ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

*«*САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

**Курсовой проект**

по дисциплине «Объектно-ориентированное программирование»

**«Алгоритм Краскала, алгоритм Прима»**

|  |  |  |
| --- | --- | --- |
| Выполнил  студент гр. 3331506/20401 | *(подпись)* | Грейсер С. Е. |
| Работу принял | *(подпись)* | Ананьевский М.С. |

Санкт-Петербург

2025

**Введение**

Остов графа — это подграф, содержащий все вершины исходного графа и являющийся деревом, то есть не содержащим циклов. Иными словами, остов соединяет все вершины графа минимальным числом рёбер без образования замкнутых путей.

Минимальное остовное дерево (Minimum Spanning Tree, MST) — это остовное дерево взвешенного неориентированного графа, имеющее минимально возможную сумму весов рёбер. Поиск минимального остовного дерева является одной из фундаментальных задач теории графов и широко применяется в различных прикладных областях.

**Предмет исследования**: алгоритмы поиска минимального остовного дерева.

**Задача**: изучить два классических алгоритма построения минимального остовного дерева: Краскала и Прима, провести их реализацию на языке программирования C++, исследовать их временную эффективность на различных объемах данных, провести сравнительный анализ полученных результатов, сделать выводы о целесообразности применения каждого из алгоритмов в различных ситуациях.

**Актуальность темы**. Алгоритмы построения минимального остовного дерева находят применение в различных отраслях, среди которых можно выделить:

* Проектирование линий электропередач и коммуникационных сетей. При необходимости соединить несколько объектов с минимальными затратами на строительство (например, прокладку кабеля) применяется построение MST.
* Оптимизация логистических маршрутов. MST позволяет минимизировать суммарную стоимость дорог или маршрутов между складами и пунктами доставки.
* Кластеризация данных в машинном обучении.

**1 Алгоритм Краскала**

* 1. **Описание алгоритма**

Алгоритм Краскала — это жадный алгоритм, предназначенный для построения минимального остовного дерева связного взвешенного неориентированного графа. Он был предложен Джозефом Краскалом в 1956 году [1].

Основная идея алгоритма заключается в пошаговом добавлении рёбер с наименьшим весом при условии, что их включение не создаёт цикла. Для контроля образования циклов используется структура данных "система непересекающихся множеств" (Disjoint Set Union, DSU).

Общий принцип работы:

1. Отсортировать все рёбра графа по неубыванию веса.
2. Инициализировать каждую вершину как отдельное дерево (компоненту связности).
3. Повторять следующее, пока не будет добавлено V−1 рёбер (где V — количество вершин):
   * Взять минимальное по весу ребро, которое соединяет вершины из разных деревьев.
   * Добавить это ребро в остов.
   * Объединить множества, к которым принадлежат соединённые вершины.
   1. **Оценка сложности алгоритма**

Пусть:

* V — количество вершин в графе,
* E — количество рёбер.

Временная сложность

Алгоритм Краскала состоит из следующих шагов:

1. Сортировка рёбер по весу — основной этап, имеющий сложность:

O(ElogE)

1. Инициализация DSU — создаются массивы parent и rank, каждый размером V. Эта операция требует:

O(V)

1. Обработка каждого ребра — каждое ребро обрабатывается один раз. Для проверки принадлежности вершин одному компоненту выполняются операции find, для объединения — union.

С использованием компрессии пути и балансировки по рангу операции find и union имеют практически константную сложность:

O(α(V))

где α(V) — обратная функция Аккермана, чрезвычайно медленно растущая; для всех практических значений V её можно считать константой:

α(V) ≤ 5.

Таким образом, суммарная стоимость всех DSU-операций:

O(E⋅α(V)) ≈ O(E)

Итоговая временная сложность алгоритма Краскала:

O(ElogE)

Алгоритм требует память для:

* хранения списка рёбер: O(E);
* хранения массива parent и rank для DSU: O(V);

Таким образом, общая сложность по памяти составляет:

O(V+E)

* 1. **Экспериментальное исследование**

Для экспериментального исследования были сгенерированы 10 графов, содержащие 10 000, 100 000, 300 000, 700 000, 1 000 000 вершин. 5 графов содержало в 3 раза больше ребер по сравнению с вершинами (разреженные графы), 5 графов содержало в себе в 10 раз больше ребер по сравнению с вершинами (плотные графы). Результаты исследования представлены на рисунках 1, 2.

