МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой №\_\_\_

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

БАКАЛАВРСКАЯ РАБОТА

|  |  |
| --- | --- |
| на тему | Практическое использование алгоритмов раскраски планарных графов и |
| вопросы их реализации | |
|  | |

|  |  |
| --- | --- |
| выполнена | Тимановской Татьяной Сергеевной |
| фамилия, имя, отчество студента в творительном падеже | |

|  |  |  |  |
| --- | --- | --- | --- |
| по направлению подготовки | 01.03.02 |  | Прикладная математика и информатика |
|  | код |  | наименование направления |
|  | | | |
| наименование направления | | | |
| направленности |  |  |  |
|  | код |  | наименование направленности |
|  | | | |
| наименование направленности | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент группы № | 4518 |  |  |  | Т.С.Тимановская |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Руководитель

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| проф. |  |  |  | С.Д.Шапорев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2019

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

УТВЕРЖДАЮ

Заведующий кафедрой №\_\_\_

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

ЗАДАНИЕ НА ВЫПОЛНЕНИЕ БАКАЛАВРСКОЙ РАБОТЫ

|  |  |  |  |
| --- | --- | --- | --- |
| студенту группы № | 4518 |  | Тимановской Татьяне Сергеевне |
|  |  |  | фамилия, имя, отчество |

|  |  |
| --- | --- |
| на тему | Практическое использование алгоритмов раскраски планарных графов и |
| вопросы их реализации | |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| утвержденную приказом ГУАП от |  | № |  |

|  |  |
| --- | --- |
| Цель работы: | изучение алгоритмов раскраски графов |
| сравнительный анализ алгоритмов раскраски графов | |
| описание практического применения алгоритмов раскраски графов | |

|  |  |
| --- | --- |
| Задачи, подлежащие решению: | 1.Реализовать алгоритмы раскраски графов в среде |
| Microsoft Visual Studio 2017; 2. Провести сравнительный анализ алгоритмов. | |
|  | |

|  |  |
| --- | --- |
| Содержание работы (основные разделы): | Введение; Основные понятия теории |
| графов; Раскраска графов; Алгоритмы раскраски графов, Практическое применение | |
| раскраски графов, Заключение | |
|  | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Срок сдачи работы « |  | » |  | 201 | 9 |

Руководитель

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| проф. |  |  |  | С.Д. Шапорев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

Задание принял к исполнению

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| студент группы № | 4518 |  |  |  | Т.С.Тимановская |
|  |  |  | подпись, дата |  | инициалы, фамилия |

# Реферат

Пояснительная записка к бакалаврской работе содержит 66 страниц, 15 рисунков, 3 таблицы, 1 приложение. Использовано 30 источников.

Ключевые слова: алгоритмы раскраски графов, практическое применение раскраски графов.

Объектом проектирования являлись алгоритмы раскраски планарных графов, разработанные в среде Microsoft Visual Studio, на языке программирования С++.

Целью работы является систематизация информации о данной области знаний, программная реализация алгоритмов раскраски графов, сравнительный анализ полученных результатов.

В результате выполнения бакалаврской работы:

* была найдена и систематизирована имеющаяся информации о раскраске графов, как численные характеристики, так и алгоритмизация.
* была описана область практического применения данных алгоритмов в целом, и подробнее разобрана область применения, которая близка к моей специальности.
* были реализованы три алгоритма раскраски графов в среде Microsoft Visual Studio 2017.
* был проведен сравнительный анализ полученных результатов.

Оглавление

[ВВЕДЕНИЕ 5](#_Toc10651928)

[ГЛАВА 1 ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ ГРАФОВ 7](#_Toc10651929)

[1.1 ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ 7](#_Toc10651930)

[1.2 СПОСОБЫ ЗАДАНИЯ ГРАФА 10](#_Toc10651931)

[ГЛАВА 2 РАСКРАСКА ГРАФОВ 13](#_Toc10651932)

[2.1 ХРОМАТИЧЕСКОЕ ЧИСЛО 13](#_Toc10651933)

[2.1.1 ОЦЕНКИ ХРОМАТИЧЕСКОГО ЧИСЛА 13](#_Toc10651934)

[2.2 ТЕОРЕМА БРУКСА 14](#_Toc10651935)

[2.2.1 ДОКАЗАТЕЛЬСТВО ТЕОРЕМЫ БРУКСА МЕТОДОМ ЧЕРЕДУЮЩИХСЯ ЦЕПЕЙ 15](#_Toc10651936)

[2.3 ТЕОРЕМА О ПЯТИ КРАСКАХ 18](#_Toc10651937)

[2.4 РАСКРАСКА РЕБЕР 20](#_Toc10651938)

[2.5 ХРОМАТИЧЕСКИЙ МНОГОЧЛЕН 25](#_Toc10651939)

[ГЛАВА 3 АЛГОРИТМЫ РАСКРАСКИ ГРАФОВ 28](#_Toc10651940)

[3.1 ПОСЛЕДОВАТЕЛЬНЫЙ МЕТОД, ОСНОВАННЫЙ НА УПОРЯДОЧИВАНИИ МНОЖЕСТВА ВЕРШИН 28](#_Toc10651941)

[3.2 АЛГОРИТМ ВЕЙСМАНА 29](#_Toc10651942)

[3.3 АЛГОРИТМ ПОСЛЕДОВАТЕЛЬНОЙ РАСКРАСКИ (ЖАДНЫЙ АЛГОРИТМ) 32](#_Toc10651943)

[3.4 АЛГОРИТМ ЛОРЬЕРА-НОВИКОВА 32](#_Toc10651944)

[3.5 RLF(RECURSIVE-LARGEST-FIRST) АЛГОРИТМ 34](#_Toc10651945)

[3.6 BSC (BACKTRACKING SEQUENTIAL COLORING) АЛГОРИТМ 35](#_Toc10651946)

[3.7 АЛГОРИТМ ВИНДЕРСОНА ДЛЯ *K*-РАСКРАШИВАЕМОГО ГРАФА 37](#_Toc10651947)

[3.8 СРАВНЕНИЕ АЛГОРИТМОВ RLF,BSC И АЛГОРИТМА ЖАДНОЙ РАСКРАСКИ 38](#_Toc10651948)

[ГЛАВА 4 ПРИМЕНЕНИЕ РАСКРАСКИ ГРАФОВ 42](#_Toc10651949)

[4.1 ТЕХНОЛОГИЯ ЦИФРОВЫХ ВОДЯНЫХ ЗНАКОВ 44](#_Toc10651950)

[ЗАКЛЮЧЕНИЕ 48](#_Toc10651951)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 49](#_Toc10651952)

[ПРИЛОЖЕНИЕ 1 52](#_Toc10651953)

# ВВЕДЕНИЕ

Теория графов развивалась дедуктивно, изобретались новые вид графов и алгоритм их обработки, но до середины двадцатого века она оставалась исключительно теорией, не воплощенной в реальной действительности. Позднее, в результате распространения системного подхода к исследованию, графы стали широко применяться в качестве структурных моделей систем, и теория графов получила практическое применение. Модели в виде графов имеют многочисленные применения, например, модели коммуникационных и электрических сетей, модели комплексов работ и другие.

Возникновение проблемы раскраски графов было связано с необходимостью решения задачи раскраски карт. Как алгоритмическая проблема раскраска графов стала изучаться с конца двадцатого века, примерно в то же время были разработаны разнообразные алгоритмы раскраски графов.

В данной работе рассматриваются алгоритмы раскраски, и их применение в реальной жизни. Актуальными проблемами, решаемой с помощью раскраски графов, является проблема сопоставления с оригиналом, проблема эффективного планирования. Таким образом, раскраска графов находит свое применение как в области защиты интеллектуальной собственности программного обеспечения, так и экономических и технических областях знаний.

Объект исследования дипломной работы – алгоритмы раскраски планарных графов и их практическое применение.

Предмет исследования – характеристики эффективности алгоритмов, области применения алгоритмов раскраски графов.

Целью данной работы является программная реализация и сравнительный анализ алгоритмов раскраски графов, определение области их практического применения. Для достижения данной цели необходимо выполнить следующие задачи:

1. Рассмотреть основные понятия теории графов

2. Рассмотреть основные понятия и теоремы раскраски графов

3. Рассмотреть алгоритмы раскраски графов

4. Реализовать алгоритмы в программе.

5. Провести сравнительный анализ реализованных алгоритмов

6. Описать практические способы применения раскраски графов

# ГЛАВА 1 ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ ГРАФОВ

## 1.1 ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Основным понятием теории является граф. Пусть *S* - непустое множество, - множество всех его двухэлементных подмножеств, . Тогда пара, *(S,U)*  называется неориентированным графом. Элементы множества *S* называются вершинами графа, а элементы множества *U* - ребрами. Итак, *G=(S,U)* граф - это конечное множество вершин *S (*)и множество ребер *U (*. [1]

В неориентированном графе каждое ребро *u* имеет два конца, порядок которых не имеет значения. Ребро *u* называется петлёй, если начало и конец *u* совпадают. Рёбра *u* и *u*' называются кратными, если множества их концов совпадают. [2]

Если в паре вершин и указано направление связи, т.е. какая из вершин является первой, то соединяющий их отрезок называется дугой, а вершины, определяющие дугу называются концевыми вершинами. Если концевые вершины совпадают то дугу называют петлёй. В графе G могут существовать дуги с одинаковыми концевыми вершинами. Такие дуги называются параллельными.

Если в графе *G=(S,U)* все элементы множества U изображаются дугами, то граф называется ориентированным (рисунок 1) или орграфом, если ребрами, то неориентированным (рисунок 2). [1]

Рисунок 1 – полный неориентированный граф порядка 4

Рисунок 2 – ориентированный граф

В данной работе будут рассматриваться графы, не имеющие петель и кратных ребер (дуг).

Запись *u=xy* означает, что *x* и *y* – вершины, соединённые ребром *u*, такие вершины называются смежными. Ребра, имеющие общий конец так же называются смежными. Если *x* – конец ребра *u*, то *x* и *u* инцидентны.

**Определение 1.1.** 1) Для любой вершины через обозначается окрестность вершины *v* – множество всех вершин графа *G*, смежных с *v*. [2]

**Определение 1.2.** 1) Для вершины через обозначается степень вершины *v* в графе *G*, то есть количество рёбер графа *G*, инцидентных *v*.

2) Минимальная степень вершины графа *G* обозначается .

3) Максимальная степень вершины графа *G* обозначается [2]

**Определение 1.3.** 1) Граф *H* является подграфом графа *G*, если и

**Определение 1.4.** 1) Вершины *a* и *b* графа *G* называются связанными, если в графе существует пусть между ними.

2) Граф называется связным, если любые две его вершины связаны.

3) Очевидно, связность вершин – отношение эквивалентности, и все вершины графа по этому отношению разбиваются на классы эквивалентности – множества попарно связанных вершин. Эти классы эквивалентности называются компонентами связности графа.

4) Компонентами графа *G* называются подграфы, индуцированные на его компонентах связности. [1]

**Определение 1.5.** 1) Внутренне устойчивым множеством вершин графа *G* называется множество вершин *S*, все вершины которого попарно несмежны. Число внутренней устойчивости определяется как максимальная мощность такого множества *S* в графе *G*.

**Определение 1.6** 1) Граф *G* называется полным, если любые две вершины этого графа являются смежными. Такой граф обозначается , где n – число вершин.

