**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе № 4**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: **«Графы»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 | Кудрин П. В. |  |
| Преподаватель | Манирагена В. |  |

Санкт-Петербург

2024

**Цель работы**

Целью данной работы является разработка и реализация алгоритма для нахождения минимального вершинного покрытия в двудольном неориентированном графе. Работа включает в себя изучение свойств двудольных графов, применение теоретических основ теории графов, а также использование известных алгоритмов (например, алгоритма Хопкрофта-Карпа для поиска максимального паросочетания) для эффективного нахождения решения задачи вершинного покрытия.

**Текст индивидуального задания**

|  |  |
| --- | --- |
| № варианта | Алгоритм для исследования |
| 33 | Получение минимального вершинного покрытия для двудольного неориентированного графа |

**Математическая формулировка задачи в терминах теории множеств:**

Пусть G=(V,E) — двудольный неориентированный граф, где V — множество вершин, а — множество рёбер, соединяющих вершины. Граф G является двудольным, если его множество вершин V можно разделить на два непересекающихся подмножества U и W, таких что:

Причём каждое ребро соединяет вершину из U с вершиной из W.

Задача минимального вершинного покрытия заключается в следующем:

Найти подмножество (называемое вершинным покрытием), такое что:

Каждое ребро инцидентно хотя бы одной вершине из C (т.е. хотя бы одна из вершин принадлежит C).

Мощность множества C минимальна, т.е.:

Таким образом, требуется минимизировать количество вершин в покрытии при условии, что каждое ребро графа G имеет хотя бы одну вершину в C.

**Выбор и обоснование способа представления данных**

Для решения задачи получения минимального вершинного покрытия двудольного неориентированного графа было выбрано представление графа в виде списков смежности. Этот способ представления данных является наиболее подходящим по следующим причинам:

1. Компактность хранения:

Списки смежности позволяют эффективно хранить структуру графа, так как память выделяется только для существующих рёбер. Это особенно важно для разреженных графов, где число рёбер существенно меньше, чем число возможных связей между вершинами.

2. Удобство обработки:

При использовании списков смежности операции перебора смежных вершин выполняются за время, пропорциональное числу смежных вершин, что ускоряет работу алгоритмов, использующих обход графа (например, поиск в глубину или ширину).

3. Совместимость с алгоритмом нахождения максимального паросочетания:

Алгоритм нахождения максимального паросочетания (например, алгоритм Куна) удобно реализуется на графах, представленных в виде списков смежности, поскольку обход рёбер осуществляется последовательно для каждой вершины.

4. Математическая наглядность:

Списки смежности отражают структуру двудольного графа: каждая вершина из одной доли напрямую связана только с вершинами другой доли. Такое представление легко проверять на соответствие двудольным свойствам.

5. Гибкость в работе с данными:

Списки смежности позволяют легко модифицировать граф (добавлять и удалять рёбра), что упрощает тестирование различных алгоритмов и методов оптимизации.

Таким образом, выбранный способ представления данных — списки смежности — обеспечивает баланс между удобством реализации алгоритмов и эффективностью работы с данными.

**Описание алгоритма и оценка его временной сложности**

Предварительные условия:

Имеется двудольный граф G = (U, V, E), где U и V — доли вершин, а E — набор рёбер, соединяющих вершины из U с вершинами из V. Предполагается, что граф уже проверен на двудольность (например, с помощью обхода в ширину BFS, как показано в коде). Для удобства считаем, что цвет color[u] = 1 обозначает вершину из левой части U, а color[u] = -1 — из правой части V.

Основная идея:

Согласно теореме Кёнига, в двудольном графе размер максимального паросочетания равен размеру минимального вершинного покрытия. Значит, найдя максимальное паросочетание, мы можем получить минимальное вершинное покрытие.

Алгоритм Куна для максимального паросочетания основан на поиске увеличивающих путей с помощью итеративного запуска DFS. После нахождения максимального паросочетания мы используем стандартную процедуру для получения минимального вершинного покрытия.

Шаги алгоритма:

Построение графа в виде списков смежности:

Для хранения графа используются списки смежности. Это даёт возможность эффективно перебирать все рёбра, исходящие из каждой вершины. Данная операция занимает O(|E|).

