**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Информационных Систем**

**КУРСОВАЯ РАБОТА**

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

**Тема: «Графы»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3376 |  | Константинов Р.И. |
| Преподаватель |  | Молдовян Д.Н. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

**на курсовую работу**

Студент Хуснутдинов Р.Р.

Группа 3376

Тема работы: Реализация алгоритма поиска минимального остова на основе алгоритма Крускала.

Исходные данные:

Любой текстовый файл, содержащий матрицу смежности графа в виде:

A B C

0 3 1

3 0 2

1 2 0

Выходные данные:

Результат в виде отсортированных по имени пар и суммарный вес:

A C

B C

3

Максимальный размер входных данных: 50 вершин. Вершины могут быть заданы любой текстовой последовательностью без пробелов. Вес ребра ограничен интервалом от 1 до 1023 включительно.

|  |
| --- |
| **Содержание пояснительной записки:**  Введение, описание структуры рёбер, анализ времени выполнения и потребления памяти, класс для работы с непересекающимися множествами, общие алгоритмы обхода графа в глубину и ширину, реализация программы, полученные результаты, выводы, список литературы, приложение 1 — исходный код программы. |
|  |

Дата выдачи задания: 03.12.2024

Дата сдачи реферата: 20.12.2024

Дата защиты реферата: 20.12.2024

|  |  |  |
| --- | --- | --- |
| Студент |  | Константинов Р.И. |
| Преподаватель |  | Молдовян Д.Н. |

**АННОТАЦИЯ**

Данная работа посвящена изучению алгоритмов работы с графами. Основное внимание уделено алгоритму Краскала для нахождения минимального остовного дерева. Реализована программа на языке C++ с использованием системы непересекающихся множеств (DSU). В процессе работы рассматриваются структура данных для представления рёбер, сортировка, а также методы анализа временной и пространственной сложности алгоритмов. Программа находит минимальное остовное дерево, выводя его рёбра и общий вес.

**SUMMARY**

This project focuses on studying graph algorithms, particularly Kruskal’s algorithm for finding the minimum spanning tree. The program, implemented in C++, utilizes the disjoint-set union

(DSU) data structure. It explores data structures for edge representation, sorting, and analyzes time and space complexities of the algorithms. The program identifies the minimum spanning tree, outputting its edges and total weight.

**СОДЕРЖАНИЕ**

|  |  |  |
| --- | --- | --- |
|  | Введение | 6 |
| 1. | Теоретическая база | 7 |
| 1.1. | Алгоритм Краскала и его применение | 7 |
| 1.2. | Принцип работы алгоритма Краскала | 7 |
| 2. | Представление рёбер в графе | 9 |
| 2.1. | Реализация структуры данных для рёбер | 9 |
| 2.2. | Оценка времени выполнения и потребляемых ресурсов | 10 |
| 3. | Система непересекающихся множеств (DSU) | 12 |
| 3.1. | Реализация DSU | 13 |
| 3.2. | Оценка времени работы операций | 13 |
| 4. | Алгоритмы обхода в графах | 15 |
| 4.1. | Алгоритм обхода в глубину (DFS) | 15 |
| 4.2. | Алгоритм обхода в ширину (BFS) | 15 |
| 5. | Результат работы программы | 17 |
|  | Заключение | 18 |
|  | Список использованных источников | 19 |
|  | Приложение А: Программный код | 20 |

**ВВЕДЕНИЕ**

Цель работы: Реализовать алгоритм поиска минимального остова на основе алгоритма Краскала (Крускала). Продемонстрировать знания следующих вопросов:

* Сортировка
* Обход графов (в глубину и в ширину)
* Хранение графов (списки смежности, матрицы смежности, инцидентности)
* Построение системы непересекающихся множеств

Входные данные:

Любой текстовый файл, содержащий матрицу смежности графа в виде:

A B C

0 3 1

3 0 2

1 2 0

где первая строка содержит через пробел список всех рёбер, за которым следует матрица смежности. В матрице значение 0 стоит, если ребра между вершинами нет, и положительное число, которое соответствует весу, когда ребро между вершинами существует.