В данной работе мы будет говорить преимущественно он неполных графах, если обратное не будет указано.

**Определение 1.7** 1) Кликой неориентированного графа называется подмножество его вершин, любые две из которых соединены ребром.

**Определение 1.8** 1) Гамильтонов пусть в графе *G* – это простой путь, проходящий по каждой вершине графа.

2) Гамильтонов цикл в графе *G* – это просто цикл, проходящий по каждой вершине графа.

3) Граф называется гамильтоновым, если в нем есть гамильтонов цикл.

**Определение 1.9.** 1) Раскраской вершин графа *G* в *k* цветов называется функция , где . Раскраска называется правильной, если для любой пары смежных вершин *u* и *v*.

2) Граф называется двудольным, если его вершины можно правильно раскрасить в два цвета.

Плоским графом называется граф, вершины которого являются точками плоскости, а ребра – непрерывными плоскими линиями без самопересечений, причем никакие 2 ребра не имеют общих точек, кроме инцидентной им обоим вершины. Любой граф, изоморфный плоскому графу, называется планарным (рисунок 3).

Г31

Г41

Г21

G

Г1

Г51

Рисунок 3 – Планарный граф *G* и его плоская укладка

На рисунке 3 граф имеет 5 граней: Г1, Г2 … Г5. Неограниченная грань Г1 называется внешней, остальные грани – внутренними.

## 1.2 СПОСОБЫ ЗАДАНИЯ ГРАФА

[1] В подавляющем большинстве случаев граф задается матрицей. Для расчетов на ЭВМ – это единственный способ. Существует редко применяемый сейчас метод задания графа в виде латинской матрицы. В этом способе направление дуг задается порядком букв в их названии. Например, для графа, изображенного на рисунке 4 латинская матрица будет выглядеть следующим образом:

u6

u7

u5

u4

u3

u2

u1

E

D

C

B

A

Рисунок 4 – ориентированный граф с вершинами, обозначенными латинскими буквами

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A |  | AB | AC | AD |  |
| B |  |  |  |  |  |
| C |  |  |  | CD | CD |
| D |  | DB |  |  |  |
| E |  |  |  | ED |  |

Таблица 1 – латинская матрица графа

Если граф неориентированный, то в такой латинской матрице штрихуют соответствующую клетку таблицы.

Наиболее часто граф задают с помощью матриц смежности и инциденций. Матрица смежности вершин для графа, изображенного на рисунке 4, будет выглядеть:

A B C D E

A

B

C

D

E

Это квадратная матрица , где *n* – число вершин. Ее строки и столбцы соответствуют вершинам графа. Элементы матрицы смежности равны числу дуг, идущих из *i*-той вершины в *j*-тую вершину. Если орграф не содержит параллельных дуг, то матрица является бинарной и состоит только из нулей и единиц. В случаем неориентированного графа матрица смежности вершин будет симметрична.

Аналогично можно определить матрицу смежности дуг. Это также квадратная матрица , где *m* – число дуг. Элементы равны единице, если дуга непосредственно предшествует дуге и равны нулю в остальных случаях. Для графа, изображенного на рисунке 4, матрица смежности дуг будет выглядеть:

u1 u2 u3 u4 u5 u6 u7

u1u2

u3

u4

u5

u6

u7

Матрица инциденций – это прямоугольная матрица размерности , где *n* – число вершин, а *m* – число дуг. Элементы этой матрицы равны плюс единице, если дуга исходит из *i*-той вершины (начальная вершина), минус единице, если дуга входит в *i*-тую вершину (конечная вершина), нулю, если дуга не инцидентна *i*-той вершине. Для графа, изображенного на рисунке 4, матрица инциденций будет выглядеть следующим образом:

u1 u2 u3 u4 u5 u6 u7

A

B

C

D

E

И матрицы смежности, и матрица инциденций однозначно определяют структуру графа.

# ГЛАВА 2 РАСКРАСКА ГРАФОВ

## 2.1 ХРОМАТИЧЕСКОЕ ЧИСЛО

**Определение 2.1.** Через обозначим хроматическое число графа *G* – наименьшее натуральное число, для которого существует правильная раскраска вершин графа *G* в такое количество цветов.

Приведем доказательства двух лемм, содержащих свойства хроматического числа. [2]

**Лемма 2.1.** Пусть граф *G* таков, что для любого его подграфа *H* выполняется . Тогда .

**Доказательство.** По индукции докажем, что вершины любого подграфа *H* графа *G* можно правильным образом раскрасить в *k* цветов. База для подграфа из одной вершины очевидна.

Рассмотрим подграф *H*, пусть – вершина наименьшей степени. По индукционному предположению мы можем раскрасить правильным образом в *k* вершины графа *H – v*, остается лишь заметить, что

1, поэтому вершину *v* можно докрасить, не нарушая правильности покраски.

**Лемма 2.2.** Для любого графа G выполняется неравенство:

*, где –* число вершин

**Доказательство.** Утверждение очевидно следует из соображения о том, что все вершины одного цвета в правильной раскраске попарно несмежны, то есть, образуют независимое множество.

Не существует точной формулы для подсчета хроматического числа графа, однако есть его пределы можно оценить как сверху, так и снизу. Приведем некоторые формулы оценки хроматического числа.

### **2.1.1 ОЦЕНКИ ХРОМАТИЧЕСКОГО ЧИСЛА**

Поскольку число равно мощности наибольшего множества попарно несмежных вершин графа *G*, то оно совпадает также с мощностью наибольшего множества вершин графа *G*, которые могут быть раскрашены в один цвет, и, следовательно,

,

где – целая часть числа *x*.

Для любого связного *(n,m)*-графа *G* верны неравенства[1]:

**Лемма 2.3.** Пусть *G(V,E)* – произвольный связный ориентированный граф с *m* ребрами, тогда,

**Доказательство.** Пусть – множества вершин окрашенных в соответствующие цвета при правильной раскраске графа *G*. Каждое из – независимое множество, так как раскраска правильная. Заметим, что между любыми двумя различными множествами существует хотя бы одно ребро. Тогда,

## 2.2 ТЕОРЕМА БРУКСА

**Теорема 2.1.** Пусть , а *G* – связный граф, отличный от , . Тогда .

**Замечание 2.1.** При вопрос о существовании правильной раскраски вершин связного графа *G* в два цвета очевиден. Такой граф *G* – либо (пусть из *n* вершин), либо(цикл из n вершин). В первом случае очевидно, что , во втором и .

**Лемма 2.4.** Пусть *G* – связный граф, , причем хотя бы одна из вершин графа *G* имеет степень менее *d*. Тогда .

**Доказательство.** Индукция по количеству вершин. База для графа, у которого не более *d* вершин, очевидна. Будем считать, что утверждение верно для любого связного графа с меньшим чем количеством вершин.

Пусть – вершина степени менее *d*. Рассмотрим граф *G* – *u*. Пусть , …, – компоненты графа *G* – *u*. В каждом из графов , …, ввиду связности графа *G* обязательно есть вершина, смежная в графе *G с u*, и эта вершина имеет в степень меньше *d*. Так как , по индукционному предположению существует правильная раскраска вершин графа *в d* цветов. Значит, существует правильная раскраска вершин в *d* цветов и у графа *G* – *u*. Так как , мы можем докрасить в один из цветов и вершину *u*, не нарушая правильности раскраски графа.

### **2.2.1 ДОКАЗАТЕЛЬСТВО ТЕОРЕМЫ БРУКСА МЕТОДОМ ЧЕРЕДУЮЩИХСЯ ЦЕПЕЙ**

[2] Ввиду леммы 2.4 остается только доказать теорему Брукса в случае, когда все вершины имеют степень *d*. Выберем вершину и рассмотрим правильную раскраску вершин графа *G – a* в *d* цветов (такая раскраска существует согласно лемме 2.4). Интересен лишь случай, когда *d* вершин окрестности раскрашены по одной в цвета 1, 2, …, *d*: иначе найдется цвет, в который можно покрасить вершину *a*. Пусть , причем .

#### **2.2.1.1. Построение чередующейся цепи**

Пусть . Пусть – подграф *G*, индуцированный на множестве вершин степеней *i* и *j*. Рассмотрим компоненту связности графа , содержащую вершину . Если поменять цвета всех вершин в (цвет i на цвет j и наоборот), то раскраска останется правильной. Значит : иначе после смены цветов в получается раскраска в которой обе вершины и имеют цвет *j* и новых вершин цвета *i* в не добавилось, тогда можно покрасить *a* в цвет *i* и получить правильную раскраску графа *G*. Таким образом, вершины и соединены в графе *G* простым путем, на котором чередуются цвета *i* и *j*. Этот путь назовем чередующейся *ij*-цепью.

#### **2.2.1.2. Свойства чередующихся цепей**

(a). , все вершины в и покрашены в разные цвета.

Если скажем, для это не так, то ее можно перекрасить в цвет отличный от *i* и тем самым получить правильную раскраску графа *G*.

(b). Если *x* –внутренняя вершина чередующейся *ij*-цепи и , то , в – две вершины цвета *j*, и по одной вершине всех цветов кроме *i* и *j*.

Назовем внутреннюю вершину цепи, для которой это свойство не выполнено, плохой. Предположим, что такие вершины есть и рассмотрим из них ближайшую к – пусть *x* как раз такая вершина и . Тогда в нет вершин какого-то цвета, отличного от *i* – скажем, цвета *l*. Перекрасим *x* в цвет *l*, а во всех вершинах цепи от (включительно) до *x* (не включительно) поменяем цвет *i* на цвет *j* и наоборот. По предыдущему пункту и так как от до *x* нет плохих вершин, мы получим правильную раскраску, в которой *в* нет вершин цвета *i*. Покрасим *a* в цвет *i* и получим правильную раскраску графа *G*.

(с). Граф – это и есть чередующаяся *ij*-цепь. Непосредственное следствие свойства (b).

(d). При перекраске чередующейся *ij*-цепи (операции, при которой вершины цвета *i* меняет цвет на *j* и наоборот) получается правильная раскраска графа *G – a*, в которой и . Непосредственное следствие свойства (с).

(e). Чередующиеся *ij* и *is*-цепи не могут иметь общих вершин кроме . Пусть – общая внутренняя вершина этих двух цепей. Тогда , вершина *b* не является концом ни одной из цепей и в есть по две вершины цветов *j* и *s*. Это противоречит свойству (b).

#### **2.2.1.3 Перекрашивание вершин**

Итак, для любых вершины и соединены чередующейся *ij*-цепью. В принципе, такая цепь может не иметь внутренних вершин, если и смежны. Однако, так как *G* – не полный граф на *d+1* вершине, в есть две несмежные вершины, пусть это и . Тогда 12-цепь нетривиальна, то есть .

Для “исправления” раскраски нам хватит трёх цветов. Рассмотрим 23-цепь и построим новую раскраску их перекраской 23-цепи. Из свойства (d) следует, что полученная раскраска будет правильной раскраской вершин графа *G* – *a*, причем и (Рисунок 5).

В раскраске также должна существовать чередующая 12-цепь. По свойству (e) ни одна из внутренних вершин 12-цепи, построенной для раскраски , не входит в 23-цепь для раскраски , следовательно, все эти вершины не перекрашены, то есть в раскраске начало 12-цепи так же будет . (Рисунок 5).

