Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Выполнил:

Студент группы P3212

Куприянов А.А

Преподаватели:

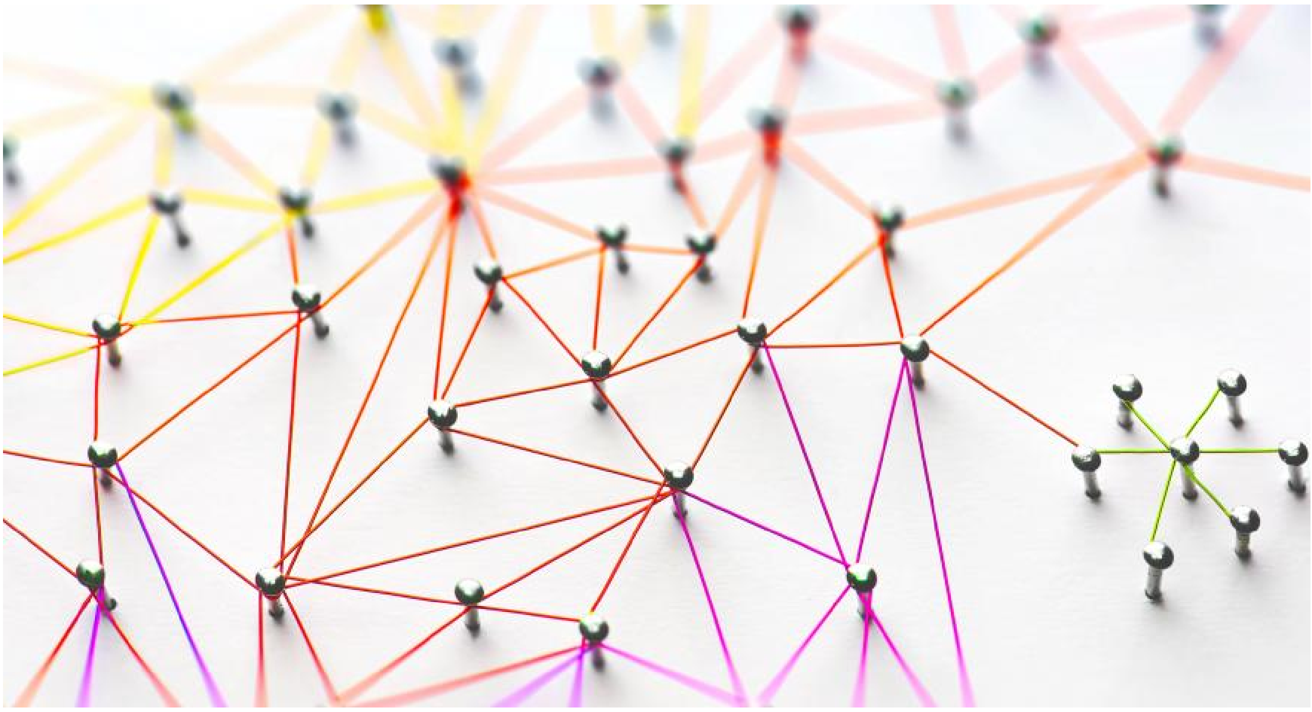
Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2020

Network 1160



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

struct e {
    int a, b, l;
};

int rank[1010], parent[1010];
int max = 0;

bool cmp(e a, e b) {
    return a.l < b.l;
}

int find(int x) {
    if (x != parent[x]) parent[x] = find(parent[x]);

    return parent[x];
}

int main() {
    int n, m;

    std::cin >> n >> m;
    std::vector<e> v;
```

```
    for (int i = 0; i < m; ++i) {
        int a, b, l;
        std::cin >> a >> b >> l;
        v.push_back({a - 1, b - 1, l});
    }

    std::sort(v.begin(), v.end(), cmp);

    for (int i = 1; i <= n; i++) {
        rank[i] = 0;
        parent[i] = i; // map to self
    }

    for (int i = 0; i < m; ++i) {
        int n1 = v[i].a;
        int n2 = v[i].b;

        if (find(n1) != find(n2)) {
            if (v[i].l > max) {
                max = v[i].l;
            }
            v[i].l *= -1; // put mark for cout in the end

