Федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский Политехнический университет Петра Великого Физико-Механический институт

Лабораторная 2 – Графы

«Вариант 1 – Проверка изоморфизма графов»

|  |  |  |
| --- | --- | --- |
| Выполнил студент гр. 5030102/30101: | Теслин К.М. |  |
| Преподаватель: | Новиков Ф. А. |  |
| Работа принята: | Дата |  |

Санкт-Петербург

2025

Содержание

[1. Введение 3](#_Toc212707944)

[1.1. Задачи работы: 3](#_Toc212707945)

[2. Теоретическая часть 4](#_Toc212707946)

[2.1. Изоморфизм графов 4](#_Toc212707947)

[2.2. Алгоритм VF2 4](#_Toc212707948)

[2.3. Основные принципы работы алгоритма VF2 5](#_Toc212707949)

[2.3.1. Построение частичного отображения. 5](#_Toc212707950)

[2.3.2. Проверка согласованности (feasibility rules). 5](#_Toc212707951)

[2.3.3. Поиск в глубину. 5](#_Toc212707952)

[2.3.4. Откат (backtracking). 5](#_Toc212707953)

[3. Практическая часть 6](#_Toc212707954)

[3.1. Реализация алгоритма 6](#_Toc212707955)

[3.2. Структура программы 6](#_Toc212707956)

[3.2.1. C++ часть (main.cpp) 6](#_Toc212707957)

[3.2.2. Структура данных: 7](#_Toc212707958)

[3.2.3. Python часть (visual.py) 7](#_Toc212707959)

[3.3. Сложность операций 7](#_Toc212707960)

[3.4. Сравнение работы алгоритма на разных типах графов 7](#_Toc212707961)

[3.4.1. Разреженные графы 8](#_Toc212707962)

[3.4.2. Графы средней плотности 8](#_Toc212707963)

[3.4.3. Плотные графы 8](#_Toc212707964)

[3.5. Когда алгоритм не сможет работать 8](#_Toc212707965)

[3.6. Выбор способа представления графов 9](#_Toc212707966)

[4. Примеры 10](#_Toc212707967)

[4.1. Пример 1. Два одинаковых треугольника (привел Теслин К.) 10](#_Toc212707968)

[4.2. Пример 2. Две цепи (привел Колбин М.) 11](#_Toc212707969)

[4.3. Пример 3. 12](#_Toc212707970)

[5. Интерфейс приложения 14](#_Toc212707971)

[5.1. Основной модуль (C++) 14](#_Toc212707972)

[5.2. Модуль визуализации (Python) 15](#_Toc212707973)

[6. Заключение 16](#_Toc212707974)

[6.1. Перспективы улучшения 16](#_Toc212707975)

[7. Приложение А 17](#_Toc212707976)

[8. Приложение Б 21](#_Toc212707977)

# Введение

В современном анализе графовых структур одна из ключевых задач — проверка изоморфизма графов, то есть определение, можно ли с помощью переименования вершин один граф преобразовать в другой без изменения структуры связей. Поэтому целью данной лабораторной работы была реализация программы, проверяющей, являются ли два заданных неориентированных графа изоморфными, с использованием итеративной версии алгоритма VF2.

## Задачи работы:

1. Изучить теоретические основы задачи изоморфизма графов и принцип работы алгоритма VF2.
2. Разработать и реализовать программу на языке **C++ версии C++17**, выполняющую проверку изоморфизма двух неориентированных графов.
3. Использовать **итеративную версию алгоритма VF2**, основанную на пошаговом сопоставлении вершин с применением стека для обхода состояний.
4. Реализовать модуль для **визуализации графов** на языке **Python версии 3.13.3** с использованием библиотеки **NetworkX** и **Matplotlib**, обеспечивающий наглядное представление исходных структур.
5. Провести тестирование программы на различных наборах входных данных, включая как изоморфные, так и не изоморфные графы.
6. Проанализировать корректность, вычислительную сложность и ограничения предложенной реализации.

