DeleteNodeList(Node *Head, Node *node);
Graph *RestoringNode(Graph *graph, Node *node);
void RestoringContact(Node *node);
HashT *SortListNode(HashT *NodeCont);

```

```

void PrintInfoGraph(Graph *graph);
void RebootGraph(Graph *graph);
void GraphImageCreation(Graph *graph);

#endif

```

Graph.c

```

#include "Graph.h"

void GetMatrix(int **Matrix, int *line, int *column)
{
    FILE *stream = fopen("Matrix.txt", "r");
    if (!stream) {
        return;
    }

    char *str = NULL;
    size_t len = 0;
    int count, index;

    while (getline(&str, &len, stream) != -1) {
        count = 0;
        for (int i = 0; str[i] != 0; i++) {
            if (str[i] == ' ') {
                count++;
            }
        }
        count++;

        if (count > *column) {
            *column = count;
        }
        (*line)++;
    }

    fseek(stream, 0, SEEK_SET);

    *Matrix = malloc(sizeof(int) * *line * *column);
    for (int i = 0; i < *line; i++) {
        for (int j = 0; j < *column; j++) {
            index = GetIndex(*line, i, j);
            fscanf(stream, "%d", &(*Matrix)[index]);
        }
    }

    free(str);
    fclose(stream);
}

int GetIndex(int line, int i, int j) { return line * i + j; }

```



```

void PrintMatrix(int *Matrix, int line, int column)
{
    int index;
    for (int i = 0; i < line; i++) {
        for (int j = 0; j < column; j++) {
            index = GetIndex(line, i, j);
            printf("%d ", Matrix[index]);
        }
        printf("\n");
    }
}

Graph *CreateGraph(int *Matrix, int line, int column)
{
    Graph *graph = malloc(sizeof(Graph));
    Node **ArrNode = malloc(sizeof(Node) * line);
    Node *parent = NULL;

    graph->number = line;

    for (int i = 0; i < line; i++) {
        int num_contact = 0;

        for (int j = 0; j < column; j++) {
            if (Matrix[GetIndex(line, i, j)] == 1)
                num_contact++;
        }

        ArrNode[i] = CreateNode(i, num_contact);

        if (i != 0)
            parent->next = ArrNode[i];

        ArrNode[i]->parent = parent;

        parent = ArrNode[i];
    }

    graph->Head = ArrNode[0];

    Node *node = graph->Head;
    for (int i = 0; i < line; i++) {
        HashT *parentCont = NULL;

        for (int j = 0; j < column; j++) {

            if (Matrix[GetIndex(line, i, j)] == 1) {
                HashT *contact = CreateNodeHashT();

                contact->node = ArrNode[j];
                contact->next = NULL;
            }
        }
    }
}

```

```

        contact->parent = parentCont;

        if (parentCont != NULL)
            parentCont->next = contact;

        parentCont = contact;

        if (node->Contact == NULL) {
            node->Contact = contact;
        }

        contact = contact->next;
    }

    node = node->next;
}

free(ArrNode);

return graph;
}

Node *CreateNode(int index, int num_contact)
{
    Node *node = malloc(sizeof(Node));
    node->next = NULL;
    node->parent = NULL;
    node->Contact = NULL;
    node->index = index;
    node->color = -1;
    node->number = num_contact;
    node->status = 0;

    return node;
}

HashT *CreateNodeHashT()
{
    HashT *Contact = malloc(sizeof(Contact));
    Contact->node = NULL;
    Contact->next = NULL;
    Contact->parent = NULL;

    return Contact;
}

Graph *DeleteNodeSave(Graph *graph, Node *node)
{
    if (node->parent != NULL) {
        node->parent->next = node->next;
    } else {

```

```

    graph->Head = node->next;
}

if (node->next != NULL)
    node->next->parent = node->parent;

graph->number--;