Выходные данные:

Результат в виде отсортированных по имени пар и суммарный вес:

A C

B C

3

Максимальный размер входных данных: 50 вершин. Вершины могут быть заданы любой текстовой последовательностью без пробелов. Вес ребра ограничен интервалом от 1 до 1023 включительно.

### 1. Теоретическая база

* 1. **Алгоритм Краскала и его применение**

**Алгоритм Краскала** — это один из классических алгоритмов для нахождения **минимального остовного дерева** (MST) в графе. Минимальное остовное дерево графа — это подмножество рёбер графа, которое соединяет все его вершины без циклов и с минимальной суммой весов рёбер.

Алгоритм Краскала работает на **неориентированных взвешенных графах**, где рёбра имеют веса, и его задача — выбрать такие рёбра, которые соединят все вершины графа, не образуя циклов, и при этом суммарный вес рёбер был минимальным.

#### **Принцип работы алгоритма Краскала**

Алгоритм Краскала находит минимальное остовное дерево (MST) в неориентированном взвешенном графе. Его задача — выбрать подмножество рёбер, которое соединяет все вершины графа, не образуя циклов, и минимизировать общую сумму весов рёбер. Основой работы является сортировка рёбер по весу и использование структуры данных «система непересекающихся множеств» (DSU), чтобы отслеживать соединённые компоненты графа и предотвращать циклы.

**Пошаговый процесс алгоритма:**

1. Все рёбра графа сортируются по весу.
2. Начинаем с самого лёгкого рёбра и проверяем, не образует ли оно цикл. Для этого используется DSU.
3. Если рёбра не образуют цикл, то оно добавляется в остовное дерево.
4. Процесс повторяется, пока не будут добавлены все рёбра, составляющие минимальное остовное дерево.

В коде процесс реализован следующим образом:

1. Рёбра собираются из матрицы смежности в список edges.
2. Эти рёбра сортируются с помощью функции quicksort.
3. Структура DSU используется для проверки, находятся ли две вершины в одном компоненте связности. Если нет — соединяем их.

std::vector<Edge> mst;

int totalWeight = 0;

std::vector<Edge> sortedEdges = edges;

std::sort(sortedEdges.begin(), sortedEdges.end());

DisjointSet ds(vertices.size());

for (const auto& edge : sortedEdges) {

int u = std::distance(vertices.begin(), std::find(vertices.begin(), vertices.end(), edge.vertex1));

int v = std::distance(vertices.begin(), std::find(vertices.begin(), vertices.end(), edge.vertex2));

if (ds.find(u) != ds.find(v)) {

ds.unite(u, v);

mst.push\_back(edge);

totalWeight += edge.weight;

}

}

### 2. Представление рёбер в графе

**2.1. Реализация структуры данных для рёбер**

В графах рёбра — это основные компоненты, которые связывают вершины. Для работы с графами необходимо правильно организовать данные, представляющие рёбра, чтобы эффективно решать задачи, такие как нахождение минимального остовного дерева (MST), выполнение поиска в глубину и в ширину, и другие. В данном разделе рассмотрим структуру данных для представления рёбер в графе, используемую в программе.

В коде для представления рёбер используется структура данных **Edge**. Эта структура инкапсулирует информацию о рёбрах графа, таких как вершины, которые она соединяет, и вес ребра. Структура **Edge** является основным элементом для алгоритма Краскала и других операций с графами.

**Структура данных Edge:**

struct Edge

{

std::string vertex1, vertex2;

int weight;

bool operator < (const Edge& edge) const

{

return weight < edge.weight;

}

};

**Поля структуры Edge:**

1. string vertex1 — первая вершина, которую соединяет ребро.
2. string vertex2 — вторая вершина, которую соединяет ребро.
3. int weight — вес рёбра, который указывает на "стоимость" соединения между вершинами.