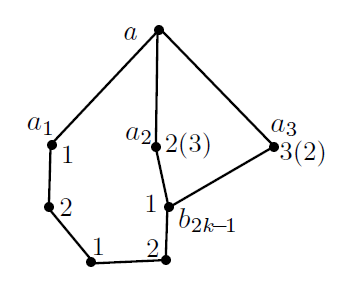


Рисунок 5 – Перекрашивание чередующейся цепи

В множестве в раскраске единственной вершиной цвета 2 была , но в новой раскраске эта вершина перекрашена в цвет 3. Значит, одна из вершин должна была изменить цвет 3 на цвет 2, то есть, это вершина входила в перекрашенную 23-цепь. Рассмотрим 2 случая.

Если , то чередующиеся 31 и 21-цепи раскраски имели общую внутреннюю вершину (Рисунок 5,в данном случае это единственная вершина цвета 1 в раскраске как в , так и в ), противоречие со свойством (е).

Пусть *x* – внутренняя вершина перекрашенной 23-цепи. 32-цепь в раскраске - это 23-цепь в раскраске , так как все ее вершины просто перекрасились (цвет 2 на 3 и наоборот). Тогда *x* является общей внутренней вершиной 32 и 12-цепей в раскраске (Рисунок 6), что так же невозможно.

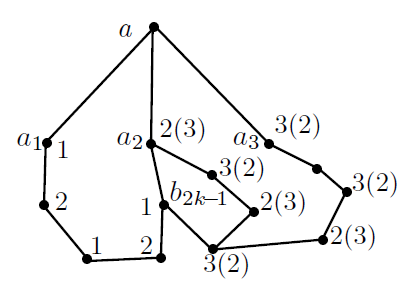


Рисунок 6 – Перекрашивание чередующейся цепи без общей вершины

Таким образом, в раскраске не существует чередующейся 12-цепи. Как показано выше, это означает возможность исправить раскраску и дополнить ее до правильной раскраски графа *G*.

## 2.3 ТЕОРЕМА О ПЯТИ КРАСКАХ

В 1878 году английский математик Кэли сформулировал гипотезу четырех красок.

**Гипотеза четырех красок.** Всякий планарный граф можно правильно раскрасить в четыре цвета.

Первым доказал ее в 1879 математик В.Кэмпе [20]. Однако доказательство оказалось ошибочным. Затем последовало множество других ошибочных доказательств. Существенно же положение с доказательством изменилось в середине XX века благодаря: идеи Дж. Д. Биркгофа позволили П. Франклину в 1913 году доказать гипотезу для планарного графа с не более чем 25 вершинами. Позже это число было увеличено до 38. [29]

На основе гипотезы о четырех красках П. Хивуд сформулировал и доказал теорему о пяти красках. [19]

**Теорема о пяти красках.** Всякий планарный граф 5-ти раскрашиваем

**Доказательство.** Будем доказывать индукцией по числу р вершин. Для любого планарного графа с *р*<5 вершинами результат тривиален, поскольку такой граф р-раскрашиваем.

Допустим, что все планарные графы с *р* вершинами (*р* >= 5) 5-раскрашиваемы. Пусть *G* — плоский граф *с р+1* вершинами. В силу следствия в графе *G* найдется вершина *v* степени 5 или менее. По предположению индукции плоский граф *G - v* 5-раскрашиваем.

Рассмотрим приписывание цветов вершинам графа *G — v*, при котором получается 5-раскраска; цвета будем обозначать через , . Ясно, что если некоторый цвет, скажем , не используется в раскраске вершин, смежных , то, приписав цвет вершине v, получим 5-раскраску графа *G*.

Осталось рассмотреть случай, когда и для вершин графа *G,* смежных с *v*, используются все пять цветов. Переставим номера цветов, если это необходимо, так, чтобы вершины, смежные с *v* и окрашенные в цвета , были циклически упорядочены (Рисунок 7). Пометим теперь вершину, смежную с *v* окрашенную цветом , буквой , .

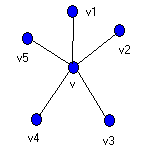


Рисунок 7 – Циклически упорядоченный граф

Обозначим через *G13* подграф графа *G – v*, порожденный всеми вершинами, окрашенными в один из цветов и . Если вершины и принадлежат различным компонентам графа *G13*, то 5-раскраску графа *G – v* можно получить, поменяв друг с другом ( на и обратно) цвета вершин той компоненты графа *G13*, которая содержит их. В этой 5-раскраске уже нет вершин, смежных с *v* и окрашенных в цвет ; поэтому, окрасив *v* в цвет , образуем 5-раскраску графа *G*. Если же вершины и принадлежат одной и той же компоненте графа *G13*, то в *G* между и существует простая цепь, все вершины которой окрашены в цвета и . Эта цепь вместе с цепью образует простой цикл, который обязательно окружает или вершину , или вершины и . В любом из этих случаев и нельзя соединить простой цепью, все вершины которой окрашены в цвета и . Следовательно, рассматривая подграф *G24* графа *G – v*, порожденный всеми вершинами, окрашенными в цвета и , заключаем, что вершины и принадлежат различным его компонентам. Таким образом, если поменять между собой цвета вершин в компоненте подграфа *G24*, содержащей , получим 5-раскраску графа *G – v*, и в ней ни одна из вершин, смежных с *v*, не будет окрашена в цвет . Поэтому, окрасив вершину *v* в цвет , образуем 5-раскраску всего графа *G* [21].

## 2.4 РАСКРАСКА РЕБЕР

**Определение 2.2.** 1) Раскраской ребер графа *G* в *k* цветов называется функция .

2) Любая раскраска ребер графа *G* в цвета [*1…k*] – это разбиение множества *E(G*) в объединение непересекающихся множеств , где принимает значение *i* на ребрах множества . [2]

Графы, рассматриваемые в этом разделе, могут иметь кратные ребра, но не могут иметь петель.

**Определение 2.3.** 1) Раскраска называется правильной, если для любой пары смежных ребер . – реберное хроматическое число или хроматический индекс графа *G* – наименьшее натуральное число, для которого существует правильная раскраска ребер графа *G* в такое количество цветов.

**Определение 2.4.** Пусть – раскраска ребер графа *G* в *k* цветов.

1) Пусть . Будем говорить, что в раскраске цвет *i* представлен в вершине *v*, если существует инцидентное *v* ребро *e* такое, что . Будем говорить, что цвет *i* представлен *t* раз в вершине *v*, если существует *t* инцидентных *v* ребер цвета *i*. Обозначим через количество цветов, представленных в вершине *v*.

2) Ведем обозначение Назовем раскраску *k*-оптимальной, если для любой другой раскраски ребер графа *G* в *k* цветов [2].

**Определение 2.5.** Обозначим как максимальную кратность ребра графа *G,* то есть, максимум для всех пар .

**Лемма 2.5.** Пусть – *k*-оптимальная раскраска ребер графа *G*. Предположим, что вершина *w* и цвета *i* и *j* таковы, что в вершине *w* хотя бы два раза представлен цвет *i* и не представлен цвет *j*. Пусть , а – компонента графа *H*, содержащая вершину *w*. Тогда – простой цикл нечетной длины.

**Доказательство.** Предположим, что не является простым циклом нечетной длины. Построим новую раскраску , отличающуюся от лишь раскраской ребер из : раскрасим их в цвета *i* и *j* так, чтобы в каждой вершине *x* степени были представлены оба цвета *i* и *j*. Тогда , а для любой другой вершины *x*, очевидно, . Таким образом, , противоречие *с k*-оптимальностью .

**Теорема Визинга для графов, допускающих кратные ребра.** Пусть граф *G* – граф без петель. Тогда . [28]

**Доказательство**. Пусть =. Рассмотрим -оптимальную раскраску ребер графа *G* цветов. Предположим, что раскраска – неправильная. Тогда существует вершина *u* и цвет , который дважды представлен в вершине *u*. Так как, , то существует цвет *j*, непредставленный в вершине *u*. Пусть – ребро цвета . Так как , существует цвет , не представленный в . Если не представлен в *u,* то, перекрасив ребро в , мы увеличим . Следовательно, цвет представлен в *u*, пусть – ребро цвета .

1. Шаг процесса построения.

Пусть различные цвета и ребра таковы, что , , а цвет не представлен в вершине (при всех ). Будем говорить, что цвет выбран для вершины . Вершины не обязательно различны. Рассмотрим вершину . Пусть в наборе вершина *v* встречается *m* раз. Понятно что . Тогда на предыдущих шагах мы рассматривали вершину *v* и *m-1* раз выбирали цвет, не представленный в этой вершине. Поскольку , то существует цвет , не представленный в вершине и не выбранный для нее на предыдущих шагах. Мы выберем именно этот цвет. Определим раскраску :

, при , на остальных ребрах *e*.

Докажем, что , то есть, – оптимальная раскраска ребер в цветов. Для вершины цвета инцидентных *x* ребер не менялись, поэтому .

Рассмотрим вершину *w*, которая входит в набор ровно *n* раз. Пусть . По построению, все выбранные для вершины *v* цвета различны, не представлены в вершине *w* в раскраске и представлены в раскраске . Все отличные от ребра, инцидентные *w*, не изменили свой цвет, и потому остальные цвета одинаково представлены в вершине *w* в раскрасках и . Поэтому .

Рассмотрим еще вершину *u*. В результате перекрашивания инцидентных *u* ребер из их цветов исчез и появился . Однако, цвет был представлен в вершине *u* в раскраске хотя бы дважды, поэтому он представлен и в раскраске . Таким образом, и , то есть раскраска оптимальна.

Более того, из оптимальности следует, что , следовательно, . Это означает, что цвет представлен в вершине *u* в раскраске . Пусть – ребро цвета .

1. Перекрашивание.

Поскольку у вершины *u* конечное число соседей, на некотором шаге построения мы впервые получим . По построению отсюда следует, что не совпадает с (иначе мы бы выбрали цвета и разными). Так как в вершинах и в раскраске не представлен цвет , а в – представлен, все три вершины , и различны.

Рассмотрим оптимальные раскраски и (мы положим ). В обеих раскраска в вершине *u* хотя бы дважды представлен цвет :

.

Пусть – множество всех ребер цвета s в раскраске , а – множество всех ребер цвета *s* в раскраске , и . На рисунке 8 изображены цвета ребер, соединяющих *u* с в раскрасках (рисунок 8a), (рисунок 8b), (рисунок 8c).

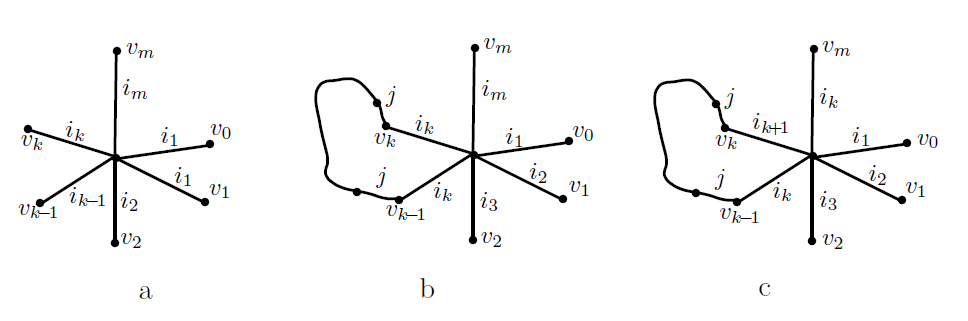


Рисунок 8 – Раскраски , ,

Применим лемму 2.5: из оптимальности наших раскрасок следует, что содержащие вершину *u* компоненты графов *H* и *H*’ – простые циклы нечетной длинны.