DeleteNodeList(graph->Head, node);

return graph;
}

void DeleteNodeList(Node *Head, Node *node)
{
    Node *head = Head;
    while (head != NULL) {
        HashT *contact = head->Contact;

        for (int i = 0; i < head->number && contact->node->index != node->index; i++) {
            contact = contact->next;
        }

        if (contact != NULL && contact->node->index == node->index) {
            if (contact->parent != NULL) {
                contact->parent->next = contact->next;
            } else {
                head->Contact = contact->next;
            }

            if (contact->next != NULL)
                contact->next->parent = contact->parent;

            head->number--;

            contact->node = NULL;
            contact->next = NULL;
            contact->parent = NULL;
            free(contact);
        }

        head = head->next;
    }
}

Graph *RestoringNode(Graph *graph, Node *node)
{
    if (node->parent != NULL) {
        node->parent->next = node;
    } else {
        graph->Head = node;
    }
}

```

```

    if (node->next != NULL)
        node->next->parent = node;

    graph->number++;

    RestoringContact(node);

    return graph;
}

void RestoringContact(Node *node)
{
    HashT *NodeCont = node->Contact;

    while (NodeCont != NULL) {
        HashT *restCont = CreateNodeHashT();
        restCont->node = node;

        restCont->next = NodeCont->node->Contact;
        NodeCont->node->Contact = restCont;
        if (NodeCont->node->Contact->next != NULL)
            NodeCont->node->Contact->next->parent = restCont;

        NodeCont->node->number++;

        NodeCont = SortListNode(NodeCont);
        NodeCont = NodeCont->next;
    }
}

HashT *SortListNode(HashT *ListNode)
{
    HashT *FirstNode = ListNode->node->Contact;
    HashT *SecondNode = ListNode->node->Contact->next;

    while (SecondNode != NULL && SecondNode->next != NULL && FirstNode->node-
>index > SecondNode->node->index) {
        SecondNode = SecondNode->next;
    }

    if (FirstNode->next != NULL && FirstNode->node->index > FirstNode->next->node-
>index) {
        HashT *Buf = FirstNode;

        ListNode->node->Contact = FirstNode->next;
        ListNode->node->Contact->parent = NULL;

        if (SecondNode->parent != NULL && Buf->node->index < SecondNode->node-
>index) {
            SecondNode->parent->next = Buf;
            Buf->parent = SecondNode->parent;

```

```

    }

    if (Buf->node->index < SecondNode->node->index) {
        SecondNode->parent = Buf;
        Buf->next = SecondNode;
    } else if (Buf->node->index > SecondNode->node->index) {
        Buf->parent = SecondNode;
        Buf->next = SecondNode->next;
        SecondNode->next = Buf;
    }
}

return ListNode;
}

void PrintInfoGraph(Graph *graph)
{
    Node *node = graph->Head;

    printf("Numder node = %d\n", graph->number);
    printf("Index\tNode\t\tNext\t\tParent\t\tColor\t\tNumCont\t\tStatus\n");

    for (int i = 0; i < graph->number && node != NULL; i++) {
        printf("%d\t", node->index);
        printf("%p\t", node);
        printf("%p\t", node->next);
        if (node->next == NULL)
            printf("\t");

        printf("%p\t", node->parent);
        if (node->parent == NULL)
            printf("\t");

        printf("%d\t", node->color);
        printf("%d\t", node->number);
        printf("%d\n", node->status);

        if (node->Contact != NULL) {
            HashT *contact = node->Contact;

            for (int j = 0; j < node->number; j++) {
                printf(">ind: %d\t", contact->node->index);
                printf("color: %d\n", contact->node->color);
                contact = contact->next;
            }
        }

        node = node->next;
    }
}

void RebootGraph(Graph *graph)

```

```

{
    Node *node = graph->Head;

    while (node != NULL) {
        node->color = -1;
        node->status = 0;

        node = node->next;
    }
}

void GraphImageCreation(Graph *graph)
{
    FILE *out = fopen("graph.gv", "w");
    if (out == NULL) {
        return;
    }

