Лабораторная работа № 9 по курсу: Дискретный анализ

Выполнил студент группы М8О-307Б-17 МАИ *Лопатин Александр.*

Задача

Вариант 3 : Поиск компонент связности
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.

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

Выходные данные: 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.

Информация

Компонента связности графа G (или просто компонента графа G) — максимальный (по включению) связный подграф графа G.

Другими словами, это подграф G(U), порождённый множеством $U \subseteq V(G)$ вершин, в котором для любой пары вершин u, $v \in U$ в графе G существует (u,v)-цепь и для любой пары вершин $u \in U$, $w \notin U$ не существует (u, w)-цепи.

Для ориентированных графов определено понятие компоненты сильной связности.

Метод решения

Для того, чтобы выделить компоненты связности, нужно обойти все вершины графа. Для этой цели могут подойти алгоритмы обхода в ширину/глубину, но для простоты реализации алгоритма был выбран поиск в глубину. Достаточно для каждой вершины выполнить метод поиска в глубину, запомнить каждую обрабатываемую вершину и поменять её цвет. Во-первых, цвет нужен для работы поиска в глубину, а во-вторых, цвет вершины даст нам понять, была ли уже рассмотрена компонента связности, которой эта вершина принадлежит. Благодаря этому гарантируется сложность по времени $O(n*m)$, где n - количество вершин, m - количество ребер в графе.

Исходный код

lab9.cpp

```
#include <iostream>
#include <fstream>
#include <set>
#include <vector>
using namespace std;

struct TVertex {
    bool Color = 1;
    vector<int> Edges;
};

vector<TVertex> vertices;

set<int> DFS(TVertex u) {
    set<int> result;
    for(int i = 0; i < u.Edges.size(); i++) {
        if(vertices[u.Edges[i]].Color) {
            vertices[u.Edges[i]].Color = 0;
            result.insert(u.Edges[i] + 1);
            set<int> subResult = DFS(vertices[u.Edges[i]]);
            result.insert(subResult.begin(), subResult.end());
        }
    }
    return result;
}

vector<set<int>> FindComponents() {
    vector<set<int>> result;
    for( int i = 0; i < vertices.size(); i++) {
        if(vertices[i].Color) {
            set<int> component = DFS(vertices[i]);
            component.insert(i + 1);
            vertices[i].Color = 0;
            result.push_back(component);
        }
    }
    return result;
}

int main(int argc, char *argv[]) {
    ifstream inFile(argv[1]);
    int n, m = 0;
    inFile >> n >> m;
    for( int i = 0; i < n; i++) {
        vertices.push_back(TVertex());
    }
    for( int i = 0; i < m; i++) {
        int first, second = 0;
        inFile >> first >> second;
        vertices[first-1].Edges.push_back(second-1);
        vertices[second-1].Edges.push_back(first-1);
```

```
    }
    inFile.close();
    vector<set<int>> result = FindComponents();
    ofstream outFile(argv[2]);
    for( int i = 0; i < result.size(); i++) {
        set<int>::iterator it;
        for (it = result[i].begin(); it != result[i].end(); ++it) {
            outFile << *it << ' ';
        }
        outFile << endl;
    }
    outFile.close();
}
```

Генератор тестов

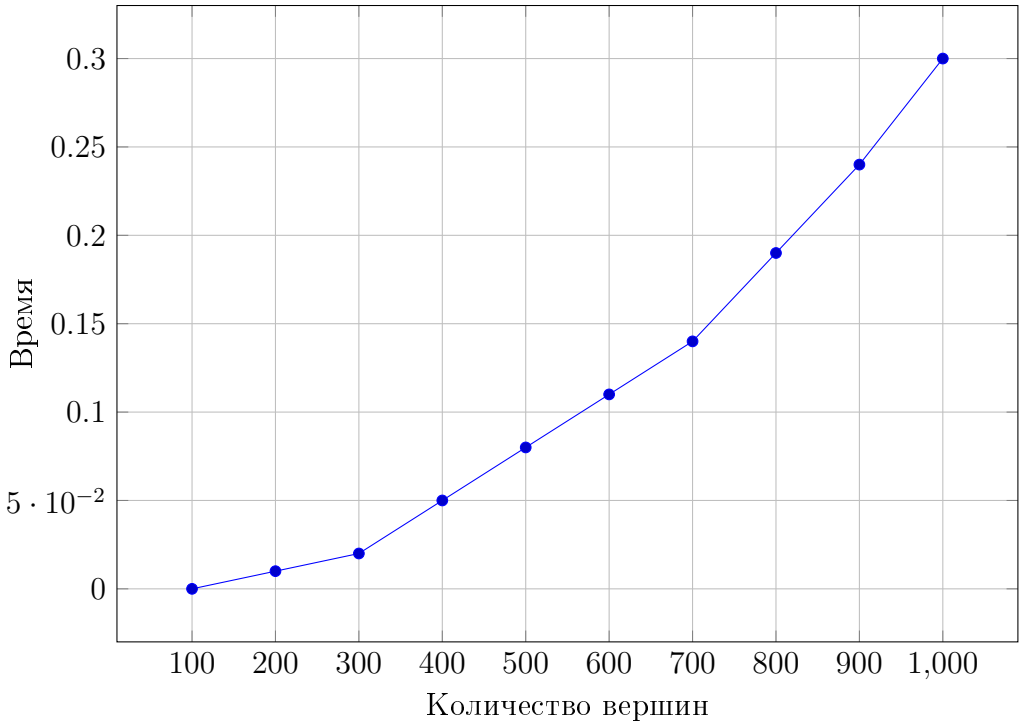
Тесты создаются следующим образом: задается количество вершин, и наличие ребра у двух любых вершин задается с вероятностью 0.25. Это означает, что количество ребер m будет примерно равно 0.25n, т.е. будет расти линейно от n.