**Оператор сравнения <:**

В структуре также определён оператор <, который позволяет сравнивать рёбра по весу. Этот оператор используется для сортировки рёбер при реализации алгоритма Краскала. Сортировка рёбер по весу позволяет выбрать минимальные рёбра для построения минимального остовного дерева.

**Перегрузка оператора сравнения:**

Чтобы алгоритм быстрой сортировки мог сравнивать рёбра по весу, в структуре Edge реализован оператор <:

bool operator < (const Edge& edge) const

{

return weight < edge.weight;

}

**2.2. Оценка времени выполнения и потребляемых ресурсов**

В данном разделе будет проведена оценка времени работы и использования памяти программы, основанной на алгоритме Краскала, а также других алгоритмов, таких как обходы графа (DFS и BFS) и сортировка рёбер. Оценка будет дана с учётом сложности используемых алгоритмов и структуры данных.

**Оценка времени выполнения:**

##### **Алгоритм Краскала**

Алгоритм Краскала работает путём перебора всех рёбер, начиная с самых лёгких, и объединяет компоненты связности с помощью структуры данных **система непересекающихся множеств** (СНМ).

**Сортировка стандартной библиотеки C++**

Стандартная библиотекаC++ предлагает алгоритм sort, который выполняют сортировку за время, пропорциональное n log(n), где N — количество элементов массива.

* **Время работы** алгоритма Краскала можно оценить как **O(m log m + m α(n))**, где:
  + **O(n log n)** — это время на сортировку рёбер.
  + **O(m α(m))** — это время на выполнение операций поиска и объединения в структуре данных СНМ. Здесь **α(m)** — это обратная функция от числа **n**, которая растёт очень медленно и практически является константой для большинства практических значений **n**.

Таким образом, для графа с **m** рёбрами и **n** вершинами общее время работы алгоритма Краскала будет **O(n log m)**, так как операция поиска и объединения имеет амортизированное время **O(α(n))**, которое можно считать константным.

#### **2.3.** **Общее время выполнения программы**

Программа выполняет несколько операций: сортировку рёбер, запуск алгоритма Краскала и выполнение обходов графа. Таким образом, общее время работы программы будет доминироваться временем сортировки рёбер, поскольку **O(n log n)** является наибольшей сложностью среди всех операций.

Итак, для графа с **n** вершинами и **m** рёбрами общее время работы программы будет **O(n log n + m + n)**.

**Оценка использования памяти**

1. **Рёбра**: каждый объект Edge содержит два строковых идентификатора и вес (int), что требует O(1) памяти. При n рёбрах общая память составит O(n).
2. **Матрица смежности**: занимает O(n²) памяти, где n — количество вершин.
3. **Система непересекающихся множеств (DSU)**: два вектора (parent и rank) требуют O(n) памяти.

Итого: суммарное использование памяти программы для плотных графов (где количество рёбер близко к n²) будет O(n²), для разреженных графов — O(n + m), где m — количество рёбер.

### 3. Система непересекающихся множеств (DSU)

**3.1. Реализация DSU**

Система непересекающихся множеств, или **Disjoint Set Union (DSU)**, представляет собой класс, который поддерживает операции объединения и поиска для множества элементов. Используется в таких задачах, как нахождение минимального остовного дерева или обработка динамических связей между элементами.

#### **Основные операции системы непересекающихся множеств:**

#### Система DSU поддерживает два основных типа операций:

1. **Поиск (find)**: Определяет, к какому множеству принадлежит элемент. Если два элемента принадлежат одному и тому же множеству, то результат их поиска будет одинаковым. Эта операция используется, чтобы проверить, не образуют ли два рёбра цикл в процессе построения остовного дерева.
2. **Объединение (**unite**)**: Объединяет два множества в одно. Это операция объединяет два элемента, находящихся в разных множествах, в одно множество. В контексте алгоритма Краскала это позволяет добавить новое ребро в остовное дерево без образования цикла.

