МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Кафедра алгебры, геометрии и дискретной математики**

Направление подготовки: «Фундаментальная информатика и информационные технологии»

Профиль подготовки: «Инженерия программного обеспечения»

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

на тему:

**«Сравнение нескольких алгоритмов решения задач о независимом множестве и вершинной раскраске для связных и хордальных графов»**

Выполнила:

студент(ка) группы 381806-1

Параничева А.В.

Подпись

Проверил:

кандидат

физико-математических наук

Сорочан С.В.

Подпись

Нижний Новгород

2022

Оглавление

[1. Введение 4](#_Toc105673853)

[2. Постановка задачи 5](#_Toc105673854)

[3. Теоретические основы, описание алгоритмов 6](#_Toc105673855)

[3.1. Общие сведения о графах 6](#_Toc105673856)

[3.2. Представление графа 7](#_Toc105673857)

[3.3. Описание алгоритмов 8](#_Toc105673858)

[3.3.1. Описание алгоритма распознавания хордальных графов 8](#_Toc105673859)

[3.3.2. Описание алгоритма построения хордального графа 9](#_Toc105673860)

[3.3.3. Описание алгоритма нахождения наибольшего независимого множества в связном графе 9](#_Toc105673861)

[3.3.4. Описание алгоритма нахождения наибольшего независимого множества в хордальном графе, основанного на факте наличия в нём смежно поглощающей вершины 10](#_Toc105673862)

[3.3.5. Описание алгоритма нахождения наибольшего независимого множества в хордальном графе, основанного на факте наличия в нём симплициальной вершины 10](#_Toc105673863)

[3.3.6. Описание общего алгоритма поиска оптимальной вершинной раскраски в произвольном графе (алгоритм с экспоненциальной сложностью) 10](#_Toc105673864)

[3.3.7. Описание полиномиального алгоритма поиска оптимальной вершинной раскраски для хордального графа, основанного на использовании факта наличия в нём симплициальной вершины 11](#_Toc105673865)

[4. Структура программы 12](#_Toc105673866)

[4.1. Класс Graph 12](#_Toc105673867)

[5. Описание экспериментов 18](#_Toc105673868)

[5.1. Задание и распознавание хордальных графов 18](#_Toc105673869)

[5.2. Исследование работы алгоритмов поиска наибольшего независимого множества 20](#_Toc105673870)

[5.3. Исследование работы алгоритмов нахождения вершинной раскраски 24](#_Toc105673871)

[6. Заключение 29](#_Toc105673872)

[7. Литература 30](#_Toc105673873)

[8. Приложение 31](#_Toc105673874)

# Введение

В современном мире теория графов играет большую роль, так как она полезна практически во всех научных отраслях. Им нашли применение в физике, биологии, химии, математике, истории, лингвистике, социальных науках, технике и т.п. Наибольшей популярностью теоретико-графовые модели используются при исследовании коммуникационных сетей, систем информатики, химических и генетических структур, электрических цепей и других систем сетевой структуры.

Объяснить популярность этого раздела математики легко: с его помощью можно наглядно представить и описать различные модели, объекты и структуры, поэтому графы используют не только в технических науках, но и в гуманитарных. Карты метро, системы дорог соединяющие города, схемы в учебниках – все это представляется с помощью графов.

Многие алгоритмы, применяемые при работе с графами, принадлежат к классу NP-полных задач. Вычислительная задача называется NP-полной (от англ. non-deterministic polynomial – «недетерминированные с полиномиальным временем»), если для неё не существует эффективных алгоритмов решения. К таким задачам относятся, например, поиск наибольшего независимого множества и вершинной раскраски. Однако существуют особые типы графов, для которых подобные задачи являются полиномиально разрешимыми [см. 8], например, таким типом графов является хордальный. Поэтому появился особый интерес к распознаванию конкретных типов графов, а также сведение графов к нужному типу [см. 4] для решения NP-полных задач.

Хордальные графы используются в различных сферах деятельности человека: в компьютерном зрении, химии, биологии. Одним из примеров их использования является изучения взаимодействия белка, так как в основе каждого белка лежит полипептидная цепь, которая имеет трехмерную структуру [см. 6, 7], которую можно представить с помощью хордального графа.

В данной работе будут рассмотрены общие алгоритмы наибольшего независимого множества и вершинной раскраски графов, их модификации для хордальных графов и будет проведен анализ, сравнение и оценка результатов их выполнения.

# Постановка задачи

В ходе исследования разработать программу, позволяющую:

* реализовать класс, позволяющий работать с графами с возможностью задавать их с помощью матрицы смежности и выводить графы на экран
* реализовать функцию, генерирующую хордальные графы
* реализовать функцию, позволяющую распознавать хордальные графы
* реализовать алгоритм нахождения наибольшего независимого множества в связном графе
* реализовать алгоритм нахождения наибольшего независимого множества в хордальном графе, основанный на использовании факта наличия в нём смежно поглощающей вершины
* реализовать полиномиальный алгоритм поиска наибольшего независимого множества для хордального графа, основанный на использовании факта наличия в нём симплициальной вершины
* реализовать общий алгоритм поиска оптимальной вершинной раскраски в произвольном графе (алгоритм с экспоненциальной сложностью)
* реализовать полиномиальный алгоритм поиска оптимальной вершинной раскраски для хордального графа, основанный на использовании факта наличия в нём симплициальной вершины.

На основе испытаний сделать оценку результатов испытаний и их сравнение.

# Теоретические основы, описание алгоритмов

## Общие сведения о графах

Основные понятия и термины, используемые в этой работе, можно найти, например, в [1, 2, 5].

Граф - это математическая модель, представленная совокупностью множества вершин и связей между ними.

Ориентированный граф — граф, рёбрам которого присвоено направление. Направленные рёбра именуются также дугами, а в некоторых источниках и просто рёбрами.

Граф, ни одному ребру которого не присвоено направление, называется неориентированным графом.

Говорят, что вершина a смежна другой вершине b, если граф содержит ребро (a, b).

Окрестностью вершины a называется порождённый подграф, образованный всеми вершинами, смежными b.