Рисунок 1 – Время выполнения алгоритма Краскала для разреженных графов

Рисунок 2 – Время выполнения алгоритма Краскала для плотных графов

**2. Алгоритм Прима**

**2.1 Описание алгоритма**

Алгоритм Прима — это жадный алгоритм, предназначенный для построения минимального остовного дерева связного взвешенного неориентированного графа. Алгоритм был предложен Робертом Примом в 1957 году [2].

Основная идея алгоритма заключается в пошаговом расширении остовного дерева, начиная с произвольной вершины, путём добавления рёбер с минимальным весом, которые соединяют уже включённые вершины с оставшимися. Алгоритм использует структуру данных кучу (или приоритетную очередь) для эффективного выбора ребра с минимальным весом.

Общий принцип работы:

1. Выбрать произвольную вершину и включить её в остов.
2. Поместить все рёбра, соединенные с этой вершиной, в очередь с приоритетом.
3. Повторять следующее, пока все вершины не будут включены в остов:
   * Извлечь ребро с минимальным весом из очереди.
   * Если вершина, соединённая этим ребром, ещё не добавлена в остов, добавить её.
   * Поместить все рёбра, соединенные с новой вершиной, в очередь с приоритетом.

**2.2 Оценка сложности алгоритма**

Пусть:

* V — количество вершин в графе,
* E — количество рёбер.

Алгоритм Прима состоит из следующих шагов:

1. Инициализация:
   * Операции инициализации (создание кучи, массивов) выполняются за O(V).
2. Обработка рёбер:
   * Для каждой вершины мы вставляем её рёбра в кучу. Операция вставки в кучу занимает O(logV), и таких операций будет не более E. Таким образом, сложность обработки всех рёбер составляет O(ElogV).
3. Извлечение минимальных рёбер:
   * Для каждой вершины извлекаем минимальное ребро. Эта операция также занимает O(logV) и выполняется V раз.

Итак, суммарная временная сложность алгоритма Прима:

O((V + E) logV)

Для плотных графов, где E ≈ V², сложность будет O(E logV).

Сложность по памяти:

* Для хранения списка рёбер: требуется O(E) памяти.
* Для хранения очереди с приоритетом (кучи): требуется O(V) памяти для хранения рёбер, инцидентных вершинам.
* Для хранения других вспомогательных структур (массивы для хранения меток вершин, минимальных рёбер и т.д.): потребуется O(V) памяти.

Таким образом, общая сложность по памяти:

O(V+E)

**2.3 Экспериментальное исследование**

Для экспериментального исследования были использованы файлы с теми же данными, что и для алгоритма Краскала. Результаты исследования представлены на рисунках 3, 4.

Рисунок 3 – Время выполнения алгоритма Прима для разреженных графов

Рисунок 4 – Время выполнения алгоритма Прима для плотных графов

**3 Выводы**

В курсовой работе были рассмотрены и реализованы два классических алгоритма построения минимального остовного дерева: алгоритм Краскала и алгоритм Прима. Оба алгоритма решают одну и ту же задачу, однако используют различные подходы: Краскал основывается на сортировке рёбер и объединении компонент связности, а Прим — на поэтапном наращивании остовного дерева с использованием очереди с приоритетом.

В теоретическом плане алгоритм Прима обладает временной сложностью O((V + E)logV), что делает его предпочтительным для плотных графов, тогда как алгоритм Краскала имеет сложность O(ElogE), что может быть более выгодно при работе с разреженными графами. Однако в ходе экспериментального исследования было выявлено, что алгоритм Краскала демонстрирует лучшее время выполнения даже на плотных графах.

**Список литературы**

1. Kruskal J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem // Proceedings of the American Mathematical Society. — 1956. — Vol. 7, No. 1. — P. 48–50. — DOI: 10.2307/2033241.

2. Prim R. C. Shortest connection networks and some generalizations // Bell System Technical Journal. — 1957. — Vol. 36. — P. 1389–1401. — DOI: 10.1002/j.1538-7305.1957.tb01515.x.