Тогда : из выходит ребро цвета и ребро цвета *j*. Для всех ребер *e* цикла *H*, кроме , мы имеем , поэтому . При этом, очевидно, и *u* лежат в одной компоненте графа *H*’, которая должна быть простым нечетным циклом. Противоречие.

Полученное противоречие показывает, что – искомая правильная раскраска ребер графа *G* в цветов.

Соответственно для графов не допускающих кратности ребер и .

Полное доказательство приведено в источнике [28].

Изучение раскраски ребер так же помогло уточнить описанную выше гипотезу о четырех красках. Данная теорема была определена Тэйтом и уточнена Таттом[30].

**Теорема 2.2.** Для того чтобы хроматическое число плоского графа *G* было равно четырем, необходимо и достаточно, чтобы его ребра допускали такую раскраску в три цвета, при которой никакие два ребра одной грани не были раскрашены одинаково.

Доказательство данной теоремы приведено в источнике [25].

## 2.5 ХРОМАТИЧЕСКИЙ МНОГОЧЛЕН

**Определение 2.6.** Для натурального числа *k* определим , как возможное количество правильных раскрасок графа *G* в *k* цветов. Функция – хроматический многочлен графа *G.* Очевидно, что и , .

Данное понятие относится к алгебраической теории графов. Первоначально, многочлен определил Джордж Дэйвид Биркгоф при попытке доказать упоминавшуюся ранее гипотезу о четырех красках. Позже, многочлен обобщил Х.Уитни, применив его к графа общего вида, а не только к планарным. У.Т.Татт вывел из него многочлен Татта, связав его со статистической физикой.

Для графа *G* с *n* вершинами хроматический многочлен определяется, как уникальный интерполирующий многочлен степени, не превосходящей *n*, проходящий через точки

Хроматическое число графа *G* является наименьшим положительным целым числом, при котором хроматический многочлен не обращается в ноль.

Ниже приведены хроматические многочлены для некоторых видов графов.

Таблица 2 – Хроматические многочлены

|  |  |
| --- | --- |
| Полный граф | *k(k-1)(k-2) … (k-(n-1))* |
| Путь | *k(* |
| Дерево с n вершинами | *k(* |
| Цикл | ( |
| Граф Барбелла (см. прим. ниже) |  |
| Пан граф (см. прим. ниже) |  |

Граф Барбелла – граф, состоящий из двух одинаковых полных графов , соединенных мостом (Рисунок 9).

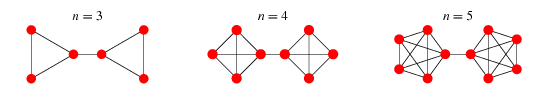


Рисунок 9 – Примеры графа Барбелла

Пан граф – граф, состоящий из цикла и единичного графа , соединенных мостом (Рисунок 10).

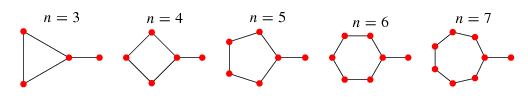


Рисунок 10 – Примеры пан графа

Два графа являются хроматически эквивалентными, если они имеют одинаковые хроматические многочлены. Изоморфные графы всегда хроматические эквивалентны, обратное утверждение неверно.

Очевидно, что если – все компоненты графа *G*, тогда

Для любого *G* число 0 – корень хроматического многочлена такой кратности, сколько компонент связности имеется в графе. Коэффициенты любого хроматического многочлена знакопеременны.

Известны также алгоритмы нахождения хроматического полинома.

Существуют эфективные алгоритмы, имеющие полиномиальное время вычисления хроматического многочлена, однако, они применимы только к некоторым видам графов: хордальные графы (цикл, имеющий хорду), графы с ограниченной кликовой шириной.

Рекурсивный способ вычисления хроматического многочлена базируется на операции стягивания ребра.

**Определение 2.6.** Для ребра через обозначается граф, полученный в результате стягивания ребра *e=xy*. То есть граф получается из графа *G-x-y* добавлением новой вершины *z*, которая будет смежна со всеми вершинами, смежными в исходном графе хотя бы с одной из вершин *x* и *y*.

Тогда хроматический многочлен удовлетворяет рекурсивному соотношению: , где *e=xy.*

Изложенные выше определения и методы относятся к алгебраической теории графов, подробно данная тема изложена в источнике [6].

# ГЛАВА 3 АЛГОРИТМЫ РАСКРАСКИ ГРАФОВ

Проблема определения хроматического числа графа относится к классу NP-полных задач. В базовом случаем нельзя вычислить хроматическое число графа оперируя только его стандартными числовыми характеристиками (число вершин, компонент связности и распределение степеней вершин. Как сказано выше, задача раскраски графа относится к классу *NP*-полных и не позволяет найти оптимальное решение (точное значение хроматического числа) за полиномиальное время (при условии *P≠NP*), поэтому на практике для ее решения применяются различные эвристические методы и алгоритмы [13].

На данный момент существую много алгоритмов раскраски графов. Они делятся на точные и приближенные. Некоторые из алгоритмов применяются только к графам определенного рода, например к 3-раскрашиваемым, данные алгоритмы подробно рассмотрены в источнике [15]. Ниже мы приведем несколько алгоритмов, которые позволяют раскрасить граф (определить его хроматическое число), они применятся к неполным планарным графам без петель и кратных ребер.

3.1 ПОСЛЕДОВАТЕЛЬНЫЙ МЕТОД, ОСНОВАННЫЙ НА УПОРЯДОЧИВАНИИ МНОЖЕСТВА ВЕРШИН

[12] В этом простейшем из методов вершины вначале располагаются в порядке невозрастания их степеней. Первая вершина окрашивается в цвет 1; затем список вершин просматривается сверху вниз (по невозрастанию степеней) и в цвет 1 окрашивается всякая вершина, которая несмежна с другой, уже окрашенной в этот цвет. Потом возвращаемся к первой в списке неокрашенной вершине, окрашиваем ее в цвет 2 и снова просматриваем список вершин сверху вниз, окрашивая в цвет 2 любую неокрашенную вершину, которая не соединена ребром с другой, уже окрашенной в цвет 2 вершиной. Аналогично действуем с цветами 3, 4 и т. д., пока не будут окрашены все вершины. Число использованных цветов будет тогда приближенным значением хроматического числа графа.

Алгоритм действий:

1. Упорядочить вершины по невозрастанию степени.

2. Окрасить первую вершину в цвет 1.

3. Выбрать цвет окраски 1.

4. Пока не окрашены все вершины, повторять п.4.1.-4.2.:

4.1. Окрасить в выбранный цвет всякую вершину, которая несмежна с другой, уже окрашенной в этот цвет.

4.2. Выбрать следующий цвет.

Приведенный выше алгоритм можно модифицировать. Для этого после каждого шага нужно упорядочивать неокрашенные вершины. Простая модификация описанной выше эвристической процедуры состоит в переупорядочивании неокрашенных вершин по невозрастанию их относительных степеней. Под относительными степенями понимаются степени соответствующих вершин в неокрашенном подграфе исходного графа. В данной модификации предполагалось, что если две вершины имеют одинаковые степени, то порядок таких вершин случаен.

Не учитывая время, затраченное на сортировку вершин в порядке невозрастания степеней, необходимо сделать цикл по всем вершинам графа. Для каждой необходимо найти минимальный цвет, что в худшем случае может занять *O()* для случая с полным графом. Значит, общее время составит *O()* в худшем случае.

## 3.2 АЛГОРИТМ ВЕЙСМАНА

[22] Данный алгоритм относится к точным алгоритмам раскраски графа.

Алгоритм состоит из двух частей:

1. Построение семейства максимальных внутренне устойчивых множеств (МВУМ) (метод Магу) (пункты 1-8 ниже).

2. Выбор минимального числа МВУМ, покрывающих все вершины графа (метод Петрика) (пункты 9-11 ниже).

Структура алгоритма:

1) В матрице инцидентности *R* для каждой вершины подсчитывается число ненулевых элементов .

2) Находится вершина c максимальным .

3) Для выбранной вершины записывается выражение , где – соседи .

4) Из матрицы *R* удаляется строка и столбец соответствующие вершине .

5) Если , то переходим к пункту 2, иначе к пункту 6.

6) Составляется конъюнкция . Раскрываются скобки и в полученной дизъюнкции по законам булевой алгебры выполняется минимизация.

7) Результат минимизации записывается в виде .

8) Для каждого строятся дополнения до всех вершин. Полученное семейство множеств

9) Для каждой вершины определяются подмножества , в которые входит вершина . Составляется дизъюнкция .

10) Составляется конъюнкция . Раскрываются скобки и выполняется минимизация булевой функции.

11) Получена дизъюнкция конъюнктивных термов . Выбирается конъюнктивный терм с минимальным числом сомножителей.

12) Количество сомножителей в полученном терме – хроматическое число графа, а каждое – множество вершин, которые можно окрасить в один цвет.

Псевдокод алгоритма Вейсмана.

*function Magu(R,X)* алгоритм Магу, *R* – матрица инцидентности

*{C,D} = SelectVertices(R,X)*

построение и минимизация булевой функции

построить дополнения для каждого элемента из

*return*

*end function*

*function SelectVertices(R,X)* выбрать вершины с максимальным . *R* – матрица инцидентности, *X* – вершины

*C=* выбранные вершины

*D=* соседи выбранных вершин

*while R do*

*Гx=GetNeighbours(R,X)*  построить соседей для текущих вершин

*C.Add(*)

*D.Add(*

*X.Remove()*

*R.Remove()* удалить выбранную строку и столбец

*end while*

*return С, D*

*end function*

*function Petrik(Ф,X)*  метод Петрика

*T=BuildT(Ф,X)*

построение и минимизация булевой функции

*z=GetMinTerm(*

*return z*

*end function*

*function BuildT(Ф,X)*

*T=*

*for all do*

*for all do*

*if then*

*end if*

*end for*

*T.Add()*

*end for*

*return T*

*end function*

*function Weisman(R,X)* алгоритм Вейсмана

*return Z*

*end function*

## 3.3 АЛГОРИТМ ПОСЛЕДОВАТЕЛЬНОЙ РАСКРАСКИ (ЖАДНЫЙ АЛГОРИТМ)

[23] Жадный алгоритм является простейшим методом поиска раскраски неориентированного графа, близкой к хроматическому числу. Жадный алгоритм состоит в последовательной раскраске всех вершин *X* графа *G* в минимально возможный цвет, при условии, что ни одна вершина, смежная данной, не раскрашена в этот цвет. Для задачи минимальной раскраски графа метод не гарантирует нахождение оптимального решения. Временная сложность метода линейна относительно количества вершин *(n)* и количества ребер *(m)* в графе – *O(n+m)*.

Жадный алгоритм основан по последовательном переборе. Вначале берется случайная вершина и окрашивается в цвет 1. Далее последовательно берутся все вершины и окрашиваются в минимально возможные для них цвета (эти минимально возможные цвета определяются в зависимости от того в какие цвета уже окрашены смежные им вершины).

Псевдокод жадного алгоритма.

Дан граф *G* с максимальной степенью вершин *∆*.

Результатом является правильная раскраска графа в количество цветов.

