Лабораторная работа № 9 по курсу дискретного анализа: Графы

Выполнил студент группы М8О-308Б-20 *Ядров Артем*.

Условие

Реализовать программу на языке C или C++, соответствующую построенному алгоритму. Формат входных и выходных данных описан в варианте задания.

Задан неориентированный граф, состоящий из n вершин и m ребер. Вершины пронумерованы целыми числами от 1 до n. Необходимо вывести все компоненты связности данного графа.

Метод решения

Для данной задачи необходимо полностью обойти граф. Сделать это можно 2 способами: обходом в глубину или обходом в ширину. Я выбрал обход в ширину.

Суть обхода в ширину заключается в последовательном обходе отдельных уровней графа, начиная с заданной вершины.

Рассматриваются все рёбра (u, v), выходящие из вершины u. Вершина v добавляется в очередь. После того, как будут проверены все рёбра, выходящие из вершины u, из очереди извлекается следующая вершина, процесс повторяется.

Ключевая разница между обходом в глубину и в ширину заключается в порядке обхода: при поиске в глубину мы "углубляемся"в граф, а при поиске в ширину мы идём вширь.

В ходе обхода будет посещена каждая вершина в компоненте и обход попробует пройти по каждому ребру. Поэтому сложность алгоритма составит $\mathcal{O}(|V| + |E|)$, где V - множество вершин в компоненте связности, E - множество ребер в компоненте.

Если граф несвязный (имеется две и более компонент связности), то требуется выполнить обход в каждой компоненте. Пусть мы начали с какой-то компоненты. Тогда мы обошли все вершины в ней и все они были помечены нами. В таком случае какая-то вершина в другой компоненте осталась непомеченной. Начав обход заново обход с непомеченной вершины, мы обойдем еще не пройденную компоненту. Таким образом будем продолжать, пока все вершины не будут пройдены.

Независимо от того, сколько имеется компонент связности, итоговая сложность составит $\mathcal{O}(n + m)$ времени и $\mathcal{O}(n)$ памяти. Учитывая время на сортировку компонент, получим временную сложность $\mathcal{O}(n \cdot \log(n) + m)$.

Исходный код

Для реализации обходов я использовал класс графа. Граф хранится в виде списка смежности.

Для реализации обхода в глубину я использовал очередь и вектор типа *bool* посещенных вершин. Ищем первый непосещенный элемент и кладем его в очередь. Пока очередь не пуста, вынимаем из очереди вершину и смотрим кладем в очередь и компоненту (она представлена в виде вектора из вершин) все непосещенные вершины, которые соединены ребром с данной. После того, как очередь осталась пуста, кладем компоненту в ответ и запускаем тот же процесс.

```
#ifndef SRC_GRAPH_H
#define SRC_GRAPH_H

#include <vector>

class Graph {
public:
    Graph(const int &n, const std::vector<std::vector<int>> &list);

    [[nodiscard]] std::vector<std::vector<int>> findConnectivityComponents()
↪   const;

    [[nodiscard]] std::vector<std::vector<int>>
↪   findConnectivityComponentsByDFS() const;

private:
    void DFS(const int &n, std::vector<bool> &used, std::vector<int>
↪   &component) const;

    static bool hasUnused(const std::vector<bool> &used);

    int n_;
    std::vector<std::vector<int>> adjacencyList_;
};


#endif //SRC_GRAPH_H
```

```
#include "Graph.h"

#include <algorithm>
#include <queue>
```

```
Graph::Graph(const int &n, const std::vector<std::vector<int>> &list) :
↪  n_(n), adjacencyList_(list) {}


std::vector<std::vector<int>> Graph::findConnectivityComponents() const {
    std::vector<bool> used(n_, false);
    std::vector<std::vector<int>> answer;
    std::vector<int> new_component;
    std::queue<int> queue;
    int last = 0;
    do {
        new_component.clear();
        int n = 0;
        for (int i = last; i < n_; ++i) { // выбираем вершину, с которой
↪  будем начинать поиск в глубину
            if (!used[i]) {
                n = i;
                break;
            }
        }
        last = n;
        new_component.emplace_back(n);
        used[n] = true;
        queue.emplace(n);
        while (!queue.empty()) {
            int vert = queue.front();
            queue.pop();
            for (const auto &to: adjacencyList_[vert]) {
                if (!used[to]) {
                    queue.emplace(to);
                    used[to] = true;
                    new_component.emplace_back(to);
                }
            }
        }
        answer.emplace_back(new_component);
    } while (hasUnused(used));
    return answer;
}

bool Graph::hasUnused(const std::vector<bool> &used) {
    if (std::any_of(used.begin(), used.end(), [](const bool &ok) { return
↪  !ok; })) {
```

```
        return true;
    }
    return false;
}

std::vector<std::vector<int>> Graph::findConnectivityComponentsByDFS() const
↪   {
    std::vector<bool> used(n_, false);
    std::vector<std::vector<int>> answer;
    std::vector<int> new_component;
    int last = 0;
    int n = 0;
    do {
        new_component.clear();
        for (int i = last; i < n_; ++i) {
            if (!used[i]) {
                n = i;
                break;
            }
        }
        last = n;
        DFS(n, used, new_component);
        answer.emplace_back(new_component);
    } while (hasUnused(used));
    return answer;
}

void Graph::DFS(const int &n, std::vector<bool> &used, std::vector<int>
↪   &component) const {
    used[n] = true;
    component.emplace_back(n);
    for (const auto &vert: adjacencyList_[n]) {
        if (!used[vert]) {
            DFS(vert, used, component);
        }
    }
}
```

```
#include <iostream>
#include <algorithm>
```

```cpp
#include <vector>

#include "Graph.h"

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);
//    std::cout << "Hello, World!" << std::endl;
    int n, m;
    std::cin >> n >> m;
    std::vector<std::vector<int>> adjacencyList(n);
    int from, to;
    for (int i = 0; i < m; ++i) {
        std::cin >> from >> to;
        --from;
        --to;
        adjacencyList[from].emplace_back(to);
        adjacencyList[to].emplace_back(from);
    }
    Graph graph(n, adjacencyList);
//    auto startBFS = std::chrono::high_resolution_clock::now();
//    auto answer = graph.findConnectivityComponents();
//    for (auto &ans: answer) {
//        std::sort(ans.begin(), ans.end());
//    }
//    std::sort(answer.begin(), answer.end(),
//              [](const std::vector<int> &left, const std::vector<int>
↪  &right) { return left[0] < right[0]; });
//    for (const auto &ans: answer) {
//        for (const auto &vert: ans) {
//            std::cout << vert + 1 << " ";
//        }
//        std::cout << '\n';
//    }
//    auto endBFS = std::chrono::high_resolution_clock::now();
//    std::cout << "BFS TIME: " <<
↪  std::chrono::duration_cast<std::chrono::microseconds>(endBFS -
↪  startBFS).count()
//              << "[microsec]" << std::endl;
//    auto startDFS = std::chrono::high_resolution_clock::now();
    auto answerDFS = graph.findConnectivityComponentsByDFS();
```