#### **3.1.1. Реализация СНМ (DSU):**

В коде для реализации системы непересекающихся множеств используется два основных вектора: **parent** и **rank**. Операции поиска и объединения выполняются с использованием этих векторов.

class DisjointSet {

public:

DisjointSet(int size) : parent(size), rank(size, 0)

{

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

parent[i] = i;

}

}

int find(int u)

{

if (u != parent[u])

parent[u] = find(parent[u]);

return parent[u];

}

void unite(int u, int v)

{

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV])

parent[rootV] = rootU;

else if (rank[rootU] < rank[rootV])

parent[rootU] = rootV;

else

{

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

private:

std::vector<int> parent;

std::vector<int> rank;

};

* vector<int> parent: Содержит информацию о родительском элементе множества, к которому принадлежит элемент **u**. Если **parent[u] == u**, то **u** является корнем множества.
* vector<int> rank: Используется для оптимизации операции объединения. Он хранит "высоту" (или глубину) дерева, которое представляет множество. Чем больше **rank**, тем глубже дерево.

##### **3.1.2. Операция find** Операция **find** используется для поиска представителя множества (или корня дерева), к которому принадлежит элемент.

int find(int u) {

if (u != parent[u])

parent[u] = find(parent[u]);

return parent[u];

}

При выполнении операции **find** мы идём по дереву до его корня, а затем сжимаем путь, что помогает ускорить выполнение последующих операций.

##### **3.1.3. Операция unite**

Операция unite объединяет два множества. Чтобы избежать образования слишком глубоких деревьев, используется объединение по рангу. Это означает, что более мелкое дерево (с меньшим рангом) присоединяется к корню более глубокого дерева. Таким образом, деревья остаются сбалансированными, что повышает эффективность операций.

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

}

else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

}

else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

**3.2. Оценка времени работы операций**

Обе операции — find и unite — могут быть выполнены за амортизированное время O(n), являясь константным.

* Операция find: Благодаря пути сжатия, она работает очень эффективно, даже для больших наборов данных.
* Операция unite: Использование объединения по рангу также улучшает производительность, сводя к минимуму возможную глубину деревьев.

Таким образом, система непересекающихся множеств является высокоэффективной структурой данных для решения задач, связанных с объединением компонент и проверкой их связности.

### 4. Основные алгоритмы обхода в графах

Обходы — это методы, которые посещают все вершины графа, следуя определенному порядку, в отличие от поиска, который ориентирован на нахождение конкретных элементов.

1. **Обход в глубину (DFS)**
2. **Обход в ширину (BFS)**

**4.1. Алгоритм обхода в глубину (DFS)**

Обход в глубину (DFS, Depth-First Search) — это алгоритм, который посещает вершины графа, начиная с начальной, и дальше углубляется по пути, посещая соседей, пока не дойдёт до конечных вершин, не имеющих непосещённых соседей. После этого он возвращается и продолжает обход других вершин.

Алгоритм продолжает обходить граф по пути, пока не дойдёт до конечных вершин. Если одна ветвь заканчивается, алгоритм возвращается и пытается исследовать другие ветви.

* 1. **Алгоритм обхода в ширину (BFS)**

Обход в ширину (BFS, Breadth-First Search) — это алгоритм, который исследует все вершины на одном уровне (соседей текущей вершины) перед тем, как перейти к следующему уровню.

В отличие от DFS, который углубляется в граф, BFS исследует все соседние вершины сначала, а затем переходит к следующему уровню.

### 5. Результат работы программы

Продемонстрировано чтение матрицы смежности (рис. 1). Так же показан пример работы алгоритма Краскала (рис. 2).