# Теоретическая часть

## Изоморфизм графов

**Изоморфизм графов** — это биекция между множествами вершин двух графов, сохраняющая смежность. То есть, если вершины u и v соединены ребром в первом графе, то их образы φ(u) и φ(v) также должны быть соединены ребром во втором графе.

Изоморфизм графов — это биекция между множествами вершин двух графов, сохраняющая структуру их связей. Формально, два неориентированных графа  
 называются *изоморфными*, если существует взаимно-однозначное отображение

такое, что для любых вершин

Иными словами, изоморфные графы имеют одинаковую структуру рёбер, отличаясь только обозначениями вершин.

## Алгоритм VF2

Алгоритм **VF2** (Vento–Foggia 2) - один из наиболее эффективных алгоритмов для проверки изоморфизма и поиска подграфов. Он использует:

* построение частичного отображения вершин;
* правила согласованности (feasibility rules), чтобы отсекать заведомо неверные соответствия;
* поиск в глубину (в данной реализации - итеративный с использованием стека).

Алгоритм гарантирует корректность результата, но имеет экспоненциальную временную сложность в худшем случае. Однако на практике он эффективен для графов умеренного размера.

## Основные принципы работы алгоритма VF2

* + 1. Построение частичного отображения.  
       Алгоритм постепенно формирует сопоставление вершин добавляя по одной паре вершин за шаг.  
       На каждом шаге выбирается вершина ​, для которой подбирается подходящая вершина , ещё не участвующая в сопоставлении.
    2. Проверка согласованности (feasibility rules).  
       Прежде чем добавить новую пару в текущее сопоставление, алгоритм проверяет, не противоречит ли она уже установленным связям:
* если соединена с ранее сопоставленной вершиной в ,  
  то и должна быть соединена с ​;
* вершина не должна быть уже использована ранее.
  + 1. Поиск в глубину.  
       Процесс построения сопоставления организуется как поиск в глубину по дереву состояний.  
       В данной реализации используется **итеративный** подход с применением **стека**, где каждое состояние содержит информацию о текущем уровне сопоставления, возможных кандидатах и выбранных вершинах.
    2. Откат (backtracking).  
       Если на текущем шаге не удаётся найти подходящего кандидата, алгоритм возвращается на предыдущий уровень и пробует другие варианты.

1. Практическая часть

## Реализация алгоритма

Программа реализует **итеративную версию алгоритма VF2** с использованием стека вместо рекурсии. Каждое состояние алгоритма представлено структурой State, которая хранит:

* depth — текущий уровень сопоставления (номер вершины из ​);
* mapping — текущее соответствие вершин ;
* candidates — список возможных кандидатов для сопоставления;
* current\_candidate — индекс текущего кандидата, который проверяется.

На каждой итерации алгоритм:

1. Извлекает текущее состояние из стека.
2. Если все вершины уже сопоставлены (depth = n) — найден изоморфизм.
3. Иначе выбирает следующего кандидата, создаёт новое состояние и помещает его в стек.
4. При исчерпании кандидатов происходит возврат (backtracking) к предыдущему уровню.

Такой подход полностью эквивалентен рекурсивной реализации, но позволяет явно управлять стеком и экономить память стека вызовов ОС.

## Структура программы

Программа состоит из двух частей:

### C++ часть (main.cpp)

* Язык реализации: C++ (совместим с C++11 и выше).
* **Основные функции:**
  + - 1. read\_graph\_from\_file() — сохранение и загрузка графа в файл. Формат файла: текстовый или бинарный, поддерживает удобное хранение матрицы смежности.
      2. generate\_candidates() — формирование списка допустимых вершин для отображения при проверке изоморфизма. Используется для оптимизации алгоритма VF2, сокращая количество проверок.
      3. vf2\_iterative() — основной алгоритм проверки изоморфизма графов (итеративная реализация VF2). Работает с матрицей смежности, сравнивает графы по вершинам и рёбрам.

### Структура данных:

* + Матрица смежности реализована как vector<vector<int>>.
  + Позволяет эффективно проверять наличие рёбер и реализовывать алгоритм VF2.

### Python часть (visual.py)

* Язык реализации: Python 3.13.3
* Библиотеки:

1. networkx — работа с графами, построение структур.
2. matplotlib — визуализация графов.

