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: | М. М. Касимов |
| Преподаватель: | А. А. Кухтичев |
| Группа: | М8О-206Б |
| Дата: | |
| Оценка: | |
| Подпись: | |



**Москва, 2020**

# Лабораторная работа №9

**Задача:** Разработать жадный алгоритм решения задачи, определяемой своим вариантом. Оценить сложность по времени и объём затрачиваемой оперативной памяти.

**Вариант задачи**

Задан неориентированный двудольный граф, состоящий из $n$ вершин и $m$ ребер. Вершины пронумерованы целыми числами от 1 до $n$. Необходимо найти максимальное паросочетание в графе алгоритмом Куна. Для обеспечения однозначности ответа списки смежности графа следует предварительно отсортировать. Граф не содержит петель и кратных ребер.

**Формат входных данных**

В первой строке заданы $1 \leq n \leq 110000$ и $1 \leq m \leq 40000$. В следующих $m$ строках записаны ребра. Каждая строка содержит пару чисел – номера вершин, соединенных ребром.

**Формат результата**

В первой строке следует вывести число ребер в найденном паросочетании. В следующих строках нужно вывести сами ребра, по одному в строке. Каждое ребро представляется парой чисел – номерами соответствующих вершин. Строки должны быть отсортированы по минимальному номеру вершины на ребре. Пары чисел в одной строке также должны быть отсортированы.

# 1 Описание

Требуется реализовать алгоритм Куна для нахождения максимального паросочетания в двудольном графе, разбиение на доли в графе не задано. Для начала требуется найти разбиение графа на доли. Сам двудольный граф хранится посредством массива списков вершин. Разбиение графа будем находить следующим образом – запустим поиск в ширину и будем смотреть получающиеся пути в ходе обхода этого графа, по пути будем окрашивать вершины в разные цвета, то есть если зашли в вершину на нечётном шаге – она относится к левой доле, если на четном – к правой. Таким образом за линейную сложность от количества вершин получим разбиение графа. Далее применим алгоритм Куна, который основан на теореме Бержа - паросочетание является максимальным тогда и только тогда, когда не существует увеличивающих относительно него цепей. Увеличивающаяся цепь- чередующуюся цепь, у которой начальная и конечная вершины не принадлежат паросочетанию. Чередующаяся цепь- цепь, в которой рёбра поочередно принадлежат не принадле-жат паросочетанию. Цепь – простой путь в графе, который не содержит повторяющихся вершин или ребер. Изначально зададим пустое паросочетание. Далее пытаемся найти увеличивающуюся цепь в графе, если таковая имеется выполняем чередование паросочетания вдоль этой цепи. Повторяем процесс, пока не найдем максимальную цепь. Искать увеличиващуюся цепь будем с помощью обхода в глубину следующим образом – если все вершины из текущей уже были посещены, то найдена максимальная цепь, иначе переходим в следующую вершину и формируем увеличивающуюся цепь. Этот обход запускаем от всех вершин левой доли, так как в графе могут быть несвязанные пути и от другой вершины может быть найдена большая цепь. Итоговая сложность $O(nm)$, n - кол-во вершин в первой доле, m - всего вершин. По итогу получаем массив в котором заданы итоговые паросочетания.

## 2 Исходный код

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <algorithm>
#include <queue>

bool DFS(int v, std::set<int>& used, std::vector<std::vector<int>>& graph, std::vector
    <int>& matching) {
    if (used.count(v)) {
        return false;
    }
    used.insert(v);
    for (int& elem : graph[v]) {
        if (matching[elem] == -1 || DFS(matching[elem], used, graph, matching)) {
            matching[elem] = v;
            return true;
        }
    }
    return false;
}

std::vector<std::pair<int,int>> KuhnAlgorithm(std::vector<std::vector<int>>& graph) {
    std::vector<int> matching (graph.size(), -1);
    std::set<int> used;
    for (int i = 0; i < graph.size(); ++i) {
        used.clear();
        DFS(i, used, graph, matching);
    }
    std::vector<std::pair<int, int>> answer;
    for (int i = 0; i < matching.size(); ++i) {
        if (matching[i] != -1) {
            answer.push_back(std::make_pair(std::min(i, matching[i]), std::max(i,
                matching[i])));
        }
    }
    std::sort(answer.begin(), answer.end(), [](std::pair<int,int> l, std::pair<int, int
        > r){ return l.first < r.first; });
    return answer;

}


std::vector<int> SplitForPart(std::vector<std::vector<int>>& graph) {
    std::vector<int> part(graph.size(), -1);
    std::vector<bool> used(graph.size(), false);
    std::queue<int> queue;
```

```
    for (int i = 0; i < graph.size(); ++i) {
        if (part[i] == -1) {
            part[i] = 0;
            queue.push(i);
            used[i] = true;
            while (!queue.empty()) {
                int cur = queue.front();
                queue.pop();
                int parent = cur;
                for (int j = 0; j < graph[cur].size(); ++j) {
                    if (part[graph[cur][j]] == -1 && !used[graph[cur][j]]) {
                        part[graph[cur][j]] = !part[parent];
                        used[graph[cur][j]] = true;
                        queue.push(graph[cur][j]);
                    }
                }
            }
        }
    }
    return part;

}

int main() {
    int n, m, begin, end;
    std::cin >> n >> m;
    std::vector<std::vector<int>> graph(n);

    for (int i = 0; i < m; ++i) {
        std::cin >> begin >> end;
        graph[begin - 1].push_back(end - 1);
        graph[end - 1].push_back(begin - 1);
    }
    std::vector<int> part = SplitForPart(graph);
    std::vector<std::vector<int>> biGraph (graph.size());
    for (size_t i = 0; i < graph.size(); ++i) {
        if (!graph[i].empty())
            std::sort(graph[i].begin(),graph[i].end());
    }
    for (int i = 0; i < graph.size(); ++i) {
        if (!part[i]) {
            biGraph[i] = graph[i];
        }
    }
    std::vector<std::pair<int, int>> result = std::move(KuhnAlgorithm(biGraph));
    std::cout << result.size() << '\n';
    for (const std::pair<int, int>& pair : result) {
        std::cout << pair.first + 1 << ' ' << pair.second + 1 << '\n';
    }
```

```
    return 0;
}
```

# 3 Консоль

```
magomed@magomed-TM1701:~/programming/C++/da_lab9$ ./a.out
4 3
1 2
2 3
3 4
2
1 2
3 4
magomed@magomed-TM1701:~/programming/C++/da_lab9$ ./a.out
3 2
1 2
1 3
1
1 2
```

# 4 Выводы

Данная лабораторная работа стала для меня первым опытом работы с алгоритмами на графах. Я узнал что такое двудольный граф и реализовал алгоритм Куна для поиска максимального паросочетания.