МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

ОТЧЕТ

по курсовой работе

по дисциплине «Алгоритмы и структуры данных»

Тема: «Графы»

|  |  |  |
| --- | --- | --- |
| Студенты гр. 3311 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Сапронов К. Д.  Землякова С. А. |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Преподаватель |  | Манирагена Валенс |

Санкт-Петербург

2024

**Алгоритм для исследования:** получение множества компонент двусвязности для произвольного неориентированного графа.

По заданию требовалось реализовать алгоритм поиска двусвязных компонент в неориентированном графе с представлением графа в виде списка смежности. Также предусмотреть возможность ручного ввода и случайной генерации графа.

**Математическая формулировка задачи в терминах теории множеств**

Граф G описывается множествами: G = (V, E), где: V — множество вершин (V = {v1​, v2​, …, vn​}), E — множество рёбер (E ⊆ V×V).

В программе поддерживаются неориентированные графы с возможностью задания веса для рёбер. Если рёбра взвешенные, задаётся функция: w:E→R, определяющая вес w(e) для каждого ребра e ∈ E. Задача сводится к разбиению рёбер E на группы E1,E2,…,Ek​, где каждая группа соответствует компоненте, сохраняющей связность после удаления любой вершины.

**Модель графа**

Вершины V представлены в виде индексов от 0 до n−1, а рёбра E описываются через список смежности vector<list<int>> adjList.

Матрица рёбер vector<vector<int>> edgeMatrix дополнительно позволяет хранить веса рёбер w(e).

**Выбор и обоснование способа представления данных**

Для представления графа в программе использованы два способа: список смежности и матрица ребер.

Список смежности представляет граф в виде массива списков. Каждый элемент этого массива соответствует вершине графа и содержит список вершин, с которыми она соединена рёбрами. Например, если вершина *v* соединена с вершинами *u* и *w,* то adjList[*v*] будет содержать u и w.

Данный способ, во-первых, экономит память – список смежности занимает меньше места для разреженных графов. Поэтому хранятся не все возможные связи между вершинами, а только существующие ребра. Во-вторых, таким образом граф удобно обходить. При обходе графа (например, при поиске в глубину или ширину) требуется рассматривать только смежные вершины, которые хранятся в списке. Это позволяет обрабатывать граф за время O(∣V∣+∣E∣), что особенно важно для задач, где графы могут быть большими.

В-третьих, при таком хранении прост добавлять ребра. Для добавления нового ребра достаточно добавить элемент в список для обеих его конечных вершин, что реализуется за O(1).

Список смежности используется в программе для выполнения обхода графа и поиска двусвязных компонент. Это реализовано в функции DBL, которая исследует граф с помощью поиска в глубину (DFS).

Матрица ребер применяется в программе реже. Она представляет граф в виде квадратной таблицы (матрицы), где строки и столбцы соответствуют вершинам графа. Если между вершинами 𝑣𝑖 и 𝑣𝑗 существует ребро, то элемент adjMatrix[i][j] будет равен 1 (или весу ребра для взвешенного графа). Если рёбра нет, то adjMatrix[i][j] будет равен 0 (или бесконечности для взвешенного графа).

Этот способ был выбран для работы с графом по следующим причинам:

1. Быстрый доступ к информации о рёбрах. Матрица позволяет за время O(1) проверить наличие ребра между двумя вершинами или узнать его вес. Это удобно, например, для алгоритма Дейкстры, где многократно проверяются веса рёбер.
2. Матрица ребер в данной программе используется при поиске кратчайшего пути между двумя вершинами.

Матрица ребер применяется в программе для решения задачи нахождения кратчайшего пути с использованием алгоритма Дейкстры для взвешенного графа (если граф остается невзвешенным, то используется список смежности). Функция DijkstraShortestPath позволяет находить минимальный путь между двумя вершинами графа, используя веса рёбер из матрицы.

Таким образом, сочетание списка смежности и матрицы ребер делает программу более универсальной и позволяет эффективно решать разные типы задач, связанных с графами.