* Функционал:
  1. Чтение графов из файла, созданного C++ частью.
  2. Визуализация графов в для наглядного сравнения.

## Сложность операций

Основной итеративный DFS:

* Перебирает все возможные сопоставления вершин
* Для каждого состояния вызывает generate\_candidates.

**Худший случай:**

* Всего возможных отображений (перестановок) вершин:
* Сложность генерации кандидатов:
* **Итоговая сложность:**

**Примечание:** на практике алгоритм эффективнее благодаря отсеву кандидатов, но теоретический верхний предел остается экспоненциальным.

## Сравнение работы алгоритма на разных типах графов

Алгоритм VF2 предназначен для проверки изоморфизма двух неориентированных графов. Эффективность его работы существенно зависит от **структуры** и **плотности** графов, которые подаются на вход.

### Разреженные графы

**Пример:** деревья, цепочки, звёздные графы.

* В таких графах большинство вершин имеют малую степень (1–2), поэтому при проверке совместимости (функция generate\_candidates) **многие кандидаты отбрасываются уже на первых шагах**.
* Алгоритм быстро находит изоморфизм или убеждается, что его нет.
* **Время работы близко к линейному или полиномиальному** на практике, хотя теоретически остаётся экспоненциальным.

**VF2 работает лучше всего на разреженных графах.**

### Графы средней плотности

**Пример:** решётки, графы с равномерным распределением рёбер.

* Отсев кандидатов всё ещё работает, но хуже — больше вершин имеют одинаковую структуру связей.
* Количество возможных сопоставлений увеличивается, но всё же не доходит до полного перебора.

Алгоритм работает среднеэффективно, часто справляется с 10–20 вершинами за миллисекунды.

## Плотные графы

**Пример:** графы, где почти каждая вершина соединена с каждой.

* У всех вершин одинаковая степень, структура почти одинакова.
* Проверка compatible почти не отбрасывает кандидатов — все вершины считаются допустимыми.
* Алгоритм вынужден рассматривать почти все перестановки (n! возможных отображений).

VF2 работает значительно хуже, время растёт экспоненциально.  
На 10–12 вершинах такие графы уже могут привести к заметным задержкам.

* 1. Когда алгоритм не сможет работать
* Теоретически VF2 всегда завершает работу, но на практике может работать очень долго.
* На плотных графах с 12+ вершинами время работы становится неприемлемым (из-за фактора n!).
* Также производительность падает, если графы представлены неоптимально (например, как список рёбер без быстрого доступа к связности).

## ****Выбор способа представления графов****

В коде граф представлен в виде **матрицы смежности**.

Причины выбора:

1. Быстрая проверка связности. Проверка выполняется за O(1), что критично для алгоритма, который делает миллионы таких проверок.
2. Простота реализации. Удобно читать граф из файла и проверять рёбра без сложных структур.
3. Подходит для малых и средних графов.  
   Матрица смежности занимает памяти, что допустимо при
4. Совместимость с VF2. Алгоритм часто обращается к элементам по индексам, поэтому прямой доступ через матрицу эффективнее, чем обход списка смежности.