1. [Инициализация] *i=0*
2. [Следующая вершина] increment *i*. *if i=n+1*, *f* – конечная раскраска
3. [Поиск цветов вершин в окрестности текущей] *C*=список цветов уже использованных в окрестности текущей вершины.
4. [Определение наименьшего доступного для текущей вершины цвета] *for* (*c*=1;;*c*++) {*if (cC) {f()=c; break;*}}. переход на 2

## 3.4 АЛГОРИТМ ЛОРЬЕРА-НОВИКОВА

[24] Данный алгоритм так же относиться к точным алгоритмам. Для его формализации потребуется постановка задачи нахождения максимально независимых множеств.

Пусть задан граф *G(V,E).* Найти такое множество *X*, , что

*,* где

Основная идея состоит в построении рекурсивной процедуры P:

1. Выбрать в графе *G* некоторое максимально независимое множество вершин *S*.
2. Покрасить вершины множества *S* в очередной цвет.
3. Применить процедуру *P* к графу *G – S*

Псевдокод процедуры *P*.

*С* Номера цветов приписанные вершинам

*procedure P(G(V,E),i)* *i* – номер свободного цвета

*if V= then*

*return* раскраска закончена

*end if*

*S:=Selectmax(G)* *S* – максимально независимое множество

*C[S]:=i* раскрашиваем вершины множества *S* в цвет *i*

*P(G-S,i+1)* рекурсивный вызов процедуры

*end procedure*

Псевдокод процедуры *Selectmax* (построение максимально независимых множеств).

*procedure Selectmax* (граф *G(V,E)*, заданный списками смежности *Г[v]*)

*k:=0*  количество элементов в текущем независимом множестве

*S[k]:=* независимое множество из *k* вершин

множество вершин уже использованных для расширения *S[k]*

множество вершин, которые можно использовать для расширения *S[k]*

*M1:* шаг вперед

*select v* расширяющая вершина

*S[k+1]:=S[k]* расширенное множество

вершина *v* использована для расширения

] вершины, смежные с *v*, не могут быть использованы для расширения

*k:=k+1*

*M2:*  проверка

*for do*

*if = then*

*goto M3*

*end if*

*end for*

*if = then*

*if = then*

*yield S[k]* множество *S[k]* максимально

*end if*

*goto M3* можно возвращаться

*else*

*goto M1* можно идти вперед

*end if*

*M3:*

*v:=last(S[k])* последний добавленный элемент

*k:=k-1*

*S[k]:=S[k+1]-{v}*

вершина *v* уже добавлялась

*if k=0 & then*

*stop*  перебор завершен

*else*

*goto M2* переход на проверку

*end if*

*end procedure*

## 3.5 RLF(RECURSIVE-LARGEST-FIRST) АЛГОРИТМ

В данном алгоритме, *x* – фиксированная вершина максимальной степени. Несмежные вершины *y* с максимальным числом возможных вершин, смежных с *x* – переопределяются *x*, пока *x* не будет примыкать ко всем вершинам. Чтобы повысить эффективность, *x* затем удаляется и выбирается новая вершина с максимальной степенью в графе *G – x*. Вершина *x* и все вершины, переопределенные как *x* окрашиваются в один цвет (*color class*). И так пока все вершины не будут раскрашены (не будет определен *color class* для них). В худшем случае временная сложность RLF алгоритма .

Псевдокод RLF алгоритма. [3]

Дан граф *G=(V,E)* с множеством вершин *V=V(G)* и множеством ребер *E*.

Возвращается раскраска

*colornumber=0;* количество использованных цветов

*while (|V(G)|>0)* {

определяем вершину *x* с максимальной степенью в графе *G*;

*colornumber=colornumber+1;*

*F(x)=colornumber;*

*NN*=множество вершин несмежных с *x*;

*while (|NN|>0)* { ищем множество вершин *y* в *NN* которое можно переопределить как *x*

*maxcn = -1* максимальное число возможных соседей

*ydegree = -1* получаем степени *y*

*for every vertex z in NN*{

*cn*=количество возможных соседей *z* и *x*;

*if (cn>maxcn or(cn==maxcn and degree(z)<ydegree)){*

*y=z;*

*ydegree=degree(y);*

*maxcn=cn;*

*};*

*};*

*if (maxcn==0){* в этом случае граф несвязный

*y*=вершина с максимальной степенью в *NN*;

}

*F(y)=colornumber;*

переопределяем *y* как *x*;

обновляем список вершин несмежных с *x* *(NN);*

}

*G=G-x*; удаляем *x* из *G*

};

*return F*

## 3.6 BSC (BACKTRACKING SEQUENTIAL COLORING) АЛГОРИТМ

Алгоритм последовательной раскраски обратного обхода был впервые разработан Брауном[5] и в последствии улучшен Брельцем[4].

Вершины графа заданы массивом *A*. Изначально они сортированы в соответствии с неувелечением их степеней. Порядок динамически изменяется. Предположим *A[0], A[1], … A[i-1]* уже окрашены. Количество различных цветов этих вершин определяется как *colors(i-1)=*. Множество свободных цветов для *x=A[i]*, *U=freeColors(x)*, это подмножество цветов *{1, 2, … }*, которые не представлены в окрестности *x*. Если верхняя граница *optColorNumber*, определена для раскраски *F*, все остальные цвета, которые больше *optColorNumbers* могут быть удалены из *U*. Следующая раскрашиваемая вершина – вершина с максимальной степенью насыщения. Она окрашивается в наименьший цвет из *U*. Если *U* – пустое множество, обратный обход окончен. Временная сложность данного алгоритма .

Псевдокод BSC алгоритма. [3]

Поиск *,* сортированный в соответствии с неувеличением степеней вершин *a*;

*start=0* начальный индекс

*optColorNumber=|V|+1;*  максимальное количество цветов

*x=A[0]*; текущая вершина для окрашивания

*colors(-1)=0;* номер цвета использованного в *A* до этого

*U=[1];* переменная для списка свободных цветов

*freeColors(x)=U;* список свободных цветов

*while (start≥0){*

*back=false;* переменная типа *bool* для обратного обхода

*for (i=start;i<|V|;i++){*

*if (i>start){*

выбираем вершину *х* с макс степенью насыщения.

*U(x)*=множество свободных цветов < *optColorNumber*;

сортируем *U* по возрастанию;

*}*

*if (|U|>0){*

*k=U[0];* выбранный свободный цвет

*F(x)=k;* текущая раскраска

удаляем *k* из *U*;

*freeColors(x)=U;*

*l=colors(i-1);*

*colors(i)=max{k,l};*

*}*

*else{*  *U* – пустое множество, возврат на одну позицию

*start=i-1;*

*back=true;*

*break;* выход из *for*

*}*

*}*

*if (back){*

*if (start≥0){*

*x=A[start];* новая стартовая вершина

*uncolor x;*

*U=freeColors(x);*

*}*

*}*

*else {* в этом случае не было выхода из *for* через *break*

*Fopt=F;*

*optColorNumber = colors(|V|-1);*

*i*=последний индекс вершины с *F(A[i])=optColorNumber*;

*start=i-1;*

*if (start<0){ break;}* покидаем цикл *while*

*uncolor* все вершины *A[i],* где *i≥start*;

*for (i=0;i≤start; i++){*

*x=A[i];*

*U=freeColors(x);*

Удаляем из *U* все цвета ≥ *optColorNumber*;

*freeColors(x)=U;*

*}*

*}*

*}*

*return Fopt*

## 3.7 АЛГОРИТМ ВИНДЕРСОНА ДЛЯ *K*-РАСКРАШИВАЕМОГО ГРАФА

Стандартным методом приближенной раскраски *k*-раскрашиваемого графа является выделение вершины я максимальной степенью, и реккурсивные попытки раскрасить ее *(k-1)*-раскрашиваемую окрестность в минимально возможное количество цветов. Данный алгоритм позволяет раскрасить граф правильно, если известно его хроматическое число, то есть если определено *k*.

Описание алгоритма Виндерсона: [27]

Дан *k*-раскрашиваемый граф *G* с *n* вершинами

Результат: правильная раскраска графа в максимум цвет.

Шаг 1. Если существует вершина с по меньшей мере соседями, тогда раскрашиваем реккурсивно вершины окрестности в

цветов. Затем удаляем эти узлы из графа и цвета из палитры.

Важно, что этот шаг может быть использован не более количества раз.

Шаг 2. В другом случае, использует жадную раскраску оставшегося графа в цветов.

В итоге, общее количество цветов использованных на обоих шагах будет равно .

## 3.8 СРАВНЕНИЕ АЛГОРИТМОВ RLF,BSC И АЛГОРИТМА ЖАДНОЙ РАСКРАСКИ

Данные алгоритмы были реализованы мной программно в пакете Microsoft Visual Studio 2017. Программный код написан на языке C++. Программа не имеет пользовательского интерфейса. Во всех случаях граф задается матрицей смежности вершин. Для наглядности алгоритмы были применены к пяти планарным графам из 15, 26, 35, 42, 50 вершин соответственно (Рисунки 11-15). Все результаты работы программы выводятся на консоль. Сравниваются алгоритмы по таким парамерам, как быстродействие, точность оценки хроматического числа графа.