**Описание алгоритма и оценка его временной сложности**

1. **Алгоритм поиска двусвязных компонент (DBL)**

Описание:  
Алгоритм выполняется с использованием поиска в глубину (DFS). Он определяет узлы, удаление которых увеличивает количество компонент связности графа, и выделяет двусвязные компоненты – подграфы, в которых каждая пара вершин соединена хотя бы двумя путями, не имеющими общих рёбер.

Основные этапы:

1. Инициализация массивов NUM и L для всех вершин:

NUM[v] — время первого посещения вершины v в процессе обхода. L[v] — минимальное значение NUM, достижимое из вершины v с учётом обратных рёбер.

1. Выполнение DFS, начиная с произвольной вершины.

При посещении вершины:

а) если вершина u ещё не посещена, выполняется рекурсивный вызов для неё.

б) если L[u] ≥ NUM[v], текущая вершина является точкой сочленения. Ребро (v,u) закрывает компоненту, которая извлекается из стека edgeStack.

в) если u уже посещена, обновляется значение L[v].

После завершения работы алгоритма все двусвязные компоненты выделены.

**Временная сложность:** алгоритм выполняет обход DFS, посещая каждую вершину и ребро один раз. Для графа с ∣V∣ вершинами и ∣E∣ рёбрами временная сложность составляет O(∣V∣+∣E∣).

1. **Алгоритм Дейкстры для нахождения кратчайшего пути**

Описание**:** алгоритм находит кратчайший путь от одной вершины графа до другой с учётом весов рёбер.

Основные этапы:

1. Инициализация массива расстояний dist значением INT\_MAX для всех вершин, кроме стартовой (для неё расстояние равно 0). Массив prev используется для восстановления пути.
2. Выбор не посещённой вершины v с минимальным значением dist[v].
3. Обновление расстояний до всех соседей u вершины v, если новый путь через v короче текущего.

Этот процесс повторяется до тех пор, пока не будут посещены все вершины или не найдётся путь до конечной вершины. В программе алгоритм реализован с использованием списка смежности для соседей каждой вершины, а если граф взвешенный, веса хранятся в edgeMatrix.

**Временная сложность:**

Выбор вершины с минимальным расстоянием (линейный поиск) выполняется за время O(|V|^2). Для обновления расстояний алгоритм проходит по всем рёбрам за время O(|E|). Общая временная сложность: O(|V|^2) при использовании массива для хранения расстояний.

Тесты и результаты проверки алгоритма на ЭВМ.

1. Ручной ввод

****

1. **Случайная генерация**

**Выводы**

В ходе выполнения работы была разработана программа для работы с графами, которая включает алгоритмы поиска двусвязных компонент (DBL) и нахождения кратчайшего пути с использованием алгоритма Дейкстры.

Алгоритм поиска двусвязных компонент помог выявить ключевые элементы графа, такие как точки сочленения и двусвязные компоненты. Это важно для анализа структуры графа и поиска вершин, удаление которых может повлиять на его связность. Для реализации использован DFS, что позволяет достичь линейной сложности O(∣V∣+∣E∣), что эффективно для разреженных графов.

Алгоритм Дейкстры использовался для нахождения кратчайшего пути между двумя вершинами с учётом весов рёбер. Несмотря на свою временную сложность O(∣V∣²) при использовании массивов для хранения расстояний, он остаётся достаточно эффективным для графов средней и малой размерности.

Таким образом, реализованные алгоритмы и структуры данных (список смежности для хранения рёбер и матрица весов для графа) позволили эффективно работать с графами, решать задачи нахождения кратчайших путей и анализа связности.

В целом, работа показала важность алгоритмов теории графов для решения реальных задач и продемонстрировала, как их можно эффективно реализовывать с использованием базовых структур данных.