# Примеры

## Пример 1. Два одинаковых треугольника (привел Теслин К.)

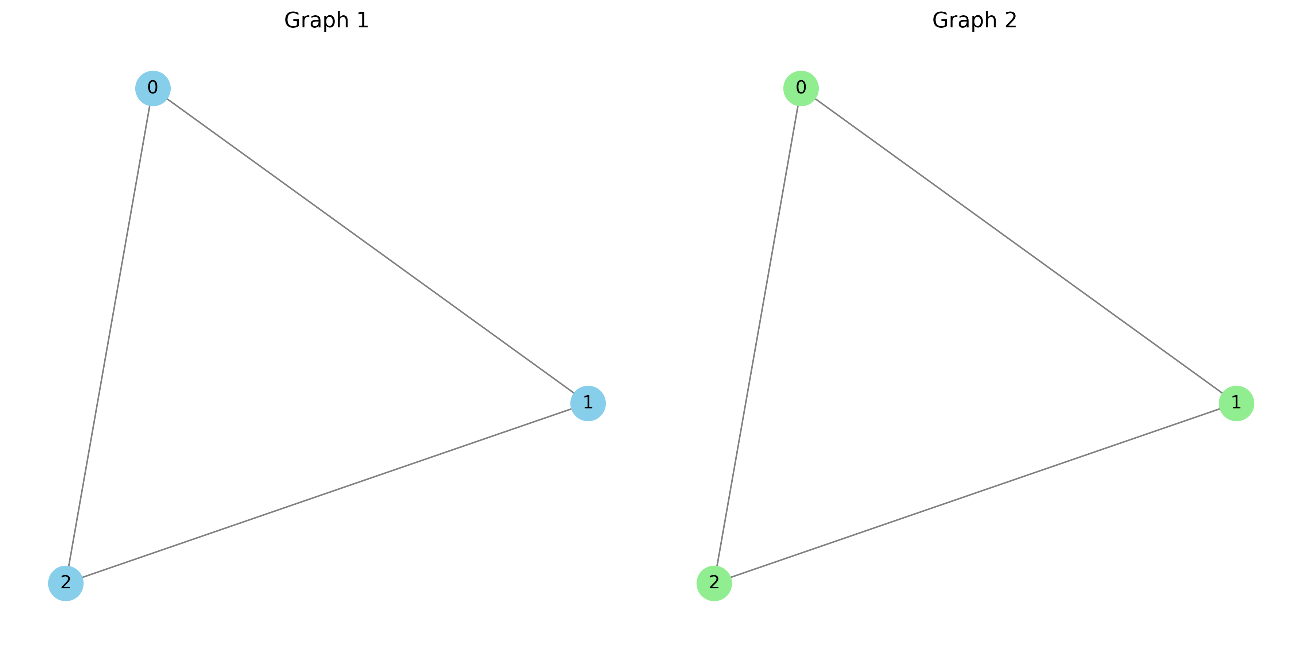


Рисунок 1. Два одинаковых треугольника

* Вершины: 0,1,2
* Рёбра: (0,1),(1,2),(0,2)

**Что делает VF2:**

* 1. depth=0 → кандидаты: [0,1,2]
  2. depth=1 → кандидаты, compatible с вершиной 0 → [1,2]
  3. depth=2 → кандидат 2 подходит → все вершины сопоставлены