Проверка двудольности графа (при необходимости):

С помощью обхода в ширину (BFS) или глубину (DFS) проверяется, можно ли окрасить все вершины в два цвета так, чтобы рёбра шли только между раз differently окрашенными вершинами. Это даёт нам разбиение (U, V). Проверка двудольности занимает O(|V| + |E|).

Поиск максимального паросочетания (алгоритм Куна): Алгоритм Куна работает следующим образом:

Изначально все вершины правой части свободны (не сопоставлены с вершинами левой части).

Для каждой вершины u из левой доли U мы пытаемся найти увеличивающий путь.

Увеличивающий путь — это чередующаяся последовательность рёбер, начинающаяся и заканчивающаяся свободной вершиной, причём рёбра поочерёдно принадлежат и не принадлежат текущему паросочетанию.

Для поиска увеличивающего пути для конкретной вершины u из U:

Выполняется DFS, который пытается "присоединить" u к какой-либо свободной вершине v из V.

Если v уже занята, но её "пару" из U можно переподключить к другой вершине из V (то есть найти альтернативный увеличивающий путь в глубину рекурсивно), то мы "смещаем" соответствия, освобождая v для u.

Если удалось найти путь, увеличиваем размер паросочетания на 1.

Каждый вызов DFS работает за O(|E|) в худшем случае, а вызовем его максимум |V| раз (для каждой вершины левой доли). Таким образом, оценка работы алгоритма Куна — O(|V|⋅|E|).

Определение множества достижимых вершин для выделения минимального вершинного покрытия:

После получения максимального паросочетания М мы находим множество достижимых вершин Z следующим образом:

Рассматриваем все вершины левой доли, которые не покрыты паросочетанием (свободные вершины из U).

Запускаем обход (BFS или DFS) по следующему правилу:

Из левой доли U в правую долю V идём только по рёбрам, не входящим в паросочетание.

Из правой доли V в левую долю U идём только по рёбрам, входящим в паросочетание.

В результате мы получим набор достижимых вершин Z, помеченный после этого обхода. Этот этап занимает O(|V| + |E|).

Формирование минимального вершинного покрытия:

Минимальное вершинное покрытие по теореме Кёнига задаётся так:

Возьмём все непомеченные вершины из U (U \ Z).

Возьмём все помеченные вершины из V (V ∩ Z).

Эти вершины в сумме дают множество минимального вершинного покрытия. Определение помеченных и непомеченных вершин требует O(|V|).

Вывод результата:

Возвращаем найденное минимальное вершинное покрытие.

Оценка временной сложности

Построение графа:

O(|E|).

Проверка двудольности:

O(|V| + |E|), но эта операция выполняется однократно и не превосходит по сложности основного этапа.

Поиск максимального паросочетания (алгоритм Куна):

Алгоритм Куна имеет временную сложность O(|V|⋅|E|) в худшем случае.

Обход графа для определения достижимых вершин:

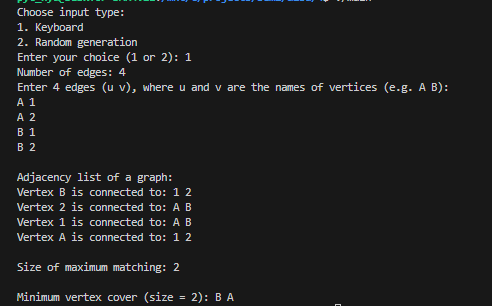
O(|V| + |E|).

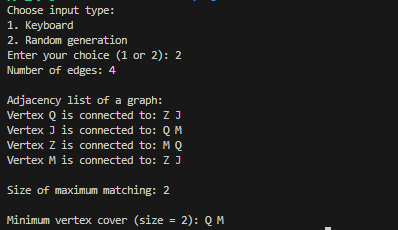
Формирование вершинного покрытия:

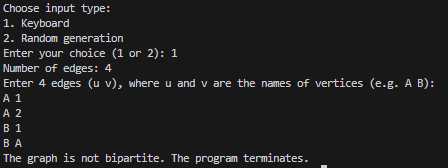
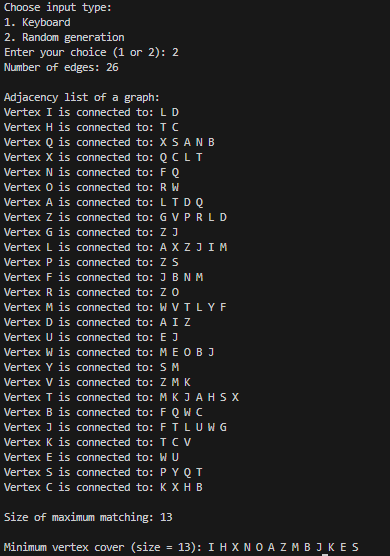
O(|V|).