**Текст программы**

#include <iostream>

#include <vector>

#include <list>

#include <stack>

#include <cstdlib>

#include <ctime>

#include <queue>

#include <climits>

#include <algorithm>

#include <iomanip>

using namespace std;

const int MaxV = 26; // Maximum number of vertices

// Convert integer index to corresponding character

char Ch(int c) { return c + 'a'; }

class Graph {

int num, n, m; // Counters for DFS numbers, vertices, and edges

vector<list<int>> adjList; // Adjacency list representation of the graph

vector<int> NUM, L; // Discovery times and lowest points

vector<vector<int>> edgeMatrix; // Edge weight matrix

stack<pair<int, int>> edgeStack; // Stack to track edges during DFS

void DBL(int v, int p); // Depth-first search for biconnected components

void ManualInput(); // Manual graph creation from user input

void Random(); // Random graph generation

public:

Graph(); // Constructor

void PrintAdjList() const; // Print adjacency list

void PrintMatrix() const; // Print adjacency matrix

void ExecuteDBL(); // Driver to start DFS on the graph

void CreateEdgeMatrix(); // Generate edge weight matrix

void PrintEdgeMatrix() const; // Print edge weight matrix

void DijkstraShortestPath(int start, int end, vector<int>& path, int& length) const; // Find shortest path

void FindShortestPath(); // Interactive shortest path finder

~Graph() = default; // Destructor

};

Graph::Graph() : num(0), n(0), m(0) {

int choice;

cout << "Choose an option:\n";

cout << "1. Manual input\n";

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

cin >> choice;

if (choice == 1) {

ManualInput();

} else if (choice == 2) {

Random();

}

}

void Graph::ManualInput() {

adjList.resize(MaxV);

string s;

cout << "\nEnter adjacency sets (lines with letters from 'a' to 'z'):\n";

while (true) {

cout << "v[" << Ch(n) << "]=";

cin >> s;

if (!isalpha(s[0])) break;

for (auto c : s) {

if (isalpha(c)) {

int j = tolower(c) - 'a';

// Проверка, чтобы избежать дублирования

if (find(adjList[n].begin(), adjList[n].end(), j) == adjList[n].end()) {

adjList[n].push\_back(j);

adjList[j].push\_back(n);

++m;

}

}

}

++n;

if (n >= MaxV) break;

}

m /= 2; // Each edge is counted twice

}

void Graph::Random() {

srand(time(0));

adjList.resize(MaxV / 2);

n = MaxV / 2;

m = 0;

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

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

if (static\_cast<double>(rand()) / RAND\_MAX < 0.5) {

adjList[i].push\_back(j);

adjList[j].push\_back(i);

++m;

}

}

}

}

void Graph::PrintAdjList() const {

cout << "Graph adjacency list:\n";

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

cout << Ch(i) << ": ";

if (adjList[i].empty()) {

cout << "-";

} else {

for (int j : adjList[i]) {

cout << Ch(j) << ' ';

}

}

cout << '\n';

}

cout << "\n|V|=" << n << " |E|=" << m << '\n';

}

void Graph::PrintMatrix() const {

cout << " ";

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

cout << Ch(i) << " ";

}

cout << "\n";

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

cout << Ch(i) << " ";

vector<int> row(n, 0);

for (int neighbor : adjList[i]) {

row[neighbor] = 1;

}

for (int value : row) {

cout << value << " ";

}

cout << '\n';

}

}

void Graph::ExecuteDBL() {

NUM.resize(n, 0);

L.resize(n, 0);

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

if (!NUM[i]) {

DBL(i, -1);

}

}

}

void Graph::DBL(int v, int p) {

using edge = pair<int, int>;

NUM[v] = L[v] = ++num;

for (int u : adjList[v]) {

if (!NUM[u]) {

edge currentEdge(u, v), poppedEdge(0, 0);

edgeStack.push(currentEdge);

DBL(u, v);

L[v] = min(L[v], L[u]);

if (L[u] >= NUM[v]) {

cout << "Edge <" << Ch(v) << '-' << Ch(u)

<< "> closes a component [";

do {

poppedEdge = edgeStack.top();

edgeStack.pop();

cout << Ch(poppedEdge.first) << '-' << Ch(poppedEdge.second) << "; ";

} while (poppedEdge != currentEdge && !edgeStack.empty());

cout << "]\n";

}

} else if (u != p && NUM[u] < NUM[v]) {

edgeStack.push(make\_pair(u, v));