Найден изоморфизм: 0→0, 1→1, 2→2

**Итог:** графы изоморфны.

## Пример 2. Две цепи (привел Колбин М.)

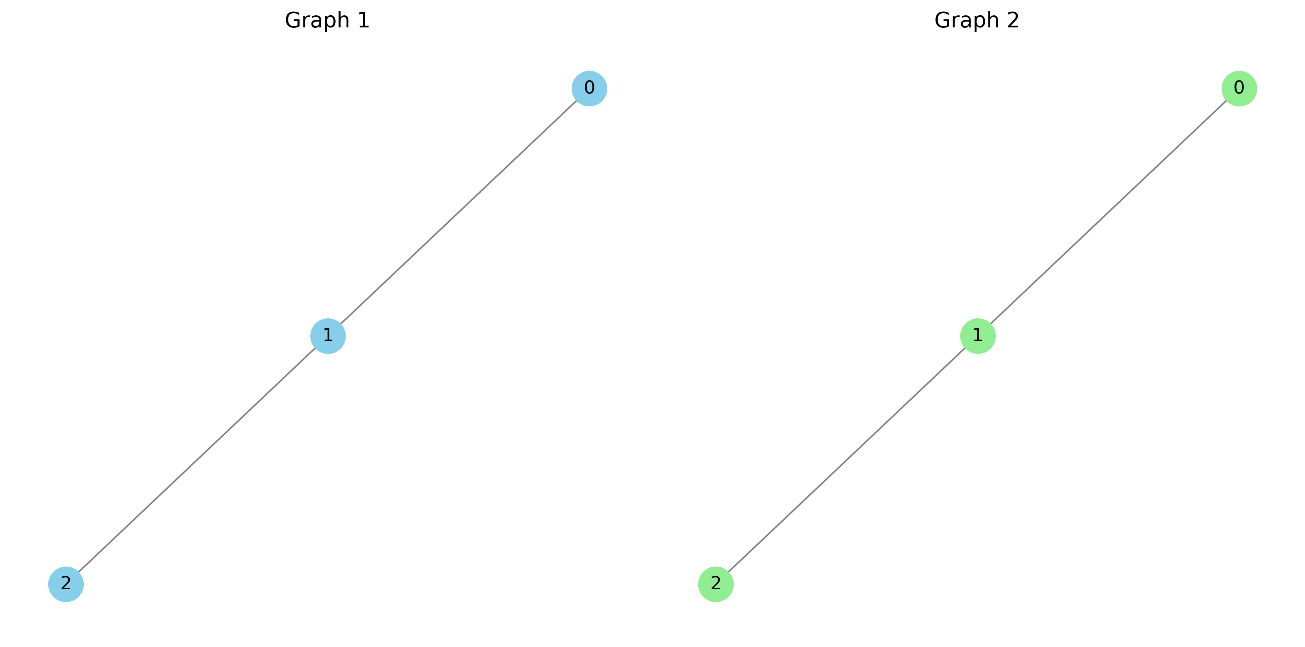


Рисунок 2. Две цепи

* Вершины: 0,1,2
* Рёбра: (0,1),(1,2)

**Что делает VF2:**

1. depth=0 → кандидаты: [0,1,2]
2. depth=1 → compatible с вершиной 0 → [1,2]
3. depth=2 → compatible с предыдущими → только [2] подходит
4. Изоморфизм найден: 0→0, 1→1, 2→2

**Особенность:** порядок вершин важен, но алгоритм учитывает рёбра и отсеивает несовместимые варианты.

## Пример 3.

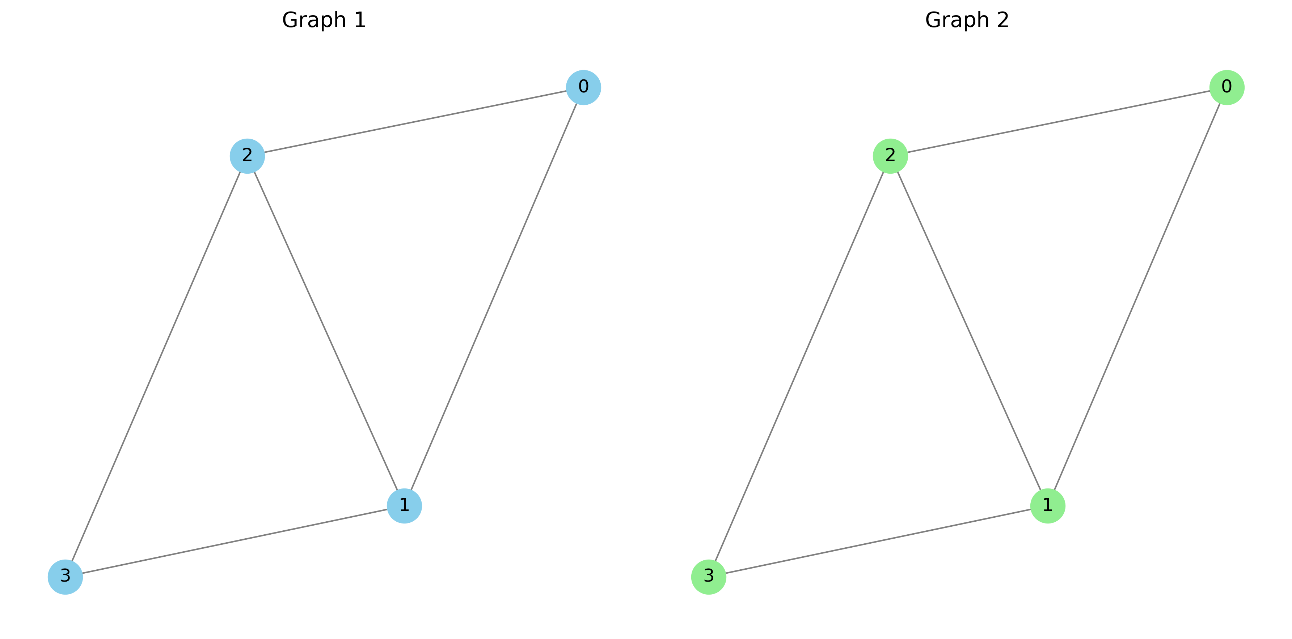


Рисунок 3. Два изоморфных графа

Пошаговое выполнение VF2

Начало: depth=0, mapping = [-1,-1,-1,-1]