            int x = find(n1);
            int y = find(n2);
            if (rank[x] > rank[y]) parent[y] = x;
            else {
                parent[x] = y;
                if (rank[x] == rank[y]) rank[y]++;
            }
        }
    }

    std::cout << max << "\n" << n - 1 << "\n";

    for (int j = 0; j < m; ++j) {
        if (v[j].l < 0) {   // print marked edges
            std::cout << v[j].a + 1 << " " << v[j].b + 1 << "\n";
        }
    }
    return 0;
}
```

Эту задачу я решил применив метод Краскала. Потому что граф здесь разреженный (количество вершин < 10_000 и количество ребер <15_000), а также сама задача подразумевает найти максимальную длину одного кабеля, когда как в алгоритме Краскала - это последний добавленный кабель.

Алгоритм Крускала изначально помещает каждую вершину в своё дерево, а затем постепенно объединяет эти деревья, объединяя на каждой итерации два некоторых дерева некоторым ребром.

Перед началом выполнения алгоритма, все рёбра сортируются по весу (в порядке неубывания).

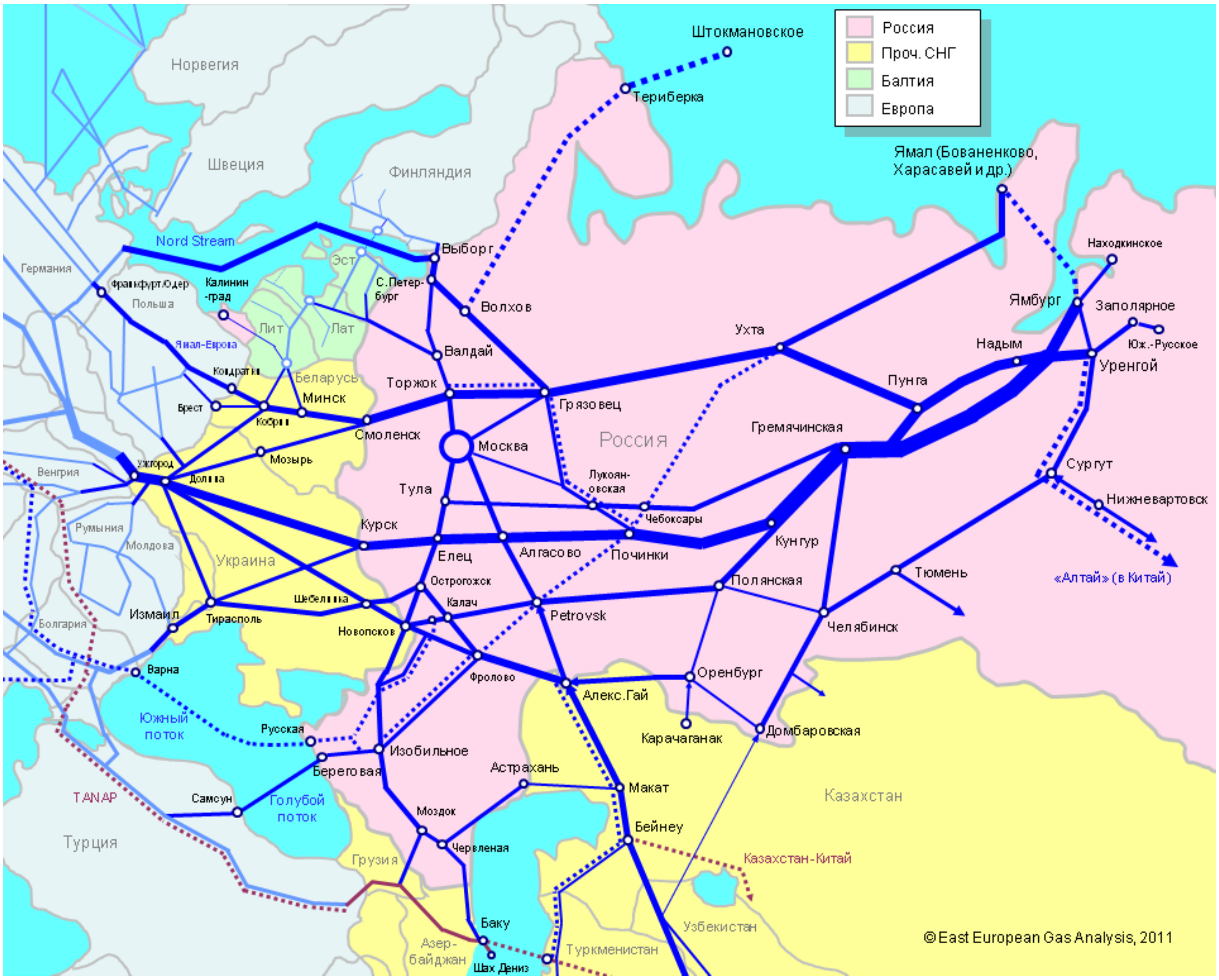
Затем начинается процесс объединения: перебираются все рёбра от первого до последнего (в порядке сортировки), и если у текущего ребра его концы принадлежат разным поддеревьям, то эти поддеревья объединяются, а ребро добавляется к ответу.

По окончании перебора всех рёбер все вершины окажутся принадлежащими одному поддереву, и ответ найден.

Чтобы найти сложность алгоритма можно выделить две основные операции : сортировка и нахождение конца дерева. Для сортировки мы получаем сложность log(E * logE), далее для второй части используется обратная функция аккермана, которая меньше 5, что можно взять за константу.

Следовательно, получим сложность : O(E(logE+α(V)))=O(E* logE)

Газопроводы России 1450