Путем от А1 до Аn в графе называется такая последователь­ность ребер, ведущая от A1 к Аn, в которой каждые два соседних ребра имеют общую вершину и никакое ребро не встречается более одного раза.   
Путь от А1 до Аn называется простым, если он не проходит ни через одну из вершин графа более одного раза.

**Циклом** называется путь, в котором совпадают его начальная и конечная вершины.  
**Простым циклом** в графе называется цикл, не проходящий ни через одну из вершин графа более одного раза.

Две вершины А и В графа называются **связными**, если в графе существует путь с концами А и В.

Две вершины графа называются **несвязными**, если в графе не суще­ствует ни одного пути, связывающего их.

Граф называется **связным**, если каждые две вершины его связные.

Полный граф — простой неориентированный граф, в котором каждая пара различных вершин смежна.

Хордальный граф - это связный граф без порождённых циклов длины более чем три (цикл длины 4 и больше содержит хорду).

Симплициальная вершина – это вершина окрестность которой порождает клику.

В хордальном графе всегда найдется симплициальная вершина, иначе граф нехордален.

Независимое множество графа - множество вершин графа G, такое, что любые две вершины в нем не смежны (никакая пара вершин не соединена ребром).

Наибольшее независимое множество графа – независимое множество графа, содержащее наибольшее количество вершин.

Вершина a смежно поглощает вершину b, если вершины a и b смежные и N(b)\{a} ⊆ N(a)\{b}. Значение этого понятия состоит в том, что при удалении любой смежно поглощающей вершины из графа не изменяется число независимости.

Теорема. В любом непустом хордальном графе имеется смежно поглощающая вершина.

Поэтому для хордального графа наибольшее независимое множество можно найти с помощью одних только сжатий по включению. Нужно только находить смежно поглощающие вершины и удалять их из графа до тех пор, пока оставшийся граф не станет пустым. Множество оставшихся вершин и является наибольшим независимым множеством.

Также для нахождения наибольшего независимого множества в хордальном графе можно использовать факт обязательного наличия в нем симплициальной вершины. Так как такая вершина содержит в окрестности клику, то все смежные ей вершины не могут быть в наибольшем независимом множестве.

Правильная вершинная раскраска – это раскраска вершин графа, при которой любые смежные вершины окрашены в разные цвета.

Теорема. Для любого хордального графа χ(G) = ω(G).

Поэтому чтобы найти хроматическое число в хордальном графе надо найти у него наибольшую клику.

Для нахождения вершинной раскраски в хордальном графе используем тот же факт наличия в нём симплициальной вершины.

Алгоритм поиска (или обхода) в глубину (англ. depth-first search, DFS) позволяет построить обход графа, при котором посещаются все вершины, доступные из начальной вершины.

Будем использовать элементы поиска в глубину для построения хордальных графов.

## Представление графа

В программе реализована такая структура данных, как граф (рис. 1).

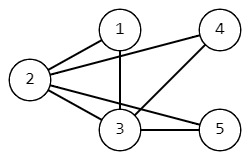


Рис. 1. Рисунок графа.

Характеристиками графа являются:

1. Количество вершин (целое число).
2. Матрица смежности (рис. 2). Матрица смежности представляет из себя матрицу, где заголовки строк и столбцов соответствуют номерам вершин графа, а само значение каждого ее элемента a(i,j) определяется наличием или отсутствием ребер между вершинами i и j: если ребро есть на пересечении будет 1, если ребра нет 0. Так как в программе реализуются неориентированные графы, то матрица смежности симметрична относительно главной диагонали.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 1 | 1 | 0 | 1 | 1 |
| 4 | 0 | 1 | 1 | 0 | 0 |
| 5 | 0 | 1 | 1 | 0 | 0 |

Рис. 2. Матрица смежности

## Описание алгоритмов

### Описание алгоритма распознавания хордальных графов

Описание алгоритмов поиска наибольшего независимого множества и оптимальной вершинной раскраски для графов общего вида можно найти, например, в [2,3,5], а для хордальных графов в [2,9].

Чтобы определить является ли граф хордальным или нет, понадобится нахождение в нем симплициальных вершин. Вершина называется симплициальной, если ее окрестность является порожденным полным подграфом.

Алгоритм распознования заключается в непрерывном нахождении сиплициальных вершин в графе и их удалении, после удаления одной из таких вершин хордальность графа должна сохраняться, иначе граф не является хордальным.

### Описание алгоритма построения хордального графа

Генерируем случайный граф с нужным количеством вершин и делаем его гарантированно связным. Например, можно сгенерировать путь из всех вершин и недостающие ребра добавить в граф.

Далее начинаем построение дерева, пока не задействуем все вершины. Для его построения будем пользоваться алгоритмом поиска в глубину.

Создаем стек и добавляем в него нулевую вершину – корень нашего дерева. Далее строим DFS дерево. Если во время построения нам попадается цикл больше 3 (смотрим циклы только с вершинами, находящимся в стеке), то из активной вершины проводим ребра ко всем вершинам цикла. Если циклов за раз оказалось несколько, то проставляем ребра тому циклу, чья вершина, образующая цикл, находится глубже в стеке.

Первоначальное построение закончится, когда опустеет стек.

Далее смотрим у каких вершин есть несколько детей и дополнительно проставляем ребра между всеми вершинами, начиная от этого родителя до корня.

### Описание алгоритма нахождения наибольшего независимого множества в связном графе

Чтобы найти наибольшее независимое множество в связном графе, воспользуемся такой структурой данных как очередь. Вначале добавляем в нее исходный граф, потом промежуточные графы на каждой итерации.

Каждый раз извлекаем из очереди граф, дублируем его и находим в нем вершину с наибольшей окрестностью. Далее в одной копии мы удаляем саму вершину с наибольшей окрестностью в другом конкретно ее окрестность (то есть все смежные с ней вершины) После проверяем оба графа на пустоту. Если матрица смежности пуста, значит мы нашли независимое множество. Определяем является ли оно наибольшим, сравнив с предыдущим наибольшим множеством. Если новое множество оказалось наибольшим, заменяем старое множество новым. Если матрица смежности оказалась непустой, то добавляем получившийся граф в очередь для дальнейших преобразований.