1. depth=0 (v0 = 0 в G1):
   * Кандидаты в G2: [0,1,2,3]
   * Выбираем w=0 → mapping=[0,-1,-1,-1]
2. depth=1 (v1 = 1 в G1):
   * Проверяем совместимость с предыдущими вершинами:
     + mapping[0]=0
     + G1[0][1]=1, G2[0][?]
   * Кандидаты: [1,2] (только те, которые имеют рёбра, совпадающие с G1)
   * Выбираем w=1 → mapping=[0,1,-1,-1]
3. depth=2 (v2 = 2 в G1):
   * Совместимость с предыдущими:
     + G1[0][2]=1, G2[0][?]
     + G1[1][2]=1, G2[1][?]
   * Кандидаты: [2,3]
   * Выбираем w=2 → mapping=[0,1,2,-1]
4. depth=3 (v3 = 3 в G1):
   * Совместимость с предыдущими:
     + G1[0][3]=0, G2[0][?]
     + G1[1][3]=1, G2[1][?]
     + G1[2][3]=1, G2[2][?]
   * Только w=3 подходит → mapping=[0,1,2,3]

Итог: найден изоморфизм:

# Интерфейс приложения

Программа состоит из двух компонентов: основной модуль на **C++** для проверки изоморфизма графов и модуль визуализации на **Python** с использованием библиотеки **NetworkX**.

## Основной модуль (C++)

Программа на C++ работает через **файловый интерфейс**:

1. **Входные данные** подаются через текстовый файл input.txt в следующем формате:

n

m1

u1 v1

u2 v2

...

m2

u1 v1

u2 v2

...

n — количество вершин в каждом графе. m1 и m2 — количество рёбер в первом и втором графах соответственно. Каждая строка u v описывает ребро между вершинами u и v (нумерация с нуля).

1. **Выходные данные** записываются в файл output.txt. Возможны два варианта:

Если графы не изоморфны: Graphs are not isomorphic. Если графы изоморфны, выводится изоморфизм: Graphs are isomorphic. Isomorphism:

0 -> 2

1 -> 0

2 -> 1

3 -> 3

## ****Модуль визуализации (Python)****

Для наглядного анализа графов используется скрипт на Python, который считывает графы из того же файла input.txt и строит их визуализацию:

* + 1. **Считывание графов:** функция read\_graphs(filename) создаёт два объекта networkx.Graph для дальнейшего отображения.
    2. **Визуализация графов:** функция plot\_graphs(G1, G2) строит два графа на одной фигуре с помощью Matplotlib:
  + Граф 1 отображается слева (синие вершины).
  + Граф 2 отображается справа (зелёные вершины).
  + Визуализация сохраняется в файл graph\_visualization.png.

**Преимущества:**

* + Позволяет пользователю визуально проверить корректность построения графов и соответствие структуры при изоморфизме.
  + Облегчает анализ сложных графов и демонстрацию работы алгоритма.

# Заключение

В ходе выполнения данной лабораторной работы была реализована программа для проверки изоморфизма двух неориентированных графов с использованием **итеративной версии алгоритма VF2**. Работа включала изучение теоретических основ задачи изоморфизма графов, алгоритма VF2, а также практическую реализацию на языке **C++** и визуализацию графов на языке **Python** с применением библиотеки **NetworkX**.

Несмотря на экспоненциальную сложность алгоритма в худшем случае, применение правил согласованности значительно сокращает количество проверяемых вариантов. Ограничения реализации связаны с ростом числа вершин: при больших графах (более 100–150 вершин) время выполнения может резко увеличиваться. Для практических задач с графами среднего размера данная реализация является эффективной и удобной.

