Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №9 по курсу «Дискретный анализ»

Студент: А. Т. Бахарев
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2018

Лабораторная работа №9

Задача: Разработать программу на языке C или C++, реализующую поиск компонент связности в неориентированном невзвешенном графе.

Формат входных данных: В первой строке заданы $n \leq 105$, $1 \leq m \leq 105$. В следующих m строках записаны ребра. Каждая строка содержит пару чисел – номера вершин, соединенных ребром.

Формат выходных данных: Каждую компоненту связности нужно выводить в отдельной строке, в виде списка номеров вершин через пробел. Строки при выводе должны быть отсортированы по минимальному номеру вершины в компоненте, числа в одной строке также должны быть отсортированы.

1 Описание

В общем смысле граф представляется как множество вершин (узлов), соединённых рёбрами. В строгом определении графом называется такая пара множеств $G = (V, E)$, где V есть подмножество любого счётного множества, а E — подмножество $V * V$.

Виды графов:

• Неориентированный

• Ориентированный

Отношение связности, компоненты связности: Две вершины u и v называются связаными, если в графе G существует путь из u в v .

Путь, цепь, цикл:

• Маршрут в графе — это чередующаяся последовательность вершин и рёбер, в которой любые два соседних элемента инцидентны. Если $v_0 = v_k$, то маршрут замкнут, иначе открыт.

• Простая цепь — маршрут, в котором все вершины различны.

•Простой граф — граф, в котором нет кратных рёбер и петель.

• Простой путь — путь, все рёбра которого попарно различны. Другими словами, простой путь не проходит дважды через одно ребро.

• Простой цикл — цикл, не проходящий дважды через одну вершину.

• Псевдограф — граф, который может содержать петли и/или кратные рёбра.

• Путь — последовательность рёбер (в неориентированном графе) и/или дуг (в ориентированном графе), такая, что конец одной дуги (ребра) является началом другой дуги (ребра). Или последовательность вершин и дуг (рёбер), в которой каждый элемент инцидентен предыдущему и последующему. Может рассматриваться как частный случай маршрута.

2 Исходный код

Для поиска компонент связности я использовал алгоритм поиска в ширину. Для каждой вершины, если мы еще там не были, запускался BFS. Одновременно записывался путь, по которому мы идем. Если обход заканчивался, значит мы нашли компоненту связности.

main.cpp

```
#include "TGraph.h"

int main() {
    TGraph graph;
    std::cin >> graph;
    std::vector<std::set<std::size_t> > answer = graph.FindComponents();

    for (int i = 0; i < answer.size(); i++) {
        std::set<std::size_t>::iterator it = answer[i].end();
        it--;
        for (std::set<std::size_t>::iterator j = answer[i].begin(); j != it; j++) {
            std::cout << *j+1 << " ";
        }

        std::cout <<*it+1 <<std::endl;
    }
    return 0;
}
```

TGraph.cpp

```
void TGraph::DFS(std::size_t start, std::vector<bool>& used_vertex) {
    used_vertex[start] = true;
    components[components.size() - 1].insert(start);
    for (std::set<std::size_t>::iterator it = data[start].begin();
                                it != data[start].end(); ++it) {

        if (used_vertex[*it] == false)
            DFS(*it, used_vertex);
    }
}
std::vector<std::set<std::size_t> > TGraph::FindComponents() {
    std::size_t size = data.size();
    std::vector<bool> used_vertex(size, false);

    for (std::size_t i = 0; i < size; ++i)
        if (used_vertex[i] == false) {
            components.emplace_back();
            DFS(i, used_vertex);

        }
```

```
    return components;
}

TGraph::TGraph() = default;

std::istream &operator>>(std::istream &is, TGraph &graph){
    std::size_t n, m;
    is >> n >> m;
    graph.data.resize(n);
    for (int i = 0; i < m; i++) {
        std::size_t x, y;
        is >> x >> y;
        graph.data[x-1].insert(y-1);
        graph.data[y-1].insert(x-1);
    }
    return is;
}
```

TGraph.h

```
#include <vector>
#include <istream>
#include <set>

class TGraph {
private:
    std::vector<std::set<std::size_t> > data;
    std::vector<std::set<std::size_t> > components;
    std::size_t size;
public:
    explicit TGraph();

    std::vector<std::set<std::size_t> > FindComponents();
    void DFS(std::size_t start, std::vector<bool>& used_vertex);
    friend std::istream &operator>>(std::istream &is, TGraph &graph);
    ~TGraph() {}
};
```

3 Консоль

```
alex$ make
g++ -std=c++11 -pedantic -Werror -Wno-sign-compare -Wno-long-long -lm -c TGraph.cpp
g++ -std=c++11 -pedantic -Werror -Wno-sign-compare -Wno-long-long -lm -o lab9
main.cpp TGraph.o
alex$ ./lab9
5 4
1 2
2 3
1 3
4 5
1 2 3
4 5
```

4 Выводы

Выполнив девятую лабораторную работу по курсу «Дискретный анализ», я разобрался с такой структурой данных как графы. Узнал их особенности, различия, принцип работы с ними. В процессе написания данной работы, я познакомился также с другими алгоритмами, которые позволяют эффективно работать с графами: Поиск в глубину, алгоритм Флойда-Уоршелла, алгоритм Дейкстры.

Список литературы

[1] *Поиск в ширину.*
URL: `https://e-maxx.ru/algo/bfs`