    Node *node = graph->Head;

    char elem = "";
    char *colors[6] = {"red", "lawngreen", "deeppink", "cyan", "indigo", "purple"};

    fprintf(out, "digraph HelloGraph {\n");

    fprintf(out, "\tnode [shape=%ccircle%c, ", elem, elem);
    fprintf(out, "style=%cfilled%c, ", elem, elem);
    fprintf(out, "margin=%c0.01%c];\n", elem, elem);

    fprintf(out, "\tedge [dir=%cnone%c];\n\n", elem, elem);

    for (int i = 0; i < graph->number; i++) {
        int ncolor = node->color;
        fprintf(out, "\t%cbox%d%c [label=%cNode: %d\nColor: %d%c,",
                elem, i, elem, elem, i, ncolor, elem);
        fprintf(out, " fillcolor=%c%s%c];\n", elem, colors[ncolor], elem);
        node = node->next;
    }
    fprintf(out, "\n");

    node = graph->Head;
    for (int i = 0; i < graph->number; i++) {
        HashT *contact = node->Contact;

        for (int j = 0; j < node->number; j++) {
            int indCont = contact->node->index;

            if (i < indCont) {
                fprintf(out, "\t%cbox%d%c -> ", elem, i, elem);
                fprintf(out, "%cbox%d%c;\n", elem, indCont, elem);
            }
            contact = contact->next;
        }
    }
}

```

```

    }
    node = node->next;
}

fprintf(out, "}\n");

fclose(out);
}

```

GraphColoring.h

```

#ifndef GRAPHCOLORING_H
#define GRAPHCOLORING_H

#include "Graph.h"

void ColoringGraph(Graph *graph);
int TwoColor(Graph *graph);
int TwoColorRun(Node *node);
int TwoColorCheckNode(Node *node, int color);
int TFFColor(Graph *graph, int pow);
int TFFColorRun(Graph *graph, Node *node, int color);
int TFFColorNode(Node *node, int color);
int TFFColorCheckContact(Node *node, int numColor, int color);
int NColor(Graph *graph);
    int NColorRun(Node *node);
int NColorRunContact(Node *node);
int NColorContact(Node *node);
int NColorCheckContact(Node *node, int color);
#endif

```

GraphColoring.c

```

#include "GraphColoring.h"

void ColoringGraph(Graph *graph)
{
    int act;

    system("clear");

    while (act != 7) {
        printf("\n");
        printf("1) Раскрасить граф в 2 цвета\n");
        printf("2) Раскрасить граф в 3 или менее цвета\n");
        printf("3) Раскрасить граф в 4 или менее цвета\n");
        printf("4) Раскрасить граф в 5 или менее цветов\n");
        printf("5) Раскрасить граф\n");
        printf("6) Вывод информации о графе\n");
        printf("7) Выход\n");
    }
}

```

```

printf("Выберите действие: ");
scanf("%d", &act);

switch(act) {
    case 1:
        system("clear");
        RebootGraph(graph);
        if (TwoColor(graph) == 0) {
            GraphImageCreation(graph);
            system("dot -Tpng graph.gv -ograph.png");
        }
        break;
    case 2:
        system("clear");
        RebootGraph(graph);
        if (TFFColor(graph, 3) == 0) {
            GraphImageCreation(graph);
            system("dot -Tpng graph.gv -ograph.png");
        }
        break;
    case 3:
        system("clear");
        RebootGraph(graph);
        if (TFFColor(graph, 4) == 0) {
            GraphImageCreation(graph);
            system("dot -Tpng graph.gv -ograph.png");
        }
        break;
    case 4:
        system("clear");
        RebootGraph(graph);
        if (TFFColor(graph, 5) == 0) {
            GraphImageCreation(graph);
            system("dot -Tpng graph.gv -ograph.png");
        }
        break;
    case 5:
        system("clear");
        RebootGraph(graph);
        if (NColor(graph) == 0) {
            GraphImageCreation(graph);
            system("dot -Tpng graph.gv -ograph.png");
        }
        break;
    case 6:
        system("clear");
        PrintInfoGraph(graph);
        break;
    case 7:
        break;
    default:
        system("clear");

```