* 1. Перспективы улучшения  
     Возможны следующие направления дальнейшего развития:
  + Оптимизация алгоритма с использованием эвристик для выбора следующей вершины, что может снизить число ветвлений.
  + Реализация многопоточной версии для ускорения работы на больших графах.
  + Добавление поддержки ориентированных и взвешенных графов, расширяющее практическое применение алгоритма.
  + Улучшение визуализации, включая отображение найденного изоморфизма и цветовое выделение сопоставленных вершин.

# Приложение А

#include <iostream>

#include <vector>

#include <stack>

#include <fstream>

#include <string>

*using* *namespace* std;

***/\*\****

***\**** @brief ***The*** ***State*** ***struct***

***\****

***\**** ***Структура*** ***состояния*** ***для*** ***итеративного*** ***обхода***

***\*/***

*struct* **State** {

*//* *текущая* *глубина*

int depth;

*//* *соответствие* *вершин* *G1* *->* *G2*

vector<int> mapping;

*//* *допустимые* *кандидаты* *для* *текущей* *вершины*

vector<int> candidates;

*//* *индекс* *текущего* *кандидата*

int current\_candidate;

};

***/\*\****

***\**** @brief ***read\_graph\_from\_file***

***\**** @param ***file***

***\**** @param ***n***

***\**** @return ***graph***

***\****

***\**** ***Функция*** ***чтения*** ***графа*** ***из*** ***файла*** ***(неориентированный*** ***граф)***

***\**** ***Формат:*** ***m*** ***(число*** ***рёбер),*** ***затем*** ***m*** ***строк*** ***вида*** ***"u*** ***v"***

***\*/***

vector<vector<int>> **read\_graph\_from\_file**(ifstream& file, int n) {

vector<vector<int>> graph(n, vector<int>(n, 0));

int m;

file >> m;

*for* (int i = 0; i < m; i++) {

int u, v;

file >> u >> v;

graph[u][v] = 1;

graph[v][u] = 1;

}

*return* graph;

}

***/\*\****

***\**** @brief ***generate\_candidates***

***\**** @param ***depth***

***\**** @param ***mapping***

***\**** @param ***G1***

***\**** @param ***G2***

***\**** @return

***\****

***\**** ***Генерация*** ***списка*** ***возможных*** ***кандидатов*** ***для*** ***текущей*** ***вершины*** ***G1[depth]***

***\*/***

vector<int> **generate\_candidates**(

int depth,

*const* vector<int>& mapping,

*const* vector<vector<int>>& G1,

*const* vector<vector<int>>& G2

) {

vector<int> candidates;

int v = depth;

int n = G2.size();

vector<bool> used(n, *false*);

*//* *Помечаем* *уже* *сопоставленные* *вершины* *G2*

*for* (int i = 0; i < depth; i++) {

*if* (mapping[i] != -1) used[mapping[i]] = *true*;

}

*//* *Проверяем* *каждую* *вершину* *G2* *как* *возможного* *кандидата*

*for* (int w = 0; w < n; w++) {

*if* (used[w]) *continue*;

bool compatible = *true*;

*//* *Проверка* *согласованности* *рёбер* *с* *уже* *сопоставленными* *вершинами*

*for* (int u = 0; u < depth; u++) {

*if* (mapping[u] == -1) *continue*;

*if* (G1[u][v] != G2[mapping[u]][w]) {

compatible = *false*;

*break*;

}

}

*if* (compatible) {

candidates.push\_back(w);

}

}

*return* candidates;

}

***/\*\****

***\**** @brief ***vf2\_iterative***

***\**** @param ***G1***

***\**** @param ***G2***

***\**** @return

***\****

***\**** ***Итеративная*** ***реализация*** ***алгоритма*** ***VF2***

***\*/***

vector<int> **vf2\_iterative**(*const* vector<vector<int>>& G1, *const* vector<vector<int>>& G2) {

int n = G1.size();

*if* (n != G2.size()) {

*return* vector<int>();