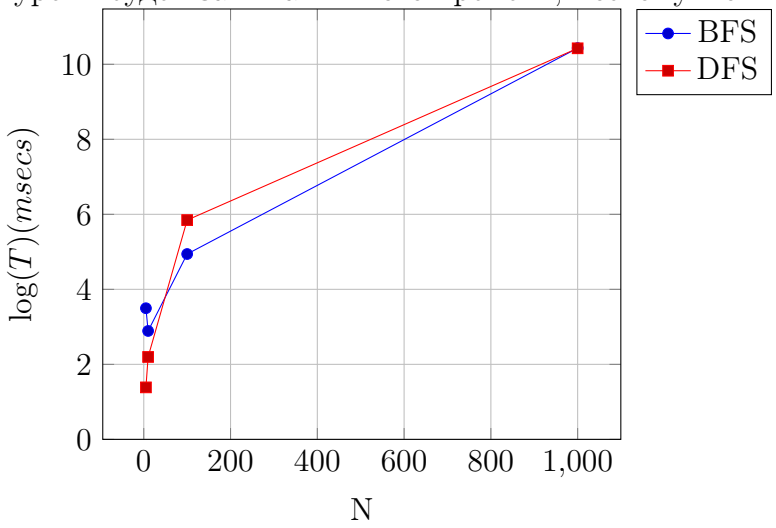
```
    for (auto &ans: answerDFS) {
        std::sort(ans.begin(), ans.end());
    }
    std::sort(answerDFS.begin(), answerDFS.end(),
              [](const std::vector<int> &left, const std::vector<int>
↪   &right) { return left[0] < right[0]; });
    for (const auto &ans: answerDFS) {
        for (const auto &vert: ans) {
            std::cout << vert + 1 << " ";
        }
        std::cout << '\n';
    }
//    auto endDFS = std::chrono::high_resolution_clock::now();
//    std::cout << "DFS TIME: " <<
↪   std::chrono::duration_cast<std::chrono::microseconds>(endDFS -
↪   startDFS).count()
//              << "[microsec]" << std::endl;
    return 0;
}
```

Дневник отладки

Во время выполнения задачи я столкнулся с проблемой в виде ограничения по времени. Как выяснилось, я передавал в конструктор класса *Graph* список смежности не по ссылке, а по значению, из-за чего происходило долгое копирование вектора.

Тест производительности

Я решил сравнить обход в ширину, который я использовал, с рекурсивным обходом в глубину. Результаты оказались неожиданными, т.к. я предполагал, что выход из рекурсии будет занимать много времени, поэтому мой вариант будет работать быстрее.



Выводы

В результате выполнения лабораторной работы я изучил способы представления графа в компьютере и базовые алгоритмы на графах: обход в ширину и в глубину.

Мы сталкиваемся с графами каждый день, прокладывая маршрут от дома до института (поиск кратчайшего пути в графе), да даже в социальных сетях можно увидеть графы! Компоненты связности можно увидеть в графе друзей в небезызвестной социальной сети "В Контакте". Вершинами являются пользователи соц. сети, а ребрами соединены пользователи, состоящие в друзьях. Поэтому графы и связанные с ними задачи максимально близки к жизни и требуют эффективных алгоритмов решения задач.