Основная часть времени уходит на поиск максимального паросочетания алгоритмом Куна, дающего асимптотику O(|V|⋅|E|).

**Набор тестов и результаты проверки алгоритма на ЭВМ:**







**Выводы**

В данной работе была решена задача получения минимального вершинного покрытия для двудольного неориентированного графа с использованием теоретических результатов и эффективных алгоритмов из теории графов. На основе проделанной работы можно сделать следующие выводы:

1. *Связь между минимальным вершинным покрытием и максимальным паросочетанием:*

Теорема Кёнига позволяет эффективно находить минимальное вершинное покрытие в двудольном графе, используя максимальное паросочетание. Эта теоретическая основа была успешно применена в реализации алгоритма.

1. *Алгоритм Куна:*

Для нахождения максимального паросочетания использовался алгоритм Куна, который имеет временную сложность . Этот алгоритм является оптимальным для задач поиска паросочетаний в двудольных графах и обеспечивает высокую производительность даже на больших входных данных.

1. *Эффективность предложенного подхода:*

За счёт связи задач и использования эффективного алгоритма максимального паросочетания была достигнута временная сложность , что делает предложенный подход применимым для больших графов с большим количеством рёбер и вершин.

1. *Практическая применимость:*

Решение задачи минимального вершинного покрытия имеет широкое применение в реальных задачах, таких как оптимизация сетей, распределение ресурсов и задачи планирования.

Таким образом, задача была успешно решена как с теоретической, так и с практической точки зрения. Разработанный алгоритм демонстрирует высокую эффективность и точность, что подтверждает его пригодность для применения в различных областях.

**Список использованных источников**

1. Методичка, выданная в начале 3 учебного семестра преподавателем
2. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — М.: Вильямс, 2013. — 1296 с.
3. Хопкрофт Дж., Карп Р. Алгоритм для задачи о максимальном паросочетании в двудольных графах. // SIAM Journal on Computing. — 1973. — Т. 2. — №4. — С. 225–231.
4. Коноваленко В. И., Ушаков И. А. Теория графов и ее приложения. — М.: Высшая школа, 2020. — 352 с.
5. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. — М.: Мир, 1983. — 432 с.
6. Bondy J. A., Murty U. S. R. Graph Theory with Applications. — London: Macmillan, 1976. — 264 p.
7. Tarjan R. E. Depth-First Search and Linear Graph Algorithms. // SIAM Journal on Computing. — 1972. — Т. 1. — №2. — С. 146–160.
8. West D. B. Introduction to Graph Theory. — Upper Saddle River: Prentice Hall, 2001. — 512 p.
9. Wikipedia. Vertex Cover Problem. [Электронный ресурс] — Режим доступа: https://en.wikipedia.org/wiki/Vertex\_cover, свободный.
10. Skiena S. S. The Algorithm Design Manual. — London: Springer, 2008. — 730 p.
11. Even S. Graph Algorithms. — Cambridge: Cambridge University Press, 2011. — 198 p.

**Исходный текст программы для ЭВМ**

#include <iostream> // Подключаем библиотеку ввода-вывода (используем cout, cin)

#include <unordered\_map> // Подключаем хеш-таблицу (ассоциативный массив с доступом по ключу, как map, но быстрее)

#include <vector> // Подключаем вектор (динамический массив)

#include <unordered\_set> // Подключаем хеш-множество (уникальные элементы, быстрый поиск)

#include <cstdlib> // Подключаем для функций rand, srand

#include <ctime> // Подключаем для функции time(), нужна для инициализации rand

#include <queue> // Подключаем очередь FIFO, нужна для обхода в ширину (BFS)

#include <string> // Подключаем для работы со строками std::string

#include <fstream> // Подключаем для записи в файл (ofstream)

using namespace std; // Чтобы не писать std:: перед cout, cin и т.д.

class Graph {

private:

// adjList — это "список смежности".