Алгоритм выполняется пока очередь не станет пустой.

### Описание алгоритма нахождения наибольшего независимого множества в хордальном графе, основанного на факте наличия в нём смежно поглощающей вершины

Для нахождения наибольшего независимого множества в хордальном графе, мы ищем вершину, которая смежно поглощает другую (т.е. первая вершина смежна второй вершине, и вся окрестность второй вершины содержится в окрестности первой). Если независимое множество содержит первую вершину, то не может содержать вторую, следовательно, будем удалять вершину с большей окрестностью. Удаляем поглощающие вершины до тех пор, пока матрица смежности не опустеет (не останется ребер).

### Описание алгоритма нахождения наибольшего независимого множества в хордальном графе, основанного на факте наличия в нём симплициальной вершины

Для нахождения наибольшего независимого множества в хордальном графе можем использовать факт обязательного наличия в нём симплициальной вершины.

Идём по порядку по всем вершинам и проверяем их симплициальность. Первую найденную симплициальную вершину удаляем из графа. Повторяем процедуру пока граф содержит ребра.

Оставшиеся вершины и есть искомое множество.

### Описание общего алгоритма поиска оптимальной вершинной раскраски в произвольном графе (алгоритм с экспоненциальной сложностью)

Изначально проверяем граф на полноту - если граф полный, то вершинная раскраска по размеру равна всем вершинам в графе.

Если все же граф не полный, то создаем очередь и помещаем в нее исходный граф.

Далее пока очередь не пуста, вынимаем оттуда граф и создаем две его копии. В первой копии проводим ребро между несмежными вершинами, а во второй копии объединяем две несмежные вершины. Если получившиеся графы не полны, то помещаем их в очередь. Модифицируем графы пока из них не получатся полные графы.

Итогом алгоритма будет полный граф с наименьшим числом вершин. Это число и будет хроматическим числом графа.

Также алгоритм сохраняет какую вершину в какой цвет покрасил, поэтому на выходе будет не только хроматическое число, но и сама раскраска вершин.

### Описание полиномиального алгоритма поиска оптимальной вершинной раскраски для хордального графа, основанного на использовании факта наличия в нём симплициальной вершины

Если рассматриваемый граф полный, то вершинная раскраска состоит из всех вершин графа.

Если граф не полный, то создаем пустой вектор. Пока хордальный граф не пустой будем искать в нем симплициальную вершину. Найдя такую, записываем ее в вектор и смотрим размер ее окрестности: размер наибольшей клики в хордальном графе будет равен хроматическому числу. Далее удаляем эту вершину из графа.

Когда граф опустеет, в векторе образуется очередь из симплициальных вершин, а также мы уже будем знать хроматическое число графа.

Последним шагом будет раздача цветов всем вершинам в обратном порядке добавления их в вектор.

# Структура программы

Код программы находится в файлах:

**main.cpp** содержит основную функцию с тестом программы,

**Graphs.h** – заголовочный файл, содержащий объявление класса Graph,

**Graphs.cpp** – файл, содержащий реализацию основных функций класса Graph,

**ChordalGraphs.cpp** – файл, содержащий реализацию методов хордальных графов,

**IndependentSet.cpp** – файл, содержащий различные реализации методов для поиска независимых множеств,

**Coloring.cpp** – файл, содержащий различные реализации вершинных раскрасок графов

**ExtraFuns.h**- заголовочный файл, содержащий объявление дополнительных функции, необходимых для решения задач,

**ExtraFuns.cpp** – файл с реализацией дополнительных функций.

**Tests.h** – заголовочный файл, содержащий объявление различных тестов с графами

**Tests.cpp** – файл, содержащий реализацию тестов

## Класс Graph

class Graph

{

private:

int n;

int\* M;

public:

Graph();

Graph(int \_n);

Graph(int \_n, std::vector<int> \_M);

Graph(const Graph& tmp);

~Graph();

const Graph& operator = (const Graph&);

bool operator == (const Graph&) const;

int GetN();

int\* GetM();

Graph Mark\_Graph();

Graph ReduceVert(int v);

int Find\_Ver\_Most\_Rel();

int Find\_Ver\_Less\_Rel();

Graph ReduceNeighborVert(int v);

bool IsCompleteGraph();

Graph Fill\_Graph();

void Gen\_ChordalGr();

bool IsCompleteArea(vector<int> arr, int\* \_M) const;

int SimplVer();

bool isChordalGraph() const;

vector<int> Independent\_set\_for\_ChordalGr\_Absorb\_Ver();

vector<int> Independent\_set\_for\_ChordalGr\_Simpl\_Ver();

int ClickNumber(int);

void Coloring\_for\_ChordalGr();

int AbsorbVer();

bool Is\_empty\_matrix() const;

vector<int> Independent\_Set();

void Coloring\_General();

pair <int, int> PairNonAdjVert();

Graph UnionPairVert(int, int);

static vector<int> LabelingUnionVerts(vector<int>, int, int);

int Check\_Cycle\_with\_Vert(int, std::vector<int>);

int Find\_Unused\_Vert(int top, std::vector<int> used);

friend istream& operator >> (istream& input, Graph& g);

friend ostream& operator << (ostream& output, const Graph& g);

};

*Поля класса:*

int n – количество вершин в графе.

int\* M – матрица смежности.