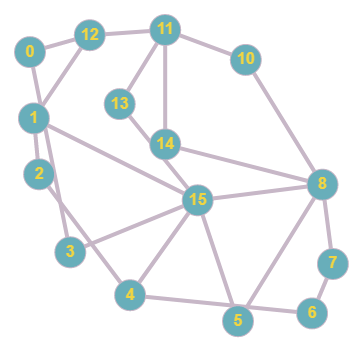


Рисунок 11 – Планарный граф на 15 вершинах

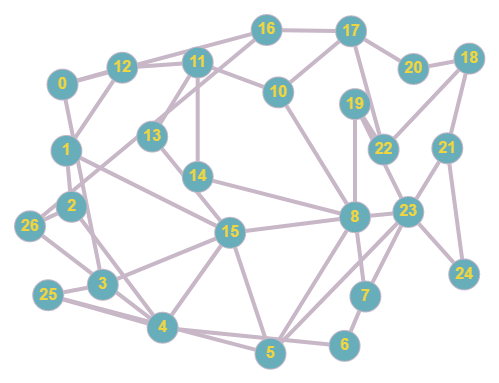


Рисунок 12 – Планарный граф на 26 вершинах

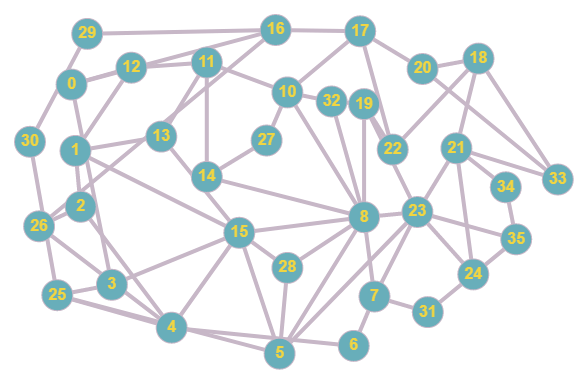


Рисунок 13 – Планарный граф на 35 вершинах

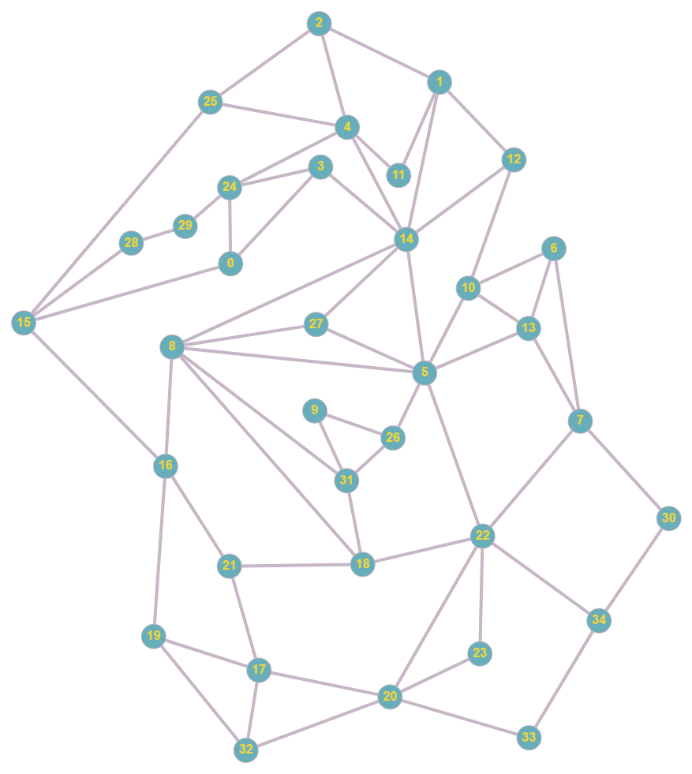


Рисунок 14 – Планарный граф на 46 вершинах

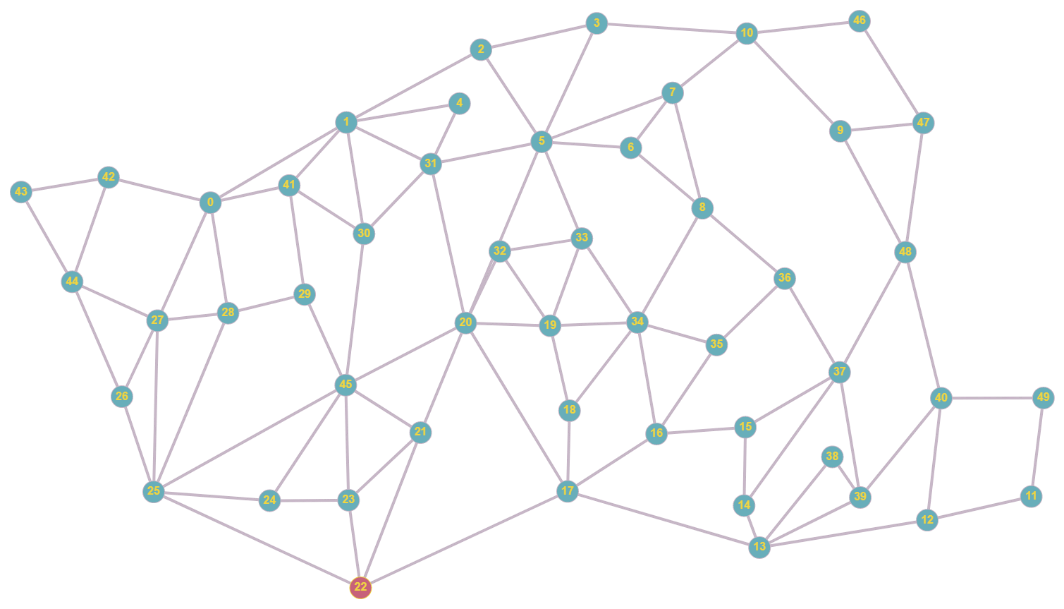


Рисунок 15 – Планарный граф на 50 вершинах

Результаты полченные при использовании RFL, BSC и алгоритма жадной раскраски на данных графах приведены в таблице ниже.

Таблица 3 – Сравнение алгоритмов раскраски графов

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| алгоритмы раскраски  кол-во вершин  в графе | RFL | | | BSC | | | Жадный алгоритм | |
|  | время работы алгоритма | |  | время работы алгоритма | |  | время работы алгоритма |
| 15 | 3 | | 4 мс | 3 | | 3 мс | 3 | 1 мс |
| 26 | 3 | | 10 мс | 3 | | 8 мс | 3 | 1мс |
| 35 | 4 | | 29мс | 4 | | 16 мс | 4 | 2мс |
| 42 | 3 | | 36 мс | 4 | | 16 мс | 4 | 2 мс |
| 50 | 4 | | 57 мс | 4 | | 21 мс | 4 | 2 мс |

Как видно из талицы, жадный алгоритм очень прост в реализации и не требует каких либо рекурсивных вычислений, поэтому выполняется очень быстро. Наилучшим алгоритмом из данных трех, по моему мнению явлется алгоритм RFL, поскольку он дает наиболее приблеженное к точному хроматическое число графа и выполняется тоже быстро. Посольку хроматическое число – это наименьшее количество цветов, требуемых для правильной раскраски, то для ыбранного граа с 42 вершинами данный алгоритм в результате дает оценку ниже, чем другие два. В целом если необходимо обеспечить наилучшее бытродействие, то редпочтение стоит отдать жадному алгориту или агоритму BSC, однако, если необходимо найти как можно меньшее число цветов для правильной расраски графа – то предпочтение стоит отдать RLF алгоритму. Листинг программы приведен в приложении 1. Код сопровожден комментариями.

# ГЛАВА 4 ПРИМЕНЕНИЕ РАСКРАСКИ ГРАФОВ

Раскраска графов применяется, например, для составления расписаний, кластерного анализа, вычисления производных, распараллеливания численных методов, распределения частот, распределения регистров процессоров, в технологии цифровых водяных знаков, и при решении других задач.

Составление расписаний [11].

Задача о составлении расписания с ограничениями подразумевает распределение занятий по времени таким образом, чтобы в одно и то же время не были назначены занятия у одной группы, одного преподавателя или в одном кабинете. Такая задача легко сводится к раскраске графов при условии, что известны все занятия для всех групп, которые надо провести, каждое занятие ведёт определённый преподаватель и известны кабинеты, в которых должны быть проведены занятия. В таком случае расписание может быть представлено графом, в котором вершины соответствуют занятиям, а рёбрами связаны вершины, соответствующие занятиям у одной группы, в одном кабинете или с одним преподавателем. После раскраски такого графа расписание сформируется по принципу: занятия, которые соответствуют вершинам одного цвета могут быть проведены одновременно.

Кластерный анализ [16].

Задача кластеризации (разбиение региона на компактные зоны или группирование объектов) может быть сведена к задаче раскраски вершин графа. Для этого строится граф несовместимости. Вершинам графа соответствуют объекты, и две вершины - смежные, если соответствующие им объекты не могут находиться в одной группе. Ставится задача раскраски вершин такого графа несовместимости при различных условиях (ограничениях, критериях), среди которых:

Минимальная раскраска графа (получение минимального числа компактных групп).

Раскраска графа в заданное число красок.

Раскраска графа в заданное число красок с ограничением на количество соцветных вершин.

Распределение частот [17].

Дан граф 𝐺, вершины которого соответствуют зоне покрытия для одной станции (вышки). Две вершины смежны тогда и только тогда, когда зона покрытия имеет общую границу и перекрывается. Если у вышек есть общая зона покрытия, то этим станциям нельзя задавать одинаковую частоту (чтобы не было помех), т.е. они конкурируют за частоту. Проблема распределения диапазонов частот в системах сотовой связи сводится к раскраске построенного графа. За каждым цветом закреплен диапазон частот. Любая правильная раскраска графа 𝐺 определяет допустимое распределение диапазонов частот по станциям. И наоборот допустимое распределение диапазонов частот по станциям определяет правильную раскраску графа *G*. Если для раскраски 𝑛 вершин использовались цвета *1,2,3,* … , 𝑘, то вершины, раскрашенные в -й цвет – это станции, которые используют один и тот же 𝑖-й диапазон частот, не создавая помех. Минимальная раскраска графа дает распределение частот по базовым станциям, а хроматическое число 𝜒(𝐺) определяет минимальное число диапазонов частот, необходимых для работы всей сети без помех.

Распараллеливание численных методов [14].

Обобщённо представим задачу так: объекты — некие вычисления, между которыми необходимо разделить вычислительные ресурсы (процессоры, компьютеры…), которые могут работать параллельно друг другу. Какие-то вычисления могут выполняться параллельно, какие-то нет. Соответственно, вершинная раскраска графа несовместимости вычислений является искомым распределением.

Численные методы, которые можно распараллелить, используя раскраску графа:

Разложение Холецкого для метода сопряжённых градиентов с предопределением. Этот итерационный метод для решения систем линейных алгебраических уравнений с большими, разреженными, симметричными, положительно определёнными матрицами [10].

Метод Гаусса—Зейделя в применении к разреженным матрицам. Итерационный метод решения систем линейных алгебраических уравнений.

Методы с использованием адаптивно уточняемой сетки. Они полезны в решении дифференциальных уравнений в частных производных.

Предопределение неполным *LU*-разложением. Для решения систем линейных алгебраических уравнений с использованием подпространств Крылова.

Распределение регистров [26].

Чтобы распределить регистры необходимо построить граф интерференции регистров для каждой процедуры в программе. Два вычисления происходящих в машинных регистрах называются интерферентными, если они происходят одновременно в каком-либо участке выполнения программы. Для каждой процедуры в программе граф интерференций строится следующим образом: вершины – определяют регистры и все вычисления данной процедуры, происходящие на машинных регистрах, ребра – определяют интерференцию регистров. Задача сводится к раскраске графа в количество цветов, соответствующее количеству регистров. Когда хроматическое число графа соответствует количеству регистров, это значит, что регистры распределены верно.

## 4.1 ТЕХНОЛОГИЯ ЦИФРОВЫХ ВОДЯНЫХ ЗНАКОВ

Технология цифровых водяных знаков относится к области защиты интеллектуальной собственности. Было разработано множество техник добавления водяных знаков к цифровым данных (изображениям, тексту, аудиофайлам, видео и мультимедиа). Эти техники просто добавляют подпись к цифровым данным меняя таким образом исходные данные.

Создание цифровых водяных знаков в целях охраны интеллектуальной собственности, осложнено тем, что необходимо учитывать корректность функционирования интеллектуальной собственности, на который наносят водяные знаки. Один из методов создания водяных знаков, названный *constraintbased*, переводит будущую подпись в ряд дополнительных ограничений в процессе создания и реализации интеллектуальной собственности, чтобы уникально закодировать подпись в самой интеллектуальной собственности [7].

Одним из алгоритмов создания цифровых водяных знаков, является *QP* алгоритм, предложенный Ку и Потконьяком в 1998 году. Он основан на распределении регистров [8].

Этот алгоритм предполагает индексирование всех вершин графа, так, что каждая вершина имеет индекс - уникальное целое число от 1 до *|V(G)|.* Суть *QP* алгоритма состоит в добавлении дополнительных ребер между вершиной *v[i]* и одной из двух ближайших несмежных с ней вершин такого же цвета. Выбор между этими двумя ближайшими несмежными вершинами определяется битом водяного знака, который будет добавлен. Важно отметить, что это концепция динамическая, поскольку каждый раз при добавлении ребра в окрестности вершины *v[i]* ближайшие несмежные вершины будут различны.