// Это способ хранить граф: для каждой вершины мы храним список всех соседей (вершин, с которыми у неё есть ребро).

// Ключ (string) — имя вершины,

// Значение (vector<string>) — список вершин, соединённых с этой вершиной ребром.

unordered\_map<string, vector<string>> adjList;

// color — здесь мы храним цвет каждой вершины после проверки двудольности.

// color[u] = 1, если вершина "u" принадлежит левой части,

// color[u] = -1, если принадлежит правой части,

// color[u] = 0, если цвет пока не назначен.

unordered\_map<string,int> color;

// matchR — хранит для каждой вершины правой доли (color[u] = -1) ту вершину из левой доли, с которой она "спарена" в текущем паросочетании.

// Если matchR[v] = u, значит есть ребро в паросочетании (u,v).

// Если matchR[v] = "", значит вершина v справа свободна, не участвует в паросочетании.

unordered\_map<string,string> matchR;

public:

// addEdge(u, v) — добавляет ребро между вершинами u и v.

// Так как граф неориентированный, мы добавляем v в список соседей u и u в список соседей v.

void addEdge(const string& u, const string& v) {

adjList[u].push\_back(v);

adjList[v].push\_back(u);

}

// isBipartite() — проверяем, является ли граф двудольным.

// Двудольный граф — это такой, в котором вершины можно разделить на 2 группы так,

// что ребра идут только между группами, а внутри группы ребер нет.

// Мы красим вершины в два цвета: 1 и -1.

// Если найдём ребро, соединяющее вершины одного цвета, граф не двудольный.

bool isBipartite() {

// Сначала для всех вершин ставим цвет 0 (неокрашены)

for (auto &pair : adjList) {

color[pair.first] = 0;

}

// Обходим все вершины, так как граф может быть не связным (из нескольких частей).

for (auto &vertex : adjList) {

// Если вершина ещё не окрашена:

if (color[vertex.first] == 0) {

// Начнём обход в ширину (BFS) с этой вершины

queue<string> q;

q.push(vertex.first);

// Красим стартовую вершину в цвет 1

color[vertex.first] = 1;

// Пока очередь не пуста, достаем по одной вершине

while(!q.empty()) {

string u = q.front();

q.pop();

// Смотрим всех соседей вершины u

for (auto &v : adjList[u]) {

if (color[v] == 0) {

// Если сосед не окрашен, красим его в противоположный цвет от u

color[v] = -color[u];

q.push(v);

} else {

// Если сосед уже окрашен, проверяем, совпадает ли цвет.

// Если совпадает — плохой знак, граф не двудольный.

if (color[v] == color[u]) {

return false;

}

}

}

}

}

}

// Если мы не нашли проблем, значит граф двудольный

return true;

}

// inputEdges — ввод рёбер с клавиатуры.

// Пользователь вводит количество рёбер, затем пары вершин, между которыми есть ребро.

void inputEdges(int edgeCount) {

cout << "Enter " << edgeCount << " edges (u v), where u and v are the names of vertices (e.g. A B):\n";

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

string u, v;

cin >> u >> v; // Читаем имена двух вершин

addEdge(u, v); // Добавляем ребро

}

}

// generateRandomBipartiteGraph — случайно генерирует двудольный граф.

// Идея: сначала генерируем набор вершин, делим их на две части (левую и правую),

// а затем случайным образом создаём рёбра только между этими двумя частями.

void generateRandomBipartiteGraph(int vertexCount) {

vector<string> vertices;

unordered\_set<string> uniqueVertices;

srand((unsigned)time(nullptr)); // Инициализируем генератор случайных чисел текущим временем

// Генерируем уникальные вершины, имена из одной буквы A-Z, пока не наберём vertexCount штук.

while ((int)uniqueVertices.size() < vertexCount) {

string vertex;

vertex += static\_cast<char>('A' + rand() % 26);

uniqueVertices.insert(vertex);

}

// Перекладываем вершины из множества в вектор, чтобы удобно было их индексировать

for (const auto& v : uniqueVertices) {

vertices.push\_back(v);

}

// Разбиваем вершины на две части: первые half — левая часть, остальные — правая.

int half = vertexCount / 2;

vector<string> leftPart(vertices.begin(), vertices.begin() + half);

vector<string> rightPart(vertices.begin() + half, vertices.end());