*Методы класса:*

Graph();

***Назначение:*** конструктор по умолчанию.

Graph(int \_n);

***Назначение:*** конструктор с параметрами (принимает заданное количество вершин).

Graph(int \_n, std::vector<int> \_M);

***Назначение:*** конструктор с параметрами (принимает заданное количество вершин и матрицу смежности).

Graph(const Graph& tmp);

***Назначение:*** конструктор копирования.

~Graph();

***Назначение:*** деструктор.

const Graph& operator = (const Graph&);

***Назначение:*** перегрузка оператора = (принимает константную ссылку на граф).

bool operator == (const Graph&) const;

***Назначение:*** перегрузка оператора == (принимает константную ссылку на граф).

int GetN();

***Назначение:*** функция, возвращающая количество вершин в графе.

int\* GetM();

***Назначение:*** функция, возвращающая матрицу смежности в графе.

Graph Mark\_Graph();

***Назначение:*** функция, маркирующая вершины графа.

Graph ReduceVert(int v);

***Назначение:*** функция, удаляющая вершину и всю ее окрестность в матрице смежности (принимает индекс вершины, которую надо удалить).

int Find\_Ver\_Most\_Rel();

***Назначение:*** функция, находящая вершину с наибольшей окрестностью.

int Find\_Ver\_Less\_Rel();

***Назначение:*** функция, находящая вершину с наименьшей окрестностью (больше 0).

Graph ReduceNeighborVert(int v);

***Назначение:*** функция, удаляющая окрестность вершины (принимает вершину, окрестность которой надо удалить).

bool IsCompleteGraph();

***Назначение:*** функция, проверяющая граф на полноту.

Graph Fill\_Graph();

***Назначение:*** функция, заполняющая пустой граф ребрами (на выходе гарантированно будет связный граф).

void Gen\_ChordalGr();

***Назначение:*** функция, генерирующая хордальные графы.

bool IsCompleteArea(vector<int> arr, int\* \_M) const;

***Назначение:*** функция, проверяющая является ли окрестность вершины полным графом (принимает вектор, содержащий окрестность вершины и матрицу смежности).

int SimplVer()

***Назначение:*** функция, возвращающая первую найденную симплициальную вершину (если такая есть).

bool isChordalGraph() const

***Назначение:*** функция, возвращающая true, если граф хордален.

vector<int> Independent\_set\_for\_ChordalGr\_Absorb\_Ver();

***Назначение:*** функция, находящая наибольшее независимое множество в хордальном графе (с использованием смежно поглощающей вершины).

int AbsorbVer();

***Назначение:*** функция, находящая вершину, которая поглощает окрестность другой смежной с ней вершины.

bool Is\_empty\_matrix() const;

***Назначение:*** функция, возвращающая true, если матрица смежности графа пуста (не содержит ребер).

int ClickNumber(int);

***Назначение:*** функция, считающая мощность клики.

void Coloring\_for\_ChordalGr();

***Назначение:*** функция, считающая хроматическое число и выводящая цвета вершин в хордальном графе.

vector<int> Independent\_Set();

***Назначение:*** функция, находящая наибольшее независимое множество в связном графе.

void Coloring\_General();

***Назначение:*** функция, считающая хроматическое число графа и выводящая цвета вершин.

pair <int, int> PairNonAdjVert();

***Назначение:*** функция, находящая пару несмежных вершин.

Graph UnionPairVert(int, int);

***Назначение:*** функция, объединяющая пару несмежных вершин.

static vector<int> LabelingUnionVerts(vector<int>, int, int);

***Назначение:*** функция, маркирующая вершины.

int Check\_Cycle\_with\_Vert(int, std::vector<int>);

***Назначение:*** функция, находящая цикл среди выбранных вершин.

int Find\_Unused\_Vert(int top, std::vector<int> used);

***Назначение:*** функция, находящая еще неиспользованные вершины в алгоритме.

friend istream& operator >> (istream& input, Graph& g)

***Назначение:*** перегрузка операции ввода для графа.

friend ostream& operator << (ostream& output, const Graph& g)

***Назначение:*** перегрузка операции вывода для графа.

# Описание экспериментов

## Задание и распознавание хордальных графов

Для определения корректности работы программы и для вычисления временных зависимостей относительно количества заданных вершин проведем ряд экспериментов.

Для начала проверим есть ли зависимость между хордальностью графа, числом вершин в графе и количеством ребер (график 1).

График 1. Вероятность хордальности графа при увеличении числа вершин.

В ходе исследования можно сделать вывод, что чем больше вершин в графе, тем меньше вероятность, что он при случайном генерировании ребер будет хордальным. Также, чем больше ребер в графе (чем ближе к полному), тем выше вероятность, что он будет хордальным. Однако, создавать только полные графы или почти полные не целесообразно, поэтому для проведения качественных и информативных тестов для дальнейших исследований потребовалось создать алгоритм, генерирующий хордальные графы.

Рассмотрим время создания хордального графа в зависимости от количества вершин (график 2).

График 2. Временная зависимость алгоритма построения хордального графа от числа вершин.

По результатам можно заметить, как сильно время алгоритма зависит от количества вершин в графе, так как в ходе него осуществляется обход по всем вершинам графа.

Теперь исследуем временную зависимость выполнения алгоритма распознавания хордальности от числа вершин в графе (график 3).

График 3. Временная зависимость алгоритма распознавания хордальности от числа вершин в графе.

На основе результатов делаем такой же вывод, что в предыдущем испытании.

## Исследование работы алгоритмов поиска наибольшего независимого множества

Далее рассмотрим время выполнения алгоритмов нахождения наибольшего независимого множества в хордальном графе и сравним результаты (график 4).

На графах испытывается общий алгоритм нахождения наибольшего независимого множества и адаптированные алгоритмы для хордальных графов.

График 4. Время выполнения нескольких алгоритмов поиска наибольшего независимого множества.

По результатам испытаний можно увидеть, что время выполнения алгоритмов для хордальных графов примерно одинаково, тогда как время общего алгоритма в разы больше на тех же графах. К тому же при одинаковом количестве вершин общий алгоритм показывает абсолютно различное время: при проведении тестов разница во времени составляла 10-20 раз для общего алгоритма (представлено наименьшее время работы общего алгоритма поиска в испытаниях для наглядности графика).

Рассмотрим отдельно два алгоритма поиска наибольшего независимого множества для хордальных графов и поэкспериментируем с количеством вершин и заполнением графов рёбрами (график 5, график 6).

График 5. Время нахождения наибольшего независимого множества в хордальных графах с помощью алгоритма со смежно поглощающими вершинами с разным числом вершин и заполненностью графа

График 6. Время нахождения наибольшего независимого множества в хордальных графах с помощью алгоритма с симплициальными вершинами с разным числом вершин и заполненностью графа

По результатам тестов можно сделать вывод, что время выполнения алгоритма со смежным поглощением в целом не зависит от степени заполненности графа, тогда как в алгоритме с симплициальными вершинами такая зависимость есть: чем меньше заполненность графа тем дольше выполняется алгоритм. Разница во времени у двух алгоритмов незначительна и все же алгоритм со смежным поглощением чаще выполняется быстрее.