```

        printf("Ошибка действия\n\n");
    }
}

int TwoColor(Graph *graph)
{
    Node *node = graph->Head;

    node->color = 0;

    if (TwoColorRun(node) == 1) {
        printf("Error, 2 color\n");
        return 1;
    }
    return 0;
}

int TwoColorRun(Node *node)
{
    if (node->status == 1)
        return 0;

    int nextColor;
    if (node->color == 0) {
        nextColor = 1;
    } else if (node->color == 1) {
        nextColor = 0;
    }

    HashT *contact = node->Contact;
    for (int i = 0; i < node->number; i++) {
        int act = TwoColorCheckNode(contact->node, node->color);
        if (act == 1) {
            return 1;
        } else if (act == 0) {
            contact->node->color = nextColor;
        }
        contact = contact->next;
    }

    node->status = 1;

    contact = node->Contact;
    for (int i = 0; i < node->number; i++) {
        if (TwoColorRun(contact->node) == 1)
            return 1;
        contact = contact->next;
    }
    return 0;
}

```

```

int TwoColorCheckNode(Node *node, int color)
{
    if (node->color == color)
        return 1;

    if (node->color != -1)
        return 2;

    return 0;
}

int TFFColor(Graph *graph, int color)
{
    Node *node = graph->Head;

    if (node->next == NULL) {
        node->status = 1;
        node->color = 0;
        return 0;
    }

    if (TFFColorRun(graph, node, color) == 1)
        return 1;

    return 0;
}

int TFFColorRun(Graph *graph, Node *node, int color)
{
    int check = 0;
    for (int i = 0; i < graph->number; i++) {
        if (node->number < color && node->status != 1) {
            check = 0;

            graph = DeleteNodeSave(graph, node);

            TFFColor(graph, color);

            graph = RestoringNode(graph, node);

            if (TFFColorCheckContact(node, color, 0) == 1)
                return 1;
        } else {
            check++;
        }
        node = node->next;
    }

    if (check == graph->number) {
        if (TFFColorNode(graph->Head, color) == 1)
            return 1;
    }
}

```

```

    return 0;
}

int TFFColorNode(Node *node, int color)
{
    if (node->status == 1)
        return 0;

    if (TFFColorCheckContact(node, color, 0) == 1)
        return 1;

    HashT *contact = node->Contact;
    for (int i = 0; i < node->number; i++) {
        if (TFFColorNode(contact->node, color) == 1)
            return 1;
        contact = contact->next;
    }
    return 0;
}

int TFFColorCheckContact(Node *node, int numColor, int color)
{
    HashT *contact = node->Contact;
    for (int i = 0; i < node->number; i++) {
        if (contact->node->color == color) {
            if (TFFColorCheckContact(node, numColor, color + 1) == 1)
                return 1;
        }
        contact = contact->next;
    }

    if (numColor <= color) {
        printf("Error, %d color\n", numColor);
        return 1;
    }

    if (node->color == -1) {
        node->color = color;
        node->status = 1;
    }
    return 0;
}

int NColor(Graph *graph)
{
    Node *node = graph->Head;

    NColorRun(node);
    return 0;
}

```

```

int NColorRun(Node *node)
{
    if (node->status == 1)
        return 0;

    if (node->color == -1) {
        int color = NColorCheckContact(node, 0);
        node->color = color;
    }

    NColorRunContact(node);

    node->status = 1;

    HashT *contact = node->Contact;
    for (int i = 0; i < node->number; i++) {
        NColorRun(contact->node);
        contact = contact->next;
    }
    return 0;
}

int NColorRunContact(Node *node)
{
    HashT *contact = node->Contact;
    for (int i = 0; i < node->number; i++) {
        NColorContact(contact->node);
        contact = contact->next;
    }
    return 0;
}

int NColorContact(Node *node)
{
    if (node->color != -1)
        return 0;
    int color = NColorCheckContact(node, 0);
    node->color = color;
    return 0;
}

int NColorCheckContact(Node *node, int color)
{
    HashT *contact = node->Contact;
    for (int i = 0; i < node->number; i++) {
        if (color == contact->node->color)
            color = NColorCheckContact(node, color + 1);
        contact = contact->next;
    }
    return color;
}

```