// Максимальное число рёбер между двумя частями — просто произведение размеров.

int maxPossibleEdges = (int)leftPart.size() \* (int)rightPart.size();

if (maxPossibleEdges == 0) {

// Если одна из частей пуста, рёбер быть не может.

return;

}

// Выбираем случайное число рёбер от half до maxPossibleEdges

int edgeCount = half + rand() % maxPossibleEdges;

unordered\_set<string> uniqueEdges;

// Добавляем рёбра, пока не наберём edgeCount штук

while ((int)uniqueEdges.size() < edgeCount) {

int i = rand() % leftPart.size();

int j = rand() % rightPart.size();

string u = leftPart[i];

string v = rightPart[j];

// Чтобы не добавить одно и то же ребро дважды (u-v и v-u это одно и то же ребро),

// создаём ключ edgeKey, отсортировав имена вершин по алфавиту.

string edgeKey = (u < v) ? (u + "-" + v) : (v + "-" + u);

if (uniqueEdges.find(edgeKey) == uniqueEdges.end()) {

addEdge(u, v);

uniqueEdges.insert(edgeKey);

}

}

}

// printGraph — печатает список смежности.

// Показывает для каждой вершины, с кем она соединена.

void printGraph() const {

cout << "\nAdjacency list of a graph:\n";

for (const auto& pair : adjList) {

cout << "Vertex " << pair.first << " is connected to: ";

for (const auto& neighbor : pair.second) {

cout << neighbor << " ";

}

cout << endl;

}

}

// bpmDfs — функция для поиска "увеличивающего пути" в контексте алгоритма Куна,

// который используется для нахождения максимального паросочетания в двудольном графе.

// u — вершина левой доли, с которой мы пытаемся "увеличить" паросочетание.

// used — отмечаем вершины правой доли, которые мы уже посетили при поиске пути, чтобы не зациклиться.

// смысл метода в том, что мы берём вершину из доли U и пробуем поставить ей в соответствие какую-то вершину из доли V,

// которая не была затронута в нашем паросочетании ранее.

// выполняя проверку для u раз за разом мы можем вычислить количество рёбер в текущем паросочетании

bool bpmDfs(const string &u, unordered\_map<string,bool> &used) {

// Перебираем всех соседей вершины u

for (auto &v : adjList[u]) {

// Рассматриваем только рёбра из левой доли (color[u] = 1) в правую долю (color[v] = -1).

// Это часть алгоритма поиска увеличивающего пути.

// if (color[v] == -1) { // данная проверка закомментирована, поскольку является избыточной, т к adjList[u] содержит вершины только из правой доли

// Если вершина v правой доли ещё не посещена в этом поиске

if (!used[v]) {

used[v] = true; // Помечаем, что посетили v

// Если v сейчас свободна (matchR[v] == "") или

// если занята, но мы можем подвинуть u\_i = matchR[v] вершину и переназначить ей другую вершину из V,

// то присваиваем matchR[v] = u, тем самым увеличивая паросочетание.

if (matchR[v] == "" || bpmDfs(matchR[v], used)) {

matchR[v] = u;

return true;

}

}

// }

}

return false; // Не смогли найти увеличивающий путь для u

}

// maxMatching — находит максимальное паросочетание в двудольном графе, используя алгоритм Куна.

// смысл метода в том, что мы буквально перебираем все вершины левой доли и применяем для них метод выше

// первую вершину мы берём как отправную точку и начинаем строить ребра из неё соседей из adjList[v], тем самым, увеличивая result на 1,

// если удалось создать ребро

// теперь о том, почему не важен порядок, т е отправная точка

// в функции выше мы имеем возможность переставлять вершины (и более того, делаем это рекурсивно!), так что мы имеем право выбрать

// в качестве отправной точки любую вершину левой доли, поскольку функция bpmDfs все равно будет перставлять наши вершины в мапе matchR,

// поскольку именно она играет здесь ключевую роль

int maxMatching() {

// Очищаем matchR и изначально все вершины правой доли свободны ("")

matchR.clear();

for (auto &pair : adjList) {

if (color[pair.first] == -1) {

matchR[pair.first] = "";

}

}

int result = 0;