Посмотрим теперь как меняется число независимости в хордальном графе в зависимости от количества вершин и разной вероятностью образования рёбер (график 7).

График 7. Среднее число независимости в хордальном графе с разным числом вершин и вероятностью выпадения ребра.

Число независимости в хордальном графе растет пропорционально увеличению вершин, резких скачков показателей нет. Больший рост числа независимости виден при уменьшении вероятности выпадения рёбер в графе, что следует из определения, так как чем меньше ребер, тем меньше смежных вершин.

Проведем подобные исследования и на произвольных графах (график 8).

График 8. Среднее число независимости в произвольном графе с разным числом вершин и вероятностью выпадения ребра.

Результаты тестов, проведенных на произвольных графах, примерно совпадают с результатами тестов проведенных на хордальных графах: с увеличением числа вершин и с уменьшением вероятности выпадения рёбер число независимости растет.

Однако есть некоторое различие в среднем значении числа независимости для одинакового числа вершин в графе и выпадения рёбер.

Чтобы наглядно увидеть разницу, занесем результаты тестов по среднему числу независимости в один график (график 9). Сплошные ломанные визуализируют результаты экспериментов, проведенных на хордальных графах, пунктирные ломанные – на произвольных.

График 9. Число независимости для произвольных и хордальных графов (зависимость от количества вершин и вероятности выпадения рёбер).

По результатам тестов видно, что при одинаковых показателях в среднем число независимости у хордального графа выше, чем у произвольного. Данную особенность можно объяснить спецификой строения хордальных графов: сгенерированные хордальные графы менее разнообразны по своей структуре, чем произвольные.

## Исследование работы алгоритмов нахождения вершинной раскраски

Рассмотрим теперь алгоритмы раскрасок.

Запуская тесты с общим алгоритмом раскраски, сразу сталкиваешься с проблемой, что при одинаковом количестве вершин тесты выдают абсолютно разное время, и это не удивительно. Время выполнения зависит от того насколько граф “близок” к полному: чем меньше ребер между вершинами, тем дольше будет работать алгоритм, так как он базируется на том, чтобы объединять вершины и проводить ребра до образования полных графов.

В ходе экспериментов будем генерировать произвольный граф с 10 вершинами и с вероятностью ребра 0,8, в этом случае в среднем получаются такие результаты (график 10):

Если в графе хроматическое число 8, то работает алгоритм 1млс, если – 7, то работает 3 млс, примерно те же результаты при хроматическом числе 6 и 5, при 4 – 5 млс.

Сгенерируем так же граф из 10 вершин, но уже с вероятностью ребра 0,5.

Результаты:

При хроматическом числе – 5, работает 18 млс; при 4 работает 31 млс, при 3 – 41 млс. Видно, что в выпадающих графах за счет уменьшения вероятности выпадения ребра сократилось и хроматическое число и, соответственно, увеличилось время выполнения алгоритма.

Теперь еще уменьшим вероятность генерации ребра до 0,3

Почти всегда выпадают графы с хроматическим числом 3 и работает алгоритм от 100 до 500 млс.

График 10. Время выполнения общего алгоритма раскраски на произвольном графе с 10 вершинами и разной вероятностью генерирования ребер

При увеличении числа вершин всего лишь до 15 временные показатели значительно вырастают (график 11):

Вероятность ребра 0,8:

хроматическое число 9 – 50млс, 8 – 90млс, 7 – 200млс, 6 – 600млс, 5 – 1500млс.

Вероятность ребра 0,5:

хроматическое число 6 от 1000 до 7000млс, 5 от 2500 до 20700млс, 4 – от 27900 до 49900млс.

Вероятность ребра 0,3:

хроматическое число 3 – 218000млс.

График 11. Время выполнения общего алгоритма раскраски на произвольном графе с 15 вершинами и разной вероятностью генерирования ребер.

Можно сделать вывод, что время выполнения алгоритма зависит в большей части не от количества вершин, а от заполненности графа ребрами. Это очень неудобно, так как при одинаковом количестве вершин алгоритм на разных графах может работать в тысячи раз дольше, из-за этого невозможно посчитать среднее время работы при определенном количестве вершин, а, следовательно, использовать данный алгоритм где-то.

Теперь рассмотрим время выполнения общего алгоритма раскраски и раскраски для хордального графа на одинаковых хордальных графах (график 12).

Для исследования сгенерируем хордальный граф из 50 вершин. Хордальный алгоритм зависит лишь от числа вершин в графе и, следовательно, время выполнения на разных графах с данным числом вершин примерно одинакова – 16 млс. Что же получается по времени у общего алгоритма: здесь оно варьируется от 1 млс до 236600 млс на тех же графах.

При генерировании графов из 100 вершин среднее время выполнения хордального алгоритма раскраски 63. Среднее время общего варианта раскраски, отследить невозможно. При генерировании графов из 150 вершин почти всегда тесты останавливаются на общем алгоритме из-за нехватки памяти.

График 12. Время выполнения хордальной раскраски.

Проведем следующие испытания, выявляющие среднее хроматическое число в хордальном графе с различным количеством вершин и ребер в нем (график 13).

График 13. Хроматическое число в хордальном графе в зависимости от количества вершин и вероятности выпадения в нем рёбер.

В результате данного испытания можно прийти к выводу, что хроматическое число растет пропорционально увеличению количеству вершин и повышению вероятности образования ребра, резких скачков в наблюдениях нет. Также можно заметить, что в хордальном графе в раскраске приходится использовать достаточно большое число цветов. Данное наблюдение объясняется спецификой алгоритма построения хордального графа, которому приходится достраивать большое число ребер для обеспечения хордальности, из-за чего образовываются большие клики, от которых и зависит хроматическое число в хордальном графе.