Псевдокод *QP* алгоритма.

На входе: граф *G* без водяного знака, битовое сообщение *W=w[1]w[2]…w[m]*

На выходе: граф *G’* с водяным знаком

*n=|V|*

*G’=G*

*for each i from 1 to n*

*if v[i]* имеет 2 ближайшие несмежные вершины *v[i1]* и *v[i2]*

*if w[i]=0* добавляем ребро между вершинами *v[i]* и *v[i1]* в графе *G’*

*else* добавляем ребро между вершинами *v[i]* и *v[i2]* в графе *G’*

*return G’*

*QP* алгоритм был доработан Майлсом и Колбергом в *QPS* алгоритм. В данной вариации *QP* алгоритма используются 2 понятия [9].

**Определение 4.1.** для графа *G=(V,E)*, если 3 вершины *v,v’,v’’* графа *G* соответствуют следующим условиям:

, , то они называются *triple* (трезубец)

**Определение 4.2.** для графа *G=(V,E)*, если *triple* окрашены в один цвет, то они называются *colored triple* (окрашенный трезубец).

Граф на входе *QPS* алгоритма – граф интерференций программы *P*.

Псевдокод *QPS* алгоритма:

На входе: граф *G* без водяного знака, битовое сообщение *W=w[1]w[2]…w[m]* встраиваемое в *G*

На выходе: граф *G’* с водяным знаком и встроенным сообщением *W*

*n=|V|*

*G’=G*

*j=0*

*for each i from 1 to n*

*if v[i]* не входит в *triple G’* и возможно найти две ближайшие вершины *v[i1]* и *v[i2]* для *v[i]* такие, что *v[i1]* и *v[i2]* имеют такой же цвет как *v[i]* в *G’*, и *v[i1],v[i2]* не включены в *triple* в *G’*

*j++*

*if w[j]=0*

соединяем *v[i]* и *v[i1]* в *G’*

*if w[j]=1*

соединяем *v[i]* и *v[i2]* в *G’*

return *G’(V,E’)* и вставленное сообщение *W’*

Улучшенным вариантом *QP* алгоритма встраивания является *QPI* алгоритм встраивания. Это алгоритм создания водяного знака для информированного программного обеспечения.

Псевдокод *QPI* алгоритма:

На входе: граф *G* без водяного знака, битовое сообщение *W=w[1]w[2]…w[m]* встраиваемое в *G*

На выходе: граф *G’* с водяным знаком и встроенным сообщением *W*

*n=|V|*

*G’=G*

*j=0*

*for each i from 1 to n*

*if v[i]* имеет две несмежные вершины такого же цвета *v[i1]* и *v[i2]*

*j++*

*if w[j]=0*

соединяем *v[i]* и *v[i1]* в *G’*

меняем цвет *v[i1]* на другой, отличный от уже использованных в *G’*

*if w[j]=1*

соединяем *v[i]* и *v[i2]* в *G’*

меняем цвет *v[i2]* на другой, отличный от уже использованных в *G’*

*return G’*

Процесс извлечения сообщения по алгоритмам *QP*, *QPS*, *QPI* полностью описан в источнике [9].

При сравнении вариантов алгоритма *QP* можно сделать вывод о том, что сообщение, встроенное в граф алгоритмом *QP*, не извлекается в общем случае. *QPS* алгоритм – это улучшенная версия *QP* алгоритма, которая была реализована для нанесения водяного знака на программное обеспечение, но он имеет некоторые неясные описания. Алгоритм *QPI* – позволяет полностью реализовать идею алгоритма *QP* и может быть эффективно использован для нанесения водяного знака на программный продукт, используя распределение регистров.

Так же существует технология электронных водяных знаков, основанная на графах, но не на их раскраске. Подробную информацию о данных методах можно получить в источнике [18].

# ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были реализованы алгоритмы раскраски графов на языке программирования высокого уровня, проведен сравнительный анализ данных алгоритмов. Были выполнены следующие задачи:

1. Рассмотрен теоретический материал теории графов и раскраски графов
2. Рассмотрены различные алгоритмы раскраски графов
3. Реализованы алгоритмы раскраски в пакете Microsoft Visual Studio
4. Проведен сравнительный анализ результатов
5. Собрана и освещена информация и практическом использовании раскраски графов

Подводя итог, хотелось бы отметить, что объект исследования нуждается в дальнейшем изучении. В данной работе были использованы только три алгоритма раскраски графа для реализации в Microsoft Visual Studio 2017.

В рамках данной работы, наилучшие результаты показал RLF алгоритм раскраски, который точнее других оценивает хроматическое число, и не требует больших затрат времени на это. Хотя любой из реализованных алгоритмов дает оценку хроматического числа, которая подходит под границы оценки хроматического числа для данных графов. Так же хотелось бы отметить последний пункт четвертой главы – использование раскраски графов в технологии цифровых водяных знаков, как, на мой взгляд, наиболее близкий, к моей специальности, именно это применение вызвало во мне интерес к раскраске графов, как к области знаний.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Шапорев С.Д. Дискретная математика: учебное пособие / Балт. гос. техн. ун-т «Военмех». СПб., 2004, c. 94-97.

[2] Карпов Д.В. Теория графов. [Электронный ресурс] – Режим доступа https://logic.pdmi.ras.ru/~dvk/graphs\_dk.pdf (обращение 01.02.2019).

[3] Walter Klotz. Graph Coloring Algorithms / Mathematic – Bericht, TU Clausthal, 5, 2002, c. 1-9.

[4] Brelaz, D., New methods to color the vertices of a graph/ Communications of the Assoc. of Comput. Machinery 22, 1979, 251-256.

[5] Brown, J. R., Chromatic shelduring and chromatic number problem / Management Science 19, 1972, c. 456-463.

[6] Ю.В. Матиясевич. Некоторые алгебраические методы вычисления количества раскрасок графов. Записки научных семинаров ПОМИ. т.283 2001, с. 193-205.

[7] William Zhu and Clark Thomborson. Algorithms to Watermark Software Though Register Allocation / Department of Computer Sciences, University of Auckland, Auckland, New Zeland, 2006.

[8] G. Qu, M. Potkonjak, Analysis of watermarking techniques for graph coloring problem / in: IEEE/ACM International Conference on Computer Aided Design, 1998, c. 190–193.

[9] W. Zhu, C. Thomborson, On the QP algorithm in software watermarking / in: ISI2005, Vol. 3495 of LNCS, 2005, c. 646–647.

[10] Евстигнеев В. А. Применение теории графов в программировании. М.: Наука, 1985, 352с.

[11] Колясников Д. В., Ватутин Э. И. Анализ степени приближения к оптимуму оценки хроматического числа графа с использованием эвристических методов / Оптико-электронные приборы и устройства в системах распознавания образов, обработки изображений и символьной информации (Распознавание – 2013): материалы XI междунар. Научн.-техн. Конф. – Курск: Изд-во ЮЗГУ, 2013, с. 253-255.

[12] Кристофиденс Н. Теория графов. Алгоритмический подход. Издательство «Мир», Москва, 1978.

[13] Новиков, Ф.А. Дискретная математика для программистов. – 2-е издание [Текст]. – Спб.: Питер, 2007, 364с.

[14] Иванов, Б.Н. Дискретная математика. Алгоритмы и программы [Текст]. – М: Лаборатория базовых знаний, 2003, 288с.

[15] D.Kirovski and M.Potkonjak. Efficient Coloring of a Large Spectrum of Graphs / 35th Design Automation Conference Proceedings, 1998, c. 427-432.

[16] Смирнов А.В., Андрианов И.А., Суконщиков А.А., Бахтенко Е.А. Математическая модель оптимизации доставки товаров автотранспортом на разветвленной сети дорог для решения задач и кластеризации / тенденции развития науки и образования / Сборник научных трудов по материалам XXII международной научной конференции. ч 1. Самара. 2017, с. 35-37.

[17] Мурзаков Д., Зенков М., Жуков А., Тишин В. Применение алгоритма последовательной раскраски графа в сотовой сети / Естественные и математические науки в современном мире / Сб. ст. по материалам XXXI междунар. науч.-практ. конф. № 6 (30). Новосибирск: Изд. «СибАК», 2015.

[18] Deo, N. Graph Theory with Applications to Engineering and Computer Science / Prentice-Hall, Englewood Cliffs, N.J. 1974, 366 с.

[19] Heawood, P.J. Map Color Theorems / Quart. J. Math., 4, 1890, с. 332- 338.

[20] Kempe, A.B. On the geographical problem of four colors / Am. J. Math., 2, 1879, с. 193-204.

[21] Gerhard Ringel, J. W. T. Youngs. Solution of the heawood map-coloring problem / UNIVERSITY OF CALIFORNIA (SANTA CRUZ), 1968.

[22] Thore Husfeldt, Graph colouring algorithms, Chapter XIII of Topics in Chromatic Graph Theory, L. W. Beineke and Robin J. Wilson (eds.), Encyclopedia of Mathematics and its Applications / Cambridge University Press, ISBN 978-1-107-03350-4, 2015, с. 277–303.

[23] A. Kosowski and K.Manuszewski, Classical coloring of graphs, Graph Colorings (ed.M. Kubale) / Amer. Math. Soc. Contemp. Math. 352, 2004, с. 1–20.

[24] M. Langberg, Graph coloring, Encyclopedia of Algorithms (ed. M. Kao) / Springer, 2008, c. 368–371.

[25] Г. А. Донец, Н. 3. Шор. Алгебраический подход к проблеме раскраски плоских графов. Киев, «НАУКОВА ДУМКА», 1982.

[26] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, P. W. Markstein. Register allocation via coloring / Computer Languages, 6, 1981, c. 47–57.

[27] A. Wigderson. Improving the performance guarantee for approximate graph coloring / JACM, 30(4), 1983, c. 729–735.

[28] Визинг В.Г. Об оценке хроматического класса р-графа / сб. Дискретный анализ, Новосибирск, ИМ СО АН СССР, т. 3, 1964, с. 25-30.

[29] Самохин А.В. Проблема четырех красок. Неоконченная история доказательства / Соросовский обр. журнал, т. 6, №7, 2000.

[30] Tutte W.T. On Hamilton circuits / London Math. Soc., 21, 1946, c. 98-101.