// Для каждой вершины левой доли пытаемся найти увеличивающий путь

// Если нашли — увеличиваем счётчик result

for (auto &pair : adjList) {

if (color[pair.first] == 1) {

unordered\_map<string,bool> used;

if (bpmDfs(pair.first, used)) {

result++;

}

}

}

return result; // Возвращаем размер максимального паросочетания

}

void findMinVertexCover() {

vector<string> leftVertices;

vector<string> rightVertices;

// Разделяем вершины на левую (U) и правую (V) доли по их цвету:

// color[u] = 1 для левой доли, color[u] = -1 для правой.

for (auto &pair : adjList) {

if (color[pair.first] == 1) {

leftVertices.push\_back(pair.first);

} else if (color[pair.first] == -1) {

rightVertices.push\_back(pair.first);

}

}

// coveredLeft — множество вершин левой доли, которые покрыты паросочетанием.

// Если вершина левой доли u покрыта, значит существует вершина v правой доли такая, что matchR[v] = u.

// matchR[v] = u означает, что ребро (u,v) входит в паросочетание.

unordered\_set<string> coveredLeft;

for (auto &m : matchR) {

// m.first — вершина правой доли, m.second — вершина левой доли, с которой она связана паросочетанием.

if (m.second != "") {

// Значит у нас есть ребро паросочетания (m.second, m.first).

coveredLeft.insert(m.second);

}

}

// freeLeft — вершины левой доли, НЕ покрытые паросочетанием.

// Это те вершины левой доли, которые не входят ни в одно ребро паросочетания.

vector<string> freeLeft;

for (auto &u : leftVertices) {

// Если u не найдена в coveredLeft, значит она свободна (не покрыта паросочетанием).

if (coveredLeft.find(u) == coveredLeft.end()) {

freeLeft.push\_back(u);

}

}

// visited — структура, где будем отмечать вершины, которых мы "достигли" при специальном обходе.

// Мы хотим определить множество достижимых вершин Z.

// Начинаем обход с freeLeft — свободных вершин левой доли.

unordered\_map<string,bool> visited;

queue<string> q;

// Помечаем все свободные вершины левой доли как посещённые и ставим их в очередь.

// Эти вершины — стартовые точки нашего специального обхода.

for (auto &u : freeLeft) {

visited[u] = true;

q.push(u);

}

// Теперь основной цикл обхода.

// Идея:

// - Из левой доли (U) в правую долю (V) идём по рёбрам, НЕ принадлежащим паросочетанию.

// - Из правой доли (V) в левую долю (U) идём по рёбрам, принадлежащим паросочетанию.

// Почему так?

// Это стандартная процедура, которая позволяет найти множество достижимых вершин Z,

// используя структуру уже найденного максимального паросочетания.

while(!q.empty()) {

string u = q.front();

q.pop();

if (color[u] == 1) {

// Если мы находимся в вершине левой доли:

// Идём только в вершины правой доли по рёбрам, которые НЕ входят в паросочетание.

for (auto &v : adjList[u]) {

// if (color[v] == -1) {

// Проверим, входит ли ребро (u,v) в паросочетание.

// Если matchR[v] == u, значит (u,v) — паросочетание.

// Нам же нужны рёбра, не входящие в паросочетание,

// следовательно, нам подходит случай, когда matchR[v] != u.

if (matchR[v] != u) {

// Если вершина v ещё не была посещена:

if (!visited[v]) {

visited[v] = true;

q.push(v);

// Добавляем v в очередь для продолжения обхода.

}

}

// }

}

} else {

// Если мы в вершине правой доли:

// Отсюда мы можем идти только по рёбрам паросочетания в левую долю.

// Т.е. находим вершину левой доли, которая связана с u в паросочетании.

// matchR[u] — это левая вершина, с которой связана правая вершина u, если такая есть.

// Если matchR[u] != "", значит у нас есть парное ребро (matchR[u], u) в паросочетании.

if (matchR.find(u) != matchR.end() && matchR[u] != "") {

string uPair = matchR[u]; // левая вершина, с которой u соединена паросочетанием

// Если эта левая вершина ещё не посещена:

if (!visited[uPair]) {

visited[uPair] = true;

q.push(uPair);

// Добавляем её в очередь, теперь будем рассматривать её соседей.

}

}

}

}

// После завершения обхода у нас есть множество достижимых вершин Z: это все вершины, для которых visited[...] = true.