Теперь попробуем провести подобное испытание на графах с тем же количеством вершин и вероятностью появления ребра, но уже на произвольных графах.

Ожидаемо, но мощи вычислительной машины хватило только на графы в пределах 20 вершин. Добавим в эксперимент графы из 15 вершин для прослеживания общей динамики. Результаты испытаний представлены на графике ниже (график 14).

График 14. Хроматическое число в произвольном графе в зависимости от количества вершин и вероятности выпадения в нем рёбер.

Среднее хроматическое число равномерно растет по мере увеличения числа в графе. При снижении вероятности заполнения графа рёбрами хроматическое число уменьшается, так как становится меньше связей между вершинами.

Так как у произвольных связных графов нет никаких требований к построению, то хроматическое число у них в среднем ниже, чем у хордальных.

# Заключение

В ходе практической работы была разработана и реализована программа, которая может генерировать графы с заданным количеством вершин. Были реализованы алгоритмы: построения хордальных графов, их распознавания, нахождения наибольшего независимого множества для связных и хордальных графов, а также нахождения вершинной раскраски в связных и хордальных графах.

По результатам испытаний можно сделать такой вывод о том, что использовать общие алгоритмы на графах нерационально: все испытания указывают на то, что время выполнения таких алгоритмов непостоянно, поэтому предугадать затратность по ресурсам такого алгоритма в конкретном случае почти нереально.

Разумным выходом из данной ситуации является усовершенствование алгоритмов, необходимых для решения поставленных задач. Для этого необходимо выявить категории графов, которые используются в исследуемой области и сведение каждого алгоритма под теоретическую базу каждого из них. Таким образом, заметно внушительное улучшение по времени выполнения алгоритмов, а также предсказуемость и стабильность результатов.

# Литература

1. Алексеев В.Е., Захарова Д.В. Теория графов: учебное пособие. – Нижний Новгород: изд-во ННГУ, 2017. – 119 с.
2. Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных: учебник. – Нижний Новгород: изд-во ННГУ, 2005. – 307 с.
3. Ахо А.В., Хопкрофт Дж., Ульман Дж.Д. Структуры данных и алгоритмов / Пер. с англ.: Уч. пос. Москва: Издательский дом «Вильямс», 2000. – 400 с.
4. Близнец И.А. Алгоритмы и нижние оценки на вычислительную сложность задач модификации графов: автореф. дис…канд. физ.-мат. наук: 13.04.16. – Санкт-Петербург. – 22 с.
5. Емеличев В.А., Мельников О. И., Сарванов В. И., Тышкевич Р. И., Лекции по теории графов. – Москва: Наука, 1990. – 384с.
6. Прогнозирование структур белков методами полуопределенного программирования / Подкопаев А.С., Карасиков М.Е., Максимов Ю.В. // Научные труды МФТИ. – 2015. – С. 66-73.
7. Турсунбай кызы Ы. Деревья клик хордального графа и деревья подграфов в теории графов // Конструирование и оптимизация параллельных программ. –Новосибирск 2009. – С.314-321.
8. Турсунбай кызы Ы. Локальные и динамические алгоритмы для анализа граф-моделей систем: автореф. дис…канд. физ.-мат. наук: 28.12.11. – Новосибирск: изд-во ИСИ СО РАН, 2011. – 21 с.
9. Файловый архив студентов. Белорусский государственный университет. – Режим доступа: <https://studfile.net/> – Загл. с экрана.

# Приложение

Graphs.h

#ifndef GRAPHS\_H

#define GRAPHS\_H

#include <iostream>

#include <iomanip>

#include <utility>

#include <vector>

using namespace std;

class Graph

{

private:

int n;

int\* M;

public:

Graph();

Graph(int \_n);

Graph(int \_n, std::vector<int> \_M);

Graph(const Graph& tmp);

~Graph();

const Graph& operator = (const Graph&);

bool operator == (const Graph&) const;

int GetN();

int\* GetM();

Graph Mark\_Graph();

Graph ReduceVert(int v);

int Find\_Ver\_Most\_Rel();

int Find\_Ver\_Less\_Rel();

Graph ReduceNeighborVert(int v);

bool IsCompleteGraph();

Graph Fill\_Graph();

void Gen\_ChordalGr();

bool IsCompleteArea(vector<int> arr, int\* \_M) const;

int SimplVer();

bool isChordalGraph() const;

vector<int> Independent\_set\_for\_ChordalGr\_Absorb\_Ver();

int AbsorbVer();

int ClickNumber(int);

void Coloring\_for\_ChordalGr();

vector<int> Independent\_set\_for\_ChordalGr\_Simpl\_Ver();

bool Is\_empty\_matrix() const;

vector<int> Independent\_Set();

void Coloring\_General();

pair <int, int> PairNonAdjVert();

Graph UnionPairVert(int, int);

static vector<int> LabelingUnionVerts(vector<int>, int, int);

int Check\_Cycle\_with\_Vert(int, std::vector<int>);

int Find\_Unused\_Vert(int top, std::vector<int> used);

friend istream& operator >> (istream& input, Graph& g)

{

for (int i = 0; i < g.n \* g.n; i++)

input >> g.M[i];

return input;

}

friend ostream& operator << (ostream& output, const Graph& g)

{

output << "Quantity of vertex is " << g.n << endl;

output << "The adjacency matrix: ";

for (int i = 0; i < g.n \* g.n; i++)

{

if (i % g.n == 0)

output << endl;

output << setw(5) << setprecision(2) << right << g.M[i];

}

output << endl;

return output;

}

};

#endif

ChordalGraphs.cpp

#include <ctime>

#include <stack>

#include <vector>

#include "Graphs.h"

#include "ExtraFuns.h"

int Graph::Check\_Cycle\_with\_Vert(int top, std::vector<int> used) {

int check\_v = used[top];

int cycle\_v = -1;

while (check\_v != 0) {

check\_v = used[check\_v];

if (M[top \* n + check\_v] == 1) {

cycle\_v = check\_v;

}

}

return cycle\_v;

}

int Graph::Find\_Unused\_Vert(int top, std::vector<int> used) {

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

if (M[top \* n + i] == 1 && used[i] == -1)

return i;

return -1;