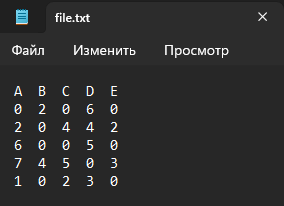


Рисунок 1 – Пример таблицы смежности в файле

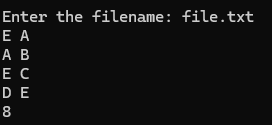


Рисунок 2 – Вывод рёбер и веса остовного дерева

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения работы была реализованна программа, которая считывает из файла матрицу смежности графа и при помощи алгоритма Краскала приводит граф к виду дерева. Выводятся ребра остовного графа и суммарный вес

В работе была продемонстрирована реализация ряда следующих алгоритмов и структур данных, таких как:

1. Алгоритм Краскала для поиска минимального остова. Была использована сиситема непересекающихся множеств (DSU), чтобы эффективно отслеживать компоненты связности и избегать образования циклов при добавлении рёбер в остовное дерево.

2. Система непересекающихся множеств (DSU), а так же функции, с помощью которых она была реализована: операция поиска (find) и объединения (unite).

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Материалы для изучения // Сайт Максима Пелевина. URL: <https://markoutte.me/students/> .

3. Теоретическая база // URL:

<https://habr.com/ru/articles/569444/>

4. Обходы графа // URL:

<https://www.youtube.com/watch?v=VehB3eglQMQ>

Дата обращения ко всем источникам: 14.12.2024.

**приложение А**

**ПРОГРАММНЫЙ КОД**

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <algorithm>

#include <sstream>

struct Edge

{

std::string vertex1, vertex2;

int weight;

bool operator < (const Edge& edge) const

{

return weight < edge.weight;

}

};

class DisjointSet {

public:

DisjointSet(int size) : parent(size), rank(size)

{

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

parent[i] = i;

}

}

int find(int u)

{

if (u != parent[u])

parent[u] = find(parent[u]);

return parent[u];

}

void unite(int u, int v)

{

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV])

parent[rootV] = rootU;

else if (rank[rootU] < rank[rootV])

parent[rootU] = rootV;

else

{

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

private:

std::vector<int> parent;

std::vector<int> rank;

};

void readGraph(const std::string& filename, std::vector<Edge>& edges, std::vector<std::string>& vertices) {

std::ifstream file(filename);

if (!file.is\_open()) {

throw std::runtime\_error("Unable to open file");

}

std::string line;

// Read vertices

if (std::getline(file, line)) {

std::istringstream iss(line);

std::string vertex;

while (iss >> vertex) {

vertices.push\_back(vertex);

}

}

int n = vertices.size();

for (int i = 0; i < n && std::getline(file, line); ++i) {

std::istringstream iss(line);

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

int weight;

iss >> weight;

if (weight > 0) {

edges.push\_back({ vertices[i], vertices[j], weight });

}

}

}

file.close();

}

void kruskalMST(const std::vector<Edge>& edges, const std::vector<std::string>& vertices) {

std::vector<Edge> mst;

int totalWeight = 0;

std::vector<Edge> sortedEdges = edges;

std::sort(sortedEdges.begin(), sortedEdges.end());

DisjointSet ds(vertices.size());

for (const auto& edge : sortedEdges) {

int u = std::distance(vertices.begin(), std::find(vertices.begin(), vertices.end(), edge.vertex1));

int v = std::distance(vertices.begin(), std::find(vertices.begin(), vertices.end(), edge.vertex2));

if (ds.find(u) != ds.find(v)) {

ds.unite(u, v);

mst.push\_back(edge);

totalWeight += edge.weight;

}

}

for (const auto& edge : mst) {

std::cout << edge.vertex1 << " " << edge.vertex2 << std::endl;

}

std::cout << totalWeight << std::endl;

}

int main() {

std::string filename;

std::cout << "Enter the filename: ";

std::cin >> filename;

try {

std::vector<Edge> edges;

std::vector<std::string> vertices;

readGraph(filename, edges, vertices);

kruskalMST(edges, vertices);

}

catch (const std::exception& e) {

std::cerr << "Error: " << e.what() << std::endl;

return 1;

}

return 0;

}