# ПРИЛОЖЕНИЕ 1

#include <iostream>

#include <malloc.h>

#include <vector>

#include <cmath>

#include <ctime>

using namespace std;

vector<int> \*NotNEIB(int\*\* x, int y) //вектор вершин несмежных y

{

vector<int> \*res=new vector<int>;

int l = \_msize(x[0]) / sizeof(x[0][0]);

for (int i = 0; i < l; i++)

{

if (i != y)

{

if (x[y][i] == 0) (\*res).push\_back(i);

}

}

return res;

}

int \*\*Copygrapf(int \*\*x, int l) //функция копирования графа, во избежания изменения исходных данных

{

int \*\*G = new int\* [l];

for (int i = 0; i < l; i++) G[i] = new int[l];

for (int i = 0; i < l; i++)

{

for (int j = 0; j < l; j++)

{

G[i][j] = x[i][j];

}

}

return G;

}

int maxdegree(int\*\* x) //поиск вершины с максимальной степенью во всем графе

{

int maxd = 0;

vector<int> degrees;

int l = \_msize(x[0]) / sizeof(x[0][0]);

int currentdeg = 0; int k = 0;

for (int i=0; i < l; i++)

{

currentdeg = 0; k = 0;

for (int j=0; j < l; j++)

{

if (x[i][j] == 1) currentdeg++;

}

for (int j = 0; j < l; j++)

{

if (x[i][j] == -1) k++;

}

if (k == l) currentdeg = -1;

degrees.push\_back(currentdeg);

if (currentdeg >= maxd) maxd = currentdeg;

}

for (int j=0; j < degrees.size(); j++)

{

if (degrees[j] == maxd) return j;

}

}

int degreeofvert(int\*\* x, int y) //степень вершины y

{

int maxd = 0;

int l = \_msize(x[0]) / sizeof(x[0][0]);

int currentdeg = 0;

for (int i=0; i < l; i++)

{

if (x[y][i] == 1) maxd++;

}

return maxd;

}

int maxdegree(vector<int> \*a, int\*\* x) // поиск вершины с максимальной степенью среди данных

{

int maxd = 0; int v = 0;

for (int j = 0; j < a->size(); j++)

{

if (degreeofvert(x, (\*a)[j]) >= maxd) {

maxd = degreeofvert(x, (\*a)[j]); v = j;

}

}

return (\*a)[v];

}

vector<int> \*n(vector<int> \*a, vector<int> \*b) // убирает из множества возможных вершин для в ключения в независимое множество соседей добавленной вершины

{

for (int i = 0; i < a->size(); i++)

{

for (int j=0; j < b->size(); j++)

{

if ((\*a)[i] == (\*b)[j])

{

(\*a).erase(a->begin() + i); i = -1; break;

}

}

}

return a;

}

vector<int> \*neibours(int\*\* x, int y) // возвращает вектор соседей вершины х

{

int l = \_msize(x[0]) / sizeof(x[0][0]);

vector<int> \*res = new vector<int>;

for (int i=0; i < l; i++)

{

if (x[y][i] == 1)

res->push\_back(i);

}

return res;

}

int modv(int\*\* x) //сколько вершин в графе (незакрашенных)

{

int l = \_msize(x[0]) / sizeof(x[0][0]);

int m = 0;

for (int i = 0; i < l; i++)

{

if (x[i][i] == 0 ) m++;

}

return m;

}

int\*\* Gminusvertex(int\*\* x, vector<int> \*y) //фукнция возвращающая граф G-x, где ч - независимое множество

{

int \*\*res = x; int l = \_msize(x[0]) / sizeof(x[0][0]);

for (int i = 0; i < l; i++)

{

for (int j = 0; j < y->size(); j++)

{

if (i == (\*y)[j])

{

for (int k=0; k < l; k++)

{

res[i][k] = -1; res[k][i] = -1;

}

}

}

}

return res;

}

int findmin(vector<int> x) //поиск минимального

{

int m = 0; int index = 0;

for (int i = 0; i < x.size(); i++)

{

if (x[i] <= m) {

m = x[i]; index = i;

}

}

return index;

}

int numcn(vector<int> \*a, vector<int> \*b) // количество общих вершин в окрестности вершины х и окрестности вершины у

{

int n = 0;

for (int i = 0; i < a->size(); i++)

{

for (int j = 0; j < b->size(); j++)

{

if ((\*a)[i] == (\*b)[j]) n++;

}

}

return n;

}

int placeofmax(vector<int> \*a) // наибольшее значение вектора

{

int maxd = -1; int k = 0;

for (int i = 0; i < (a->size()); i++)

{

if ((\*a)[i] >= maxd) { maxd = (\*a)[i]; k = i; }

}

return k;

}

vector<int> \*AA(int\*\* x) // функция возвращающая вектор вершин в порядке неувеличения их степеней

{

int l = \_msize(x[0]) / sizeof(x[0][0]);

vector<int> \*A = new vector<int>;

vector<int> \*deg = new vector<int>;

for (int i = 0; i < l; i++)

deg->push\_back(degreeofvert(&\*(x), i));

int maxd = 0;

for (int i = 0; i < deg->size(); i++)

{

maxd = placeofmax(deg);

A->push\_back(maxd);

(\*deg)[maxd] = -1;

}

return A;

}

vector<int> ucolor(int\*\* x, vector<int> \*A, int i, vector<int> color, vector<int> U, int opt) //функция поиска допустимых цветов для данной вершины

{

vector<int> \*u = &U;

int c;

vector<int> \*nei = neibours(x, i);

for (int j = 0; j < nei->size(); j++)

{

for (int j1 = 0; j1 < A->size(); j1++)

{

if ((\*nei)[j] == (\*A)[j1] && color[j1]!=-1)

{

c = color[j1];

for (int l = 0; l < (\*u).size(); l++)

{

if ((\*u)[l] == c) { (\*u).erase((\*u).begin() + l); break; }

}

}

}

}

for (int l = 0; l < (\*u).size(); l++)

{

if ((\*u)[l] > opt) { (\*u).erase((\*u).begin() + l, (\*u).end()); break; }

}

return (\*u);

}

void Greedy(int\*\* x) //жадный алгоритм раскраски

{

time\_t start = clock();

int \*\* G = x;

vector<int> color;

vector <int> \*colors = new vector<int>;

vector<int> \*neib = new vector<int>;

for (int i = 0; i < modv(G); i++) {

color.push\_back(-1);

}

int currentvertex = 0; bool C = true;

while (currentvertex != modv(G))

{

colors->clear();

neib = neibours(G, currentvertex);

for (int i = 0; i < neib->size(); i++)

{

if (color[(\*neib)[i]] != -1) colors->push\_back(color[(\*neib)[i]]);

}

for (int i = 0; i < modv(G); i++)

{

C = true;

for (int j = 0; j < colors->size(); j++)

{

if (i == (\*colors)[j]) {

C = false; break;

}

}

if (C) {

color[currentvertex] = i; break;

}

}

currentvertex++;

}

int p = placeofmax(&color);

cout << "Greedy alg" << '\t' << color[p]+1 << "\t\t" << (clock() - start) << endl;

delete colors, neib;

int c = \_msize(x[0]) / sizeof(x[0][0]);

for (int i = 0; i < c; i++)

{

delete G[i];

}delete G;

return;

}

void BSC(int\*\* x) //BSC алгоритм раскраски

{

time\_t start = clock();

vector<int> color;

int l = \_msize(x[0]) / sizeof(x[0][0]);

for (int i = 0; i < l; i++) {

color.push\_back(-1);

}

vector<int> \*nei = new vector<int>;

int init = 0;

int\*\* G = x;

bool back;

int c = 0;

vector<int> A = \*AA(G);

vector<int> U(l);

vector<vector<int>> freecolor (l);

for (int i = 0; i < l; i++)

{ U[i]=i;

}

for (int i = 0; i < l; i++)

{

freecolor[i].resize(l);

for (int j = 0; j < l; j++)

{

freecolor[i][j]=U[j];

}

}

int OptColorNumber = modv(G) + 1;

int leasti = 0;

int currentvert = A[0];

while (A[init] != A[l-1]) {

back = false;

for (int i = init; i < modv(G); i++)

{

if (i > init)

{

currentvert = A[i];

freecolor[currentvert]= ucolor(G, &A, currentvert, color, U, OptColorNumber);

}

if (freecolor[currentvert].size() > 0)

{ c = freecolor[currentvert][0];

freecolor[currentvert].erase(freecolor[currentvert].begin());

if (i > 0) color[i] = c > color[i-1] ? c : color[i-1];

else color[i] = c;

}

else

{

init = i - 1;

back = true; break;

}

}

if (back) {

if (init >= 0) {

currentvert = A[init];

color[init] = -1;

freecolor[currentvert] = ucolor(G, &A, A[init], color, U, OptColorNumber);

}

}

else {

OptColorNumber = color[modv(G)-1];

for (int kk = A.size()-1; kk >=0; kk--)

{

if (color[kk] == OptColorNumber)

{ leasti = (A.size()-1-kk); break; }

}

init = leasti - 1;

if (init < 0) break;

for (int h = 0; h < color.size(); h++)

{

if (h >= init)

color[h] = -1;

}

for (int i = 0; i < l; i++) {

freecolor[i] = ucolor(G, &A, A[i], color, U, OptColorNumber);

}

currentvert = A[init];

}

}

cout << "BSC" << "\t\t" << color[modv(G)-1]+1 << "\t\t" << (clock() - start) << endl;

delete nei;

for (int i = 0; i < l; i++)

{

delete G[i];

}

delete G;

return;

}

void RLF(int\*\* x) //RLF алгоритм раскраски графа

{

time\_t start = clock();

int \*\*G = x;

vector<int> \*colors = new vector<int>;

for (int i = 0; i < \_msize(x[0]) / sizeof(x[0][0]); i++) colors->push\_back(0);

vector<int> \*NN;

vector<int> \*clique = new vector<int>; //независимые множества

vector<int> \*neib = new vector<int>;

int colornumber = 0;

int currentvert = 0;

int l = \_msize(x[0]) / sizeof(x[0][0]);

int yvert;

int maxcn, cn, ydegree;

while (modv(G) > 0 )

{

clique->clear();

currentvert = maxdegree(G);

clique->push\_back(currentvert);

colornumber++;

(\*colors)[currentvert] = colornumber;

NN = NotNEIB(G, currentvert);

while (!(NN->empty()))

{

maxcn = -1; ydegree = -1;

for (int i = 0; i < NN->size(); i++)

{

cn = numcn(neibours(G, currentvert), neibours(G, (\*NN)[i]));

if (cn > maxcn || (cn == maxcn && degreeofvert(G, (\*NN)[i]) < ydegree))

{

yvert = (\*NN)[i]; ydegree = degreeofvert(G, (\*NN)[i]);

maxcn = cn;

}

}

if (maxcn == 0) yvert = maxdegree(NN, G);

(\*colors)[yvert] = colornumber;

neib = neibours(G, yvert);

neib->push\_back(yvert);

NN = n(NN, neib);

clique->push\_back(yvert);

}

G = Gminusvertex(G, clique);

}

time\_t end = clock() - start;

cout << "RLF" << "\t\t" << colornumber << "\t\t" << end << endl;

delete colors, neib, NN ,clique;

for (int i = 0; i < l; i++)

{

delete G[i];

}delete G;

return;

}

int main()

{

int graph1[15][15] = { { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},

{0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1},

{0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1},

{0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1},

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0},

{1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1},

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0},

{0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0} };

int graph2[26][26] = { {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },

{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },

{0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 },

{0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0},

{0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} };

int graph3[35][35] = { {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},

{ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},

{ 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0} }

;

int graph4[42][42] =

{ {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0},

{0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0},

{0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},

{0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0},

{0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1},

{0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0} };

int graph5[50][50] =

{ {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0},

{ 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0 },

{ 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 },

{ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 },

{ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 },

{ 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};

int n =50;

;// 15, 26, 35, 42, 50 };

int \*\*graph = new int\*[n];

for (int i1 = 0; i1 < n; i1++) graph[i1] = new int[n];

for (int i1 = 0; i1 < n; i1++) {

for (int j = 0; j < n; j++)

{

graph[i1][j] = graph5[i1][j];

}

};

cout << "algname \t colornum \t mc" << endl;

BSC(Copygrapf(graph,n));

RLF(Copygrapf(graph, n));

Greedy(Copygrapf(graph,n));

for (int j = 0; j < n; j++)

delete graph[j];

delete graph;

system("pause");

return 0;

}