}

void Graph::Gen\_ChordalGr() {

unsigned int start\_time = clock();

stack<int> stack;

stack.push(0);

vector<int> used(n);

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

used[i] = -1;

used[0] = 0;

while (!stack.empty()) {

int top = stack.top();

int cycle\_v = this->Check\_Cycle\_with\_Vert(top, used);

if (cycle\_v != -1) {

int new\_adj\_v = used[top];

while (new\_adj\_v != cycle\_v) { //проставляем новые ребра чтобы триангулировать граф

new\_adj\_v = used[new\_adj\_v];

M[top \* n + new\_adj\_v] = 1;

M[new\_adj\_v \* n + top] = 1;

}

}

int new\_top = -1;

while (new\_top == -1 && !stack.empty()) {

top = stack.top();

new\_top = this->Find\_Unused\_Vert(top, used);

if (new\_top != -1) {

used[new\_top] = top;

top = new\_top;

stack.push(top);

}

else {

stack.pop();

}

}

}

vector<int> branches(n);

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

branches[i] = 0;

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

branches[used[i]]++;

int full\_rel = 0;

for (int i = 1; i < n; i++)

{

if (branches[i] >= 2)

{

for (int ind = i; ind != full\_rel; ind = used[ind]) {

if (ind == 0)

break;

for (int ind2 = used[ind]; ind2 != 0; ind2 = used[ind2])

{

M[ind \* n + ind2] = 1;

M[ind2 \* n + ind] = 1;

}

full\_rel = ind;

M[ind \* n + 0] = 1;

M[ind] = 1;

}

}

}

unsigned int end\_time = clock();

unsigned int search\_time = end\_time - start\_time;

cout << "---------------------------" << endl;

cout << search\_time << " Gen Chord graph time" << endl;

cout << "---------------------------" << endl;

}

bool Graph::IsCompleteArea(vector<int> arr, int\* \_M) const

{

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

for (int j = i + 1; j < arr.size(); j++)

if (\_M[arr[i] \* n + arr[j]] != 1)

return false;

return true;

}

int Graph::SimplVer()

{

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

{

vector<int> arr;

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

if (M[i \* n + j] == 1)

arr.push\_back(j);

if (arr.empty())

continue;

if (IsCompleteArea(arr, M))

return i;

}

return -1;

}

bool Graph::isChordalGraph() const

{

Graph tmp(\*this);

tmp = tmp.Mark\_Graph();

while (tmp.n != 1)

{

int simpver;

simpver = tmp.SimplVer();

if (simpver == -1)

return false;

tmp = tmp.ReduceVert(simpver);

}

return true;

}

vector<int>Graph::Independent\_set\_for\_ChordalGr\_Absorb\_Ver()

{

Graph tmp(this->Mark\_Graph());

int delver;

while (true)

{

if (tmp.Is\_empty\_matrix())

{

vector<int> res;

for (int i = 0; i < tmp.n; i++)

res.push\_back(tmp.M[i \* tmp.n + i] - 10);

return res;

}

delver = tmp.AbsorbVer();

tmp = tmp.ReduceVert(delver);

}

}

int Graph::AbsorbVer()

{

vector<int> arr;

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

{

arr.clear();

arr.push\_back(i);

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

if (M[i \* n + j] == 1) {

arr.push\_back(j);

}

for (int j = 1; j < arr.size(); j++)

{

bool flag = true;

for (int k = 0; k < arr.size(); k++)

{

if (j == k)

continue;

if (M[arr[j] \* n + arr[k]] != 1)

{

flag = false;

break;

}

}

if (flag == true) {

return arr[j];

}

}

}

}

vector<int> Graph::Independent\_set\_for\_ChordalGr\_Simpl\_Ver()

{

Graph tmp(this->Mark\_Graph());

int simplver;

while (true)

{

if (tmp.Is\_empty\_matrix())

{

vector<int> res;

for (int i = 0; i < tmp.n; i++)

res.push\_back(tmp.M[i \* tmp.n + i] - 10);

return res;

}

simplver = tmp.SimplVer();

tmp = tmp.ReduceNeighborVert(simplver);

}

}

int Graph::ClickNumber(int simpver) {

int click\_num = 1;

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

if (M[simpver \* n + i] == 1)

click\_num++;

return click\_num;

}

void Graph::Coloring\_for\_ChordalGr() {

unsigned int start\_time = clock();

if (this->IsCompleteGraph()) {

unsigned int end\_time = clock();

cout << "===========General\_Coloring===========" << endl;

cout << endl;

cout << "Chromatic number " << this->n << endl;

cout << "Colors:" << endl;

for (int i = 0; i < this->n; i++) {

cout << i << " " << endl;

}

cout << endl;

cout << endl;

unsigned int search\_time = end\_time - start\_time;

cout << "General\_Coloring time " << search\_time << endl;

cout << endl;

cout << "======================================" << endl;

return;

}

Graph initial\_gr(this->Mark\_Graph());

Graph tmp(this->Mark\_Graph());

vector<int> mass\_simplver(tmp.n);

int chromatic\_num = 0;

int count = 0;

while (tmp.n != 1)

{

int simpver = tmp.SimplVer(); // вернулся индекс а не маркировка

mass\_simplver[count] = tmp.M[simpver \* tmp.n + simpver] - 10;

if (chromatic\_num < tmp.n) {

int click\_num = tmp.ClickNumber(simpver);

if (click\_num > chromatic\_num)

chromatic\_num = click\_num;

}

tmp = tmp.ReduceVert(simpver);

count++;

}

mass\_simplver[count] = tmp.M[0] - 10;

vector<int> coloring\_verts(initial\_gr.n);

for (int i = 0; i < initial\_gr.n; i++)

coloring\_verts[i] = -1;

int idx = initial\_gr.n - 1;

while (idx >= 0) {

int cur\_simpvert = mass\_simplver[idx];

vector<bool> order\_colors(chromatic\_num);

for (int i = 0; i < chromatic\_num; i++)

order\_colors[i] = false;

for (int i = idx + 1; i < initial\_gr.n; i++) {

int prev\_vert = mass\_simplver[i];

if (initial\_gr.M[cur\_simpvert \* initial\_gr.n + prev\_vert] == 1)

order\_colors[coloring\_verts[prev\_vert]] = true;