}

stack<State> stk;

State initial;

initial.depth = 0;

initial.mapping = vector<int>(n, -1);

initial.candidates = generate\_candidates(initial.depth, initial.mapping, G1, G2);

initial.current\_candidate = 0;

stk.push(initial);

*while* (!stk.empty()) {

State current = stk.top();

stk.pop();

*//* *Все* *вершины* *сопоставлены* *—* *найден* *изоморфизм*

*if* (current.depth == n) {

*return* current.mapping;

}

*//* *Пробуем* *текущего* *кандидата*

*if* (current.current\_candidate < current.candidates.size()) {

int w = current.candidates[current.current\_candidate];

*//* *Создаём* *следующее* *состояние* *(углубляемся)*

State next = current;

next.depth = current.depth + 1;

next.mapping[current.depth] = w;

next.candidates = generate\_candidates(next.depth, next.mapping, G1, G2);

next.current\_candidate = 0;

*//* *Обновляем* *текущее* *состояние* *для* *следующего* *кандидата*

current.current\_candidate++;

*//* *Сохраняем* *состояния* *в* *стек*

stk.push(current); *//* *чтобы* *вернуться* *позже*

stk.push(next); *//* *идём* *глубже*

}

}

*//* *Изоморфизма* *нет*

*return* vector<int>();

}

***/\*\****

***\**** @brief ***main***

***\**** @return

***\*/***

int **main**() {

ifstream infile("D:\\Users\\fyt68\\Documents\\untitled\\input.txt");

ofstream outfile("D:\\Users\\fyt68\\Documents\\untitled\\output.txt");

*if* (!infile.is\_open()) {

cerr << "Error: cannot open input.txt" << endl;

*return* 1;

}

int n;

infile >> n;

vector<vector<int>> G1 = read\_graph\_from\_file(*infile*, n);

vector<vector<int>> G2 = read\_graph\_from\_file(*infile*, n);

infile.close();

***/\*\****

***\**** @brief ***isomorphism***

***\****

***\**** ***Проверяем,*** ***изоморфны*** ***ли*** ***графы***

***\*/***

vector<int> isomorphism = vf2\_iterative(G1, G2);

*if* (isomorphism.empty()) {

outfile << "Graphs are not isomorphic." << endl;

cout << "Graphs are not isomorphic. (see output.txt)" << endl;

} *else* {

outfile << "Graphs are isomorphic.\nIsomorphism:\n";

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

outfile << i << " -> " << isomorphism[i] << endl;

}

cout << "Graphs are isomorphic. (see output.txt)" << endl;

}

outfile.close();

*return* 0;

}

# Приложение Б

import networkx as nx

import matplotlib.pyplot as plt

def read\_graphs(filename):

    with open(filename, 'r') as f:

        n = int(f.readline().strip())

        m1 = int(f.readline().strip())

        G1 = nx.Graph()

        G1.add\_nodes\_from(range(n))

        for \_ in range(m1):

            u, v = map(int, f.readline().split())

            G1.add\_edge(u, v)

        m2 = int(f.readline().strip())

        G2 = nx.Graph()

        G2.add\_nodes\_from(range(n))

        for \_ in range(m2):

            u, v = map(int, f.readline().split())

            G2.add\_edge(u, v)

        return G1, G2

def plot\_graphs(G1, G2):

    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)

    pos1 = nx.spring\_layout(G1, seed=42)

    nx.draw(G1, pos1, with\_labels=True, node\_color='skyblue',

            node\_size=500, edge\_color='gray', font\_size=12)

    plt.title("Graph 1", fontsize=14)

    plt.subplot(1, 2, 2)

    pos2 = nx.spring\_layout(G2, seed=42)

    nx.draw(G2, pos2, with\_labels=True, node\_color='lightgreen',

            node\_size=500, edge\_color='gray', font\_size=12)

    plt.title("Graph 2", fontsize=14)

    plt.tight\_layout()

    plt.savefig("graph\_visualization.png", dpi=300, bbox\_inches='tight')

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    G1, G2 = read\_graphs("input.txt")

    plot\_graphs(G1, G2)