**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра ИС**

**КУРСОВАЯ РАБОТА**

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

**Тема: Графы**

| Студент гр. 3375 |  | Порофиев Е.А. |
| --- | --- | --- |
| Преподаватель |  | Пелевин М.С. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

**НА КУРСОВУЮ РАБОТУ**

| Студент Порофиев Е.А. | | |
| --- | --- | --- |
| Группа 3375 | | |
| Тема работы: графы | | |
| Исходные данные:  Продемонстрировать знания следующих вопросов:  1) сортировка  2) обход графов (в глубину и в ширину)  3) хранение графов (списки смежности, матрицы смежности, инцидентности)  4) построение системы непересекающихся множеств  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.  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:  Перечисляются требуемые разделы пояснительной записки (обязательны разделы «Содержание», «Введение», «Заключение», «Список использованных источников») | | |
|  | | |
|  | | |
|  | | |
|  | | |
| Студент гр. 3375 |  | Порофиев Е.А. |
| Преподаватель |  | Пелевин М.С. |

**АННОТАЦИЯ**

В своем курсовом проекте, согласно заданию, я реализовал алгоритм Краскала(Kruskal) для поиска минимального остова взвешенного графа. Для этого я использовал хранение графа в виде матрицы смежности, построение системы непересекающихся множеств, а также сортировку Timsort, для сортировки ребер графа по их весу и по имени вершин. Также, были реализованы алгоритмы обхода графа в глубину и в ширину. Ввод графа происходит из текстового файла, где записана матрица смежности этого графа.

**СОДЕРЖАНИЕ**

|  | Введение | 5 |
| --- | --- | --- |
| 1. | Структуры и классы | 5-8 |
| 1.1. | Структура Edge | 5 |
| 1.2. | Структура Graph | 5-6 |
| 1.3. | Структура DisjointSets | 6-7 |
| 2. | Алгоритм Краскала | 8-9 |
| 3. | Обходы графа | 9-10 |
| 3.1. | Обход графа в глубину | 9 |
| 3.2. | Обход графа в ширину | 10 |
| 4. | Вспомогательные алгоритмы | 10-12 |
| 4.1 | Сортировка ребер Timsort | 10-11 |
| 4.2 | Чтение матрицы смежности из файла | 11 |
|  | Приложение А. Пример запуска программы | 11-12 |
|  | Заключение | 12 |
|  | Список использованных источников | 12 |
|  |  |  |

# ВВЕДЕНИЕ

Целью работы было ознакомиться с графами как со структурой данных (способами их хранения, обхода, а также с алгоритмом Краскала(Kruskal)). Для этого нам было предложено использовать язык программирования C++ и среду разработки Microsoft Visual Studio.

# 1. СТРУКТУРЫ И КЛАССЫ

* **Структура Edge.**

Для представления ребра графа была использована следующая структура:

struct Edge

{

int u, v, weight;

Edge(int \_u, int \_v, int \_weight) : u(\_u), v(\_v), weight(\_weight) {}

Edge() : u(0), v(0), weight(0) {} };weight(weight)

};

* **Класс union\_find\_data**

Для представления системы непересекающихся множеств был использован следующий класс:

class union\_find\_data

{

public:

vector parent, rank;

int n;

union\_find\_data(int \_n)

{

parent = rank = vector(\_n);

n = \_n;

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

{

parent[i] = i;

rank[i] = 0;

}

}

int findD(int x)

{

if (parent[x] != x)

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

return parent[x];

}

void unionD(int x, int y)

{

int xset = find(x);

int yset = find(y);

if (xset == yset)

return;

if (rank[xset] < rank[yset])

parent[xset] = yset;

else if(rank[xset] > rank[yset])

parent[yset] = xset;

else

{

parent[yset] = xset;

rank[xset] = rank[xset] + 1;

}

}

};

* **Класс Stack**

Для представления стека был использован следующий класс:

class Stack

{

public:

Stack() : head(nullptr) {}

void push(const S x)

{

Node\* new\_node = new Node; new\_node->data = x;

new\_node->prev = head; head = new\_node; }

bool empty() { if (head == nullptr) { return true; } else { return false; } }

S top()

{

if (empty())

{

throw length\_error("stack is empty");

}

return head->data; }

void pop()

{

if (empty())

{

throw length\_error("stack is empty");

}

Node \*temp\_prev = head;

head = temp\_prev->prev;

delete temp\_prev; }

~Stack()

{

while (!empty()) { pop(); }

}

private:

struct Node

{

S data;

Node \*prev;

};

Node\* head;

};

# 2. АЛГОРИТМ КРАСКАЛА

Алгоритм Крускала используется для нахождения минимального остовного дерева (MST) взвешенного графа. Он является одним из классических методов решения задачи поиска MST и применяется при работе с графами, где необходимо минимизировать общую стоимость ребер, входящих в дерево.