**Приложение 1. Алгоритм Краскала**

#include <iostream>

#include <vector>

#include <algorithm>

#include <fstream>

using namespace std;

struct Edge {

    int from;

    int to;

    int weight;

    bool operator<(const Edge& other) const {

        return weight < other.weight;

    }

};

class DisjointSetUnion {

private:

    vector<int> parent;

    vector<int> rank;

public:

    explicit DisjointSetUnion(int n) : parent(n), rank(n, 0) {

        for (int i = 0; i < n; ++i)

            parent[i] = i;

    }

    int find(int v) {

        if (v != parent[v])

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

        return parent[v];

    }

    void unite(int u, int v) {

        u = find(u);

        v = find(v);

        if (u == v) return;

        if (rank[u] < rank[v])

            parent[u] = v;

        else {

            parent[v] = u;

            if (rank[u] == rank[v])

                ++rank[u];

        }

    }

};

vector<Edge> load\_edges\_from\_file(const string& filename, int& vertex\_count) {

    ifstream in(filename);

    int edge\_count;

    in >> vertex\_count >> edge\_count;

    vector<Edge> edges(edge\_count);

    for (int i = 0; i < edge\_count; ++i) {

        in >> edges[i].from >> edges[i].to >> edges[i].weight;

    }

    return edges;

}

pair<int, vector<Edge>> kruskal\_mst(int vertex\_count, const vector<Edge>& edges) {

    DisjointSetUnion dsu(vertex\_count);

    vector<Edge> mst;

    int total\_weight = 0;

    vector<Edge> sorted\_edges = edges;

    sort(sorted\_edges.begin(), sorted\_edges.end());

    for (const Edge& edge : sorted\_edges) {

        if (dsu.find(edge.from) != dsu.find(edge.to)) {

            dsu.unite(edge.from, edge.to);

            mst.push\_back(edge);

            total\_weight += edge.weight;

            if (mst.size() == vertex\_count - 1) break;

        }

    }

    return {total\_weight, mst};

}

int main() {

    string filename = "C://C\_programming//coursework//graph1000\_thin.txt";

    int vertex\_count;

    vector<Edge> edges = load\_edges\_from\_file(filename, vertex\_count);

    auto mst = kruskal\_mst(vertex\_count, edges);

    cout << "Минимальное остовное дерево построено.\n";

    cout << "Количество рёбер в MST: " << mst.second.size() << "\n";

    cout << "Суммарный вес MST: " << mst.first << "\n";

    return 0;

}

**Приложение 2. Алгоритм Прима**

#include <iostream>

#include <vector>

#include <queue>

#include <fstream>

#include <stdexcept>

#include <limits>

using namespace std;

struct Edge {

    int from;

    int to;

    int weight;

    bool operator<(const Edge& other) const {

        return weight > other.weight;

    }

};

vector<Edge> load\_edges\_from\_file(const string& filename, int& vertex\_count) {

    ifstream in(filename);

    if (!in.is\_open()) throw runtime\_error("Невозможно открыть файл");

    int edge\_count;

    in >> vertex\_count >> edge\_count;

    vector<Edge> edges(edge\_count);

    for (int i = 0; i < edge\_count; ++i) {

        in >> edges[i].from >> edges[i].to >> edges[i].weight;

    }

    return edges;

}

pair<int, vector<Edge>> prim\_mst(int vertex\_count, const vector<Edge>& edges) {

    vector<vector<Edge>> graph(vertex\_count);

    for (const Edge& edge : edges) {

        graph[edge.from].push\_back({edge.from, edge.to, edge.weight});

        graph[edge.to].push\_back({edge.to, edge.from, edge.weight});

    }

    vector<bool> visited(vertex\_count, false);

    priority\_queue<Edge> pq;

    vector<Edge> mst;

    int total\_weight = 0;

    pq.push({-1, 0, 0});

    while (!pq.empty()) {

        Edge edge = pq.top();

        pq.pop();

        int u = edge.to;

        if (visited[u]) continue;

        visited[u] = true;

        total\_weight += edge.weight;

        if (edge.from != -1) {

            mst.push\_back(edge);

        }

        for (const auto& neighbor : graph[u]) {

            if (!visited[neighbor.to]) {

                pq.push(neighbor);

            }

        }

    }

    return {total\_weight, mst};

}

int main() {

    try {

        const string filename = "C://C\_programming//coursework//graph1000\_thin.txt";

        int vertex\_count;

        vector<Edge> edges = load\_edges\_from\_file(filename, vertex\_count);

        auto mst = prim\_mst(vertex\_count, edges);

        cout << "Минимальное остовное дерево построено.\n";

        cout << "Количество рёбер в MST: " << mst.second.size() << "\n";

        cout << "Суммарный вес MST: " << mst.first << "\n";

    } catch (const exception& ex) {

        cerr << "Ошибка: " << ex.what() << "\n";

    }

    return 0;

}