// По теореме Кёнига:

// Минимальное вершинное покрытие = (U \ Z) ∪ (V ∩ Z),

// где U — множество всех вершин левой доли, V — множество всех вершин правой доли,

// Z — множество достижимых вершин.

//

// U \ Z — вершины левой доли, которые НЕ достижимы.

// V ∩ Z — вершины правой доли, которые достижимы.

vector<string> minCover;

// Добавляем в минимальное покрытие все вершины левой доли, которые НЕ достижимы:

for (auto &u : leftVertices) {

if (!visited[u]) {

minCover.push\_back(u);

}

}

// Добавляем в минимальное покрытие все вершины правой доли, которые ДОСТИЖИМЫ:

for (auto &v : rightVertices) {

if (visited[v]) {

minCover.push\_back(v);

}

}

// Выводим результат:

cout << "\nMinimum vertex cover (size = " << minCover.size() << "): ";

for (auto &vert : minCover) {

cout << vert << " ";

}

cout << endl;

}

// generateDotFile — создаём файл в формате DOT (Graphviz),

// чтобы потом визуализировать наш граф.

// rankdir=LR; — расположим вершины слева направо.

// Далее рисуем вершины и ребра.

// Цвета: синие круги для левой доли, красные квадраты для правой.

void generateDotFile(const string &filename) {

ofstream out(filename); // Открываем файл для записи

out << "graph G {\n";

out << "rankdir=LR;\n";

out << "node [fontname=\"Arial\"];\n";

// Выводим вершины

for (auto &p : adjList) {

if (color[p.first] == 1) {

// Левая доля — синий круг

out << p.first << " [shape=circle, color=blue];\n";

} else if (color[p.first] == -1) {

// Правая доля — красный квадрат

out << p.first << " [shape=box, color=red];\n";

} else {

// Если вдруг есть вершина без цвета (не должно быть), рисуем чёрный овал

out << p.first << " [shape=ellipse, color=black];\n";

}

}

// Выводим рёбра

unordered\_set<string> printedEdges;

for (auto &pr : adjList) {

for (auto &nbr : pr.second) {

// Формируем ключ ребра e1

string e1 = pr.first < nbr ? pr.first + "--" + nbr : nbr + "--" + pr.first;

// Чтобы не напечатать ребро дважды, проверяем через множество printedEdges

if (printedEdges.find(e1) == printedEdges.end()) {

printedEdges.insert(e1);

out << pr.first << " -- " << nbr << ";\n";

}

}

}

out << "}\n";

out.close(); // Закрываем файл

}

};

int main() {

int choice;

cout << "Choose input type:\n";

cout << "1. Keyboard\n";

cout << "2. Random generation\n";

cout << "Enter your choice (1 or 2): ";

cin >> choice;

Graph graph;

if (choice == 1) {

// Если пользователь хочет вводить рёбра сам

int edgeCount;

cout << "Number of edges: ";

cin >> edgeCount;

graph.inputEdges(edgeCount); // Вызываем ввод рёбер

// Проверяем, двудольный ли граф

if (!graph.isBipartite()) {

cout << "The graph is not bipartite. The program terminates." << endl;

return 1; // Завершаем программу с кодом ошибки

}

} else if (choice == 2) {

// Случайная генерация

int vertexCount;

cout << "Number of edges: ";

cin >> vertexCount;

graph.generateRandomBipartiteGraph(vertexCount);

// Проверяем двудольность сгенерированного графа

if (!graph.isBipartite()) {

cout << "Error: The generated graph is not bipartite (shouldn't happen)." << endl;

return 1;

}

} else {

// Если пользователь ввёл не 1 и не 2

cout << "Incorrect choice.\n";

return 1;

}

// Печатаем список смежности

graph.printGraph();

// Находим максимальное паросочетание

int matchingSize = graph.maxMatching();

cout << "\nSize of maximum matching: " << matchingSize << endl;

// Находим минимальное вершинное покрытие

graph.findMinVertexCover();

// Создаём dot-файл для визуализации

graph.generateDotFile("graph.dot");

// Запускаем систему с командой "dot -Tpng graph.dot -o graph.png"

// Эта команда использует Graphviz для преобразования dot в png картинку

system("dot -Tpng graph.dot -o graph.png");

return 0; // Программа успешно завершена

}