*Входные**данные****:***

* adjMatrix – матрица смежности, представляющая весовые значения рёбер между вершинами графа.
* n – количество вершин в графе.

*Выходные данные:*

* mst – вектор рёбер, составляющих минимальное остовное дерево.
* totalWeight – суммарный вес всех рёбер в MST.

**Основные шаги работы алгоритма:**

*Создание списка рёбер:*

1. В начале мы собираем все рёбра графа в список edges. Для этого проходимся по верхней треугольной части матрицы смежности (так как она симметрична), чтобы избежать дублирования рёбер. Каждое ненулевое значение в матрице интерпретируется как вес ребра между соответствующими вершинами.

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

{

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

{

if (adjMatrix[i][j] != 0)

{

edges.push\_back(Edge(i, j, adjMatrix[i][j]));

}

}

}

*Сортировка рёбер по возрастанию веса:*

После того как список рёбер собран, сортируем их по возрастанию веса. Это делается методом вставки (insertion sort).

for (int i = 1; i < edges.size(); i++)

{

Edge key = edges[i];

int j = i - 1;

while (j >= 0 && edges[j].weight > key.weight)

{

edges[j + 1] = edges[j];

j--;

}

edges[j + 1] = key;

}

*Использование структуры данных «Система непересекающихся множеств» (Union-Find):*

Чтобы проверить, образуют ли два конца ребра цикл в текущем дереве, используем структуру данных Union-Find. Она позволяет эффективно проверять принадлежность двух вершин одному компоненту связности и объединять компоненты, если они разные.

union\_find\_data ds(n); // Создаем систему непересекающихся множеств для n вершин

**Построение минимального остовного дерева:**

Проходим по отсортированному списку рёбер от самого лёгкого до самого тяжёлого. Если добавление текущего ребра не создаёт цикла (то есть концы ребра принадлежат разным компонентам связности), добавляем его в итоговый набор рёбер MST и обновляем систему непересекающихся множеств.

for (int i = 0; i < edges.size(); i++)

{

const Edge& e = edges[i];

int u = e.u;

int v = e.v;

if (ds.findD(u) != ds.findD(v)) // Проверяем, находятся ли вершины в разных компонентах

{

mst.push\_back(e); // Добавляем ребро в MST

ds.unionD(u, v); // Объединяем компоненты

totalWeight += e.weight; // Увеличиваем общий вес MST

}

}

**Вывод результата:**

По завершении построения MST выводим суммарный вес рёбер и возвращаем сам набор рёбер.

cout << "Weight: " << totalWeight << endl;

return mst;

Таким образом, алгоритм Крускала последовательно добавляет самые легкие рёбра в дерево, избегая создания циклов, пока не будет построено минимальное остовное дерево.

Эти функции обходят граф, начиная с заданной стартовой вершины, используя соответственно очередь (queue) для BFS и стек (Stack) для DFS.

# 3. ОБХОДЫ ГРАФА

Функции bfs и dfs, приведённые после основной функции kruskal, используются для обхода графа в ширину (BFS) и глубину (DFS). Они не связаны непосредственно с работой алгоритма Крускала, но могут использоваться для анализа свойств полученного графа.

**3.1. Обход в графа в глубину**

Шаги функции:

1. Инициализация необходимых переменных: размер графа, массив посещённости и стек для хранения вершин:
   * n — размер матрицы смежности, то есть количество вершин в графе.
   * visited — массив булевых значений, который отслеживает посещённость каждой вершины. Изначально все вершины считаются непосещёнными (false).
   * s — стек, который используется для хранения вершин, ожидающих обработки.
2. Начало обхода. Начальная вершина помещается в стек для начала обхода:
   * Стартовая вершина помещается в стек s.

Основной цикл:

1. Пока стек не пуст, выполняется следующий процесс:
   * Извлекаем вершину из стека и проверяем её посещенность. Если она ещё не была посещена, помечаем её как посещённую и выводим информацию о ней.
   * Затем для текущей вершины просматриваются все её соседи (сначала последние элементы строки матрицы смежности):
     + Если соседняя вершина ещё не посещалась и существует ребро между ними (значение в матрице смежности больше нуля), эта вершина добавляется в стек для последующей обработки.

void dfs(const vector<vector<int>>& adjMatrix, int start, const vector<string>& vertices)

**3.2. Обход графа в ширину**

Шаги выполнения функции:

1. Инициализация переменных:
   * n — размер матрицы смежности, то есть количество вершин в графе.
   * visited — массив булевых значений, который отслеживает посещённость каждой вершины. Изначально все вершины считаются непосещёнными (false).
   * q — очередь, которая используется для хранения вершин, ожидающих обработки.
2. Начало обхода:
   * Стартовая вершина помещается в очередь q и помечается как посещенная.