L[v] = min(L[v], NUM[u]);

}

}

}

void Graph::CreateEdgeMatrix() {

edgeMatrix.assign(n, vector<int>(n, INT\_MAX));

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

edgeMatrix[i][i] = 0;

for (int j : adjList[i]) {

if (edgeMatrix[i][j] == INT\_MAX) {

edgeMatrix[i][j] = edgeMatrix[j][i] = rand() % 10 + 1;

}

}

}

}

void Graph::PrintEdgeMatrix() const {

cout << "Edge matrix:\n";

if (edgeMatrix.empty()) {

PrintMatrix();

return;

}

cout << " ";

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

cout << setw(3) << Ch(i) << " ";

}

cout << '\n';

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

cout << setw(3) << Ch(i) << " "; // Row label

for (int weight : edgeMatrix[i]) {

if (weight == INT\_MAX) {

cout << setw(3) << "INF";

} else {

cout << setw(3) << weight;

}

cout << " ";

}

cout << '\n';

}

}

void Graph::DijkstraShortestPath(int start, int end, vector<int>& path, int& length) const {

vector<int> dist(n, INT\_MAX);

vector<int> prev(n, -1);

vector<bool> visited(n, false);

dist[start] = 0;

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

int v = -1;

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

if (!visited[j] && (v == -1 || dist[j] < dist[v])) {

v = j;

}

}

if (dist[v] == INT\_MAX) break;

visited[v] = true;

for (int u : adjList[v]) {

int weight = edgeMatrix.empty() ? 1 : edgeMatrix[v][u];

if (dist[v] + weight < dist[u]) {

dist[u] = dist[v] + weight;

prev[u] = v;

}

}

}

length = dist[end];

for (int v = end; v != -1; v = prev[v]) {

path.push\_back(v);

}

reverse(path.begin(), path.end());

}

void Graph::FindShortestPath() {

char start, end;

cout << "Enter the start vertex: ";

cin >> start;

cout << "Enter the end vertex: ";

cin >> end;

if (start < 'a' || start >= 'a' + n || end < 'a' || end >= 'a' + n) {

cout << "Invalid vertices. Please use vertices from 'a' to '" << Ch(n - 1) << "'.\n";

return;

}

int startIdx = start - 'a';

int endIdx = end - 'a';

vector<int> path;

int length = 0;

DijkstraShortestPath(startIdx, endIdx, path, length);

cout << "\nShortest path from " << start << " to " << end << ":\n";

if (length == INT\_MAX) {

cout << "No path exists.\n";

} else {

cout << "Length: " << length << "\nPath: ";

for (int v : path) {

cout << Ch(v) << ' ';

}

cout << '\n';

}

}

int main() {

int answer;

cout << "===== Start =====\n";

Graph graph;

do {

cout << "\nMake a choice:\n";

cout << "1. Print adjacency list\n";

cout << "2. Print adjacency matrix\n";

cout << "3. Randomize edge matrix\n";

cout << "4. Print edge matrix\n";

cout << "5. Find shortest path between 2 points\n";

cout << "6. Find biconnected components\n";

cout << "0. Exit\n";

cin >> answer;

switch(answer){

case 1:

graph.PrintAdjList();

break;

case 2:

cout << "\nGraph adjacency matrix:\n";

graph.PrintMatrix();

break;

case 3:

graph.CreateEdgeMatrix();

break;

case 4:

graph.PrintEdgeMatrix();

break;

case 5:

graph.FindShortestPath();

break;

case 6:

graph.ExecuteDBL();

break;

default:

break;

}

} while (answer);

cout << "===== End =====\n";

cout << "Press any key to continue...";

cin.get();

cin.get();

}

**Список литературы**

1. Ахо Дж., Хопкрофт А., Ульман Дж. Структуры данных и алгоритмы. — СПб.: И. Д. Вильямс, 2001. — 382 c.
2. Бьерн Страуструп. Язык программирования С++.
3. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. В24 Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2011. — 1296 с. : ил. — Парал. тит. англ.