}

int k = 0;

while (coloring\_verts[cur\_simpvert] == -1) {

if (order\_colors[k] == false) {

coloring\_verts[cur\_simpvert] = k;

}

k++;

}

idx--;

}

unsigned int end\_time = clock();

//вывод цветов

cout << "+++++++++Coloring\_for\_Chordal\_Graph+++++++++" << endl;

cout << endl;

cout << "Chromatic number " << chromatic\_num << endl;

cout << "Colors:" << endl;

for (int i = 0; i < coloring\_verts.size(); i++)

cout << coloring\_verts[i] << " ";

cout << endl;

cout << endl;

//вывод времени

unsigned int search\_time = end\_time - start\_time;

cout << "Chordal Coloring time " << search\_time << endl;

cout << endl;

cout << "+++++++++++++++++++++++++++++++++++++++++++++" << endl;

}

IndependentSet.cpp

#include <iostream>

#include <queue>

#include <vector>

#include "Graphs.h"

using namespace std;

vector<int> Graph::Independent\_Set() {

queue<Graph> q;

Graph tmp = this->Mark\_Graph();

q.push(tmp);

Graph gr\_res;

while (!q.empty()) {

tmp = q.front();

q.pop();

int v = tmp.Find\_Ver\_Most\_Rel();

Graph tmpv(tmp.ReduceVert(v));

if (!tmpv.Is\_empty\_matrix())

q.push(tmpv);

else if (gr\_res.n < tmpv.n)

gr\_res = tmpv;

Graph tmpe = tmp.ReduceNeighborVert(v);

if (!tmpe.Is\_empty\_matrix())

q.push(tmpe);

else if (gr\_res.n < tmpe.n)

gr\_res = tmpe;

}

vector<int> res;

for (int i = 0; i < gr\_res.n; i++) {

res.push\_back(gr\_res.M[i \* gr\_res.n + i] - 10);

}

return res;

}

Coloring.cpp

#include <algorithm>

#include <ctime>

#include <iostream>

#include <queue>

#include <string>

#include <vector>

#include "Graphs.h"

#include "ExtraFuns.h"

using namespace std;

void Graph::Coloring\_General() {

unsigned int start\_time = clock();

if (this->IsCompleteGraph()) {

unsigned int end\_time = clock();

cout << "===========General\_Coloring===========" << endl;

cout << endl;

cout << "Chromatic number " << this->n << endl;

cout << "Colors:" << endl;

for (int i = 0; i < this->n; i++) {

cout << i << " " << endl;

}

cout << endl;

cout << endl;

unsigned int search\_time = end\_time - start\_time;

cout << "General\_Coloring time " << search\_time << endl;

cout << endl;

cout << "======================================" << endl;

return;

}

queue<Graph> q;

queue<vector<int>> q\_mas\_unionverts;

vector<int> mas\_unionverts(n);

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

mas\_unionverts[i] = i;

Graph tmp = this->Mark\_Graph(); //маркируем и создаем граф

q.push(tmp);

q\_mas\_unionverts.push(mas\_unionverts);

Graph gr\_res(tmp); // граф для ответа

vector<int> resmas\_unionverts(mas\_unionverts);

while (!q.empty()) {

tmp = q.front();

q.pop();

mas\_unionverts = q\_mas\_unionverts.front();

q\_mas\_unionverts.pop();

pair <int, int> pair\_nonadj\_vert = tmp.PairNonAdjVert();

int ind\_v1 = pair\_nonadj\_vert.first;

int ind\_v2 = pair\_nonadj\_vert.second;

int v1 = tmp.M[ind\_v1 \* tmp.n + ind\_v1] - 10;

int v2 = tmp.M[ind\_v2 \* tmp.n + ind\_v2] - 10;

Graph leftcase(tmp);

leftcase.M[ind\_v1 \* leftcase.n + ind\_v2] = 1;

leftcase.M[ind\_v2 \* leftcase.n + ind\_v1] = 1;

if (leftcase.IsCompleteGraph()) {

if (gr\_res.n > leftcase.n) {

gr\_res = leftcase;

resmas\_unionverts = mas\_unionverts;

}

}

else {

q.push(leftcase);

q\_mas\_unionverts.push(mas\_unionverts);

}

mas\_unionverts = LabelingUnionVerts(mas\_unionverts, v1, v2);

Graph rightcase = tmp.UnionPairVert(ind\_v1, ind\_v2);

if (rightcase.IsCompleteGraph()) {

if (gr\_res.n > rightcase.n) {

gr\_res = rightcase;

resmas\_unionverts = mas\_unionverts;

}

}

else {

q.push(rightcase);

q\_mas\_unionverts.push(mas\_unionverts);

}

}

unsigned int end\_time = clock();

cout << "===========General\_Coloring===========" << endl;

cout << endl;

cout << "Chromatic number " << gr\_res.n << endl;

cout << endl;

unsigned int search\_time = end\_time - start\_time;

cout << "General\_Coloring time " << search\_time << endl;

cout << endl;

cout << "======================================" << endl;

}

pair <int, int> Graph::PairNonAdjVert() {

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

for (int j = i + 1; j < n; j++)

if (M[i \* n + j] == 0)

return { i, j };

return { -1, -1 };

}

Graph Graph::UnionPairVert(int ind\_v1, int ind\_v2) {

Graph tmp(\*this);

for(int i = 0; i < tmp.n; i++)

if (M[ind\_v2 \* n + i] == 1) {

tmp.M[ind\_v1 \* tmp.n + i] = 1;

tmp.M[i \* tmp.n + ind\_v1] = 1;

}

tmp = tmp.ReduceVert(ind\_v2);

return tmp;

}

vector<int> Graph::LabelingUnionVerts(vector<int> label\_mas, int v1, int v2) {

for (int i = 0; i < label\_mas.size(); i++)

if (label\_mas[i] == v2)

label\_mas[i] = v1;

return label\_mas;

}