Основной цикл:

1. Пока очередь не пуста, выполняется следующий процесс:
   * Извлекаем вершину из очереди и выводим информацию о ней.
   * Для текущей вершины просматриваются все её соседи:
     + Если соседняя вершина ещё не посещалась и существует ребро между ними (значение в матрице смежности больше нуля), эта вершина добавляется в очередь для последующей обработки и помечается как посещённая.

void bfs(const vector<vector<int>>& adjMatrix, int start, const vector<string>& vertices)

**4. ВСПОМОГАТЕЛЬНЫЕ АЛГОРИТМЫ**

* **Сортировка ребер алгоритмом Timsort**

Алгоритм Timsort — это гибридная сортировка, которая сочетает в себе два алгоритма: вставочную сортировку (Insertion Sort) для малых массивов и сортировку слиянием (Merge Sort) для больших массивов. Сначала происходит предварительная сортировка каждого подмассива, а затем эти подмассивы сливаются.

Функция insertSort() выполняет вставочную сортировку для небольших подмассивов. Она применяется для "разделения" входного массива на более мелкие части перед тем, как будет применена сортировка слиянием.

После применения вставочной сортировки к подмассивам, наступает этап сортировки слиянием для объединения отсортированных подмассивов. Алгоритм слияния работает по принципу "слияния" двух отсортированных массивов в один отсортированный массив.

Функция merge() отвечает за слияние двух отсортированных подмассивов.

Алгоритм Timsort имеет следующие характеристики:

Лучший случай: n

Средний: 𝑂(𝑛log𝑛)

Худший: 𝑂(𝑛log𝑛⁡

Алгоритм Timsort комбинирует преимущества вставочной сортировки (для небольших массивов) и сортировки слиянием (для более крупных массивов), что делает его весьма эффективным для реальных данных, которые часто имеют частичную сортировку.

* **Чтение матрицы смежности из файла**

**ПРИМЕР ЗАПУСКА ПРОГРАММЫ**

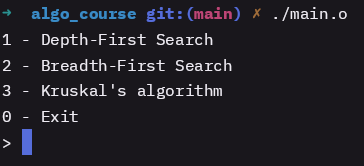


Рис.1. Запуск программы

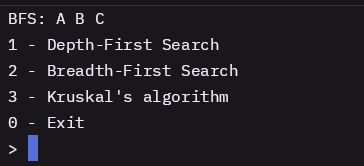


Рис.2. Обход в ширину

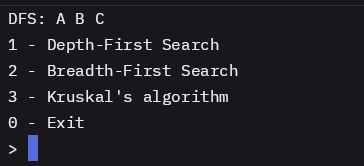


Рис.3. Обход в глубину

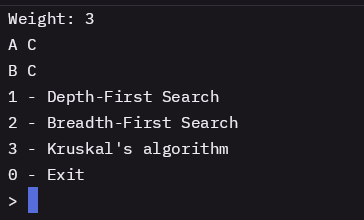
****

Рис.4. Работа алгоритма Крускала. Вес MST и ребра входящие в него

**ЗАКЛЮЧЕНИЕ**

По итогам выполнения курсовой работы удалось выполнить все поставленные задачи. Успешно реализованы функции для СНМ, хранение и обходы графов, а также алгоритм Краскала.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

* [Материалы для студентов | Сайт Максима Пелевина](https://markoutte.me/students/?ysclid=m4jshan43h506603104)(дата обращения 15.12.24)
* <https://neerc.ifmo.ru/wiki/index.php?title=%D0%9E%D0%B1%D1%85%D0%BE%D0%B4_%D0%B2_%D0%B3%D0%BB%D1%83%D0%B1%D0%B8%D0%BD%D1%83,_%D1%86%D0%B2%D0%B5%D1%82%D0%B0_%D0%B2%D0%B5%D1%80%D1%88%D0%B8%D0%BD>(дата обращения 15.12.24)
* <https://neerc.ifmo.ru/wiki/index.php?title=%D0%9E%D0%B1%D1%85%D0%BE%D0%B4_%D0%B2_%D1%88%D0%B8%D1%80%D0%B8%D0%BD%D1%83>(дата обращения 15.12.24)
* <https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D1%81%D0%BC%D0%B5%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B8>(дата обращения 15.12.24)
* <https://neerc.ifmo.ru/wiki/index.php?title=%D0%9C%D0%B0%D1%82%D1%80%D0%B8%D1%86%D0%B0_%D1%81%D0%BC%D0%B5%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D0%B3%D1%80%D0%B0%D1%84%D0%B0>(дата обращения 15.12.24)
* <https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BD%D0%B5%D0%BF%D0%B5%D1%80%D0%B5%D1%81%D0%B5%D0%BA%D0%B0%D1%8E%D1%89%D0%B8%D1%85%D1%81%D1%8F_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2>(дата обращения 15.12.24)
* <https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9A%D1%80%D0%B0%D1%81%D0%BA%D0%B0%D0%BB%D0%B0> (дата обращения 15.12.24)