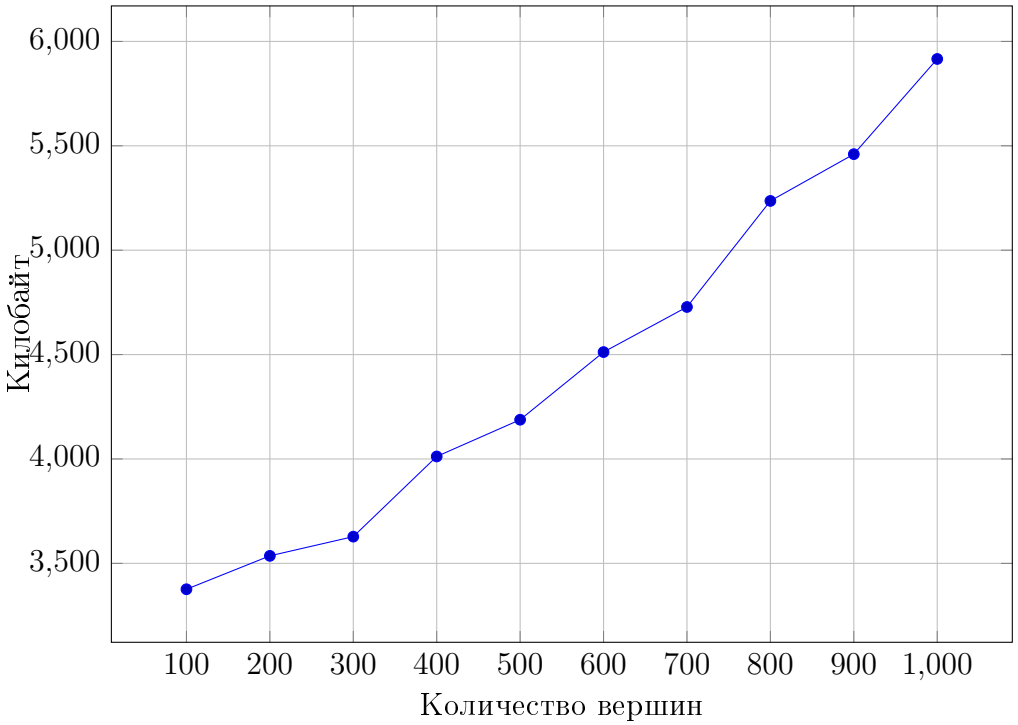
tests.py

```
import random
size = 100;
for i in range(1, 11, 1):
    countOfEdges = 0
    edges = []
    file = open('tests/test'+str(i)+'.txt','w')
    file.write(str(size*i) + ' ')
    for j in range(1, size*i, 1):
        for k in range(j + 1, size*i, 1):
            r = random.randint(1, 4)
            if (r == 1) :
                edges.append(str(j) + ' ' + str(k))
                countOfEdges += 1
    file.write(str(countOfEdges) + '\n')
    for s in edges:
        file.write(s + '\n')
    file.close()
```

Тест производительности

Так как сложность алгоритма поиска в глубину составляет $O(n*m)$, а m примерно равно 0.25n, то ожидаемое время работы будет расти квадратично.


0.3
0.25
0.2
0.15
0.1
$5 \cdot 10^{-2}$
0
Время
100
200
300
400
500
600
700
800
900
1,000
Количество вершин


6,000
5,500
5,000
4,500
4,000
3,500
Килобайт
100
200
300
400
500
600
700
800
900
1,000
Количество вершин

Выводы

Алгоритм поиска компонент связности достаточно прост, так как достаточно выполнить поиск в глубину для каждой вершины графа. Реализовать алгоритм поиска в глубину не составило труда.