```
#include <iostream>
#include <vector>
```

```cpp
using namespace std;

struct edge {
   int a, b, w;
};

vector<edge> v;

int main() {

   ios_base::sync_with_stdio(0);
   cin.tie(0);
   cout.tie(0);
   int n, m;

   cin >> n >> m;
   for (int i = 0; i < m; ++i) {
      int a, b, w;
      cin >> a >> b >> w;
      v.push_back({a - 1, b - 1, w});
   }

   int s, f;

   cin >> s >> f;
   s--;
   f--;

   vector<int> dArray(510, -1);

   dArray[s] = 0;

   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < m; j++) {
         if (dArray[v[j].a] != -1 && dArray[v[j].b] < dArray[v[j].a] + v[j].w) {
            dArray[v[j].b] = dArray[v[j].a] + v[j].w;
         }
      }
   }

   if (dArray[f] != -1) {
      cout << dArray[f];
   } else {
      cout << "No solution";
   }

   return 0;
}
```

Здесь я применил алгоритм Беллмана-Форда, только вместо нахождения минимального пути использовал проверку для нахождения максимального пути.

Записываем в вершины весы и если новое значение больше, чем предыдущее, то перезаписываем. Очень похоже на алгоритм Дейкстры.

В алгоритме используется два цикла (один вложенный). Сначала идет цикл по вершинам, а затем для каждой вершины цикл по ребрам, следовательно сложность: O(V * E)

Раскраска 1080

```
#include <iostream>
#include <bits/stdc++.h>

using namespace std;

vector<int> color(100, -1);
vector<int> edge[100];
int n;

void bfsCheck(int st) {
    queue<int> q;
    q.push(st);
    color[st] = 0;

    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int i = 0; i < edge[v].size(); i++) {
```

```
            int to = edge[v][i];
            if (color[v] == color[to]) {
                cout << "-1";
                exit(0);
            }
            if (color[to] == -1) {
                color[to] = !color[v];
                q.push(to);
            }
        }
    }
}

int main() {
    cin >> n;

    for (int i = 0; i < n; i++) {
        int e = -1;
        while (e != 0) {
            cin >> e;

            if (e != 0) {
                edge[i].push_back(e - 1);
                edge[e - 1].push_back(i);
            }
        }
    }
    for (int i = 0; i < n; i++) {
        if (color[i] == -1) {
            // call bfs if graph not fully linked
             bfsCheck(i);
        }
    }
    for (int i = 0; i < n; i++) {
        cout << color[i];
    }
    return 0;
}
```

Обходим граф в ширину, но я не осмелился назвать свой алгоритм просто bfs, так как внутри с этого метода программа может завершить свою работу, если нельзя произвести раскраску: если вершина уже покрашена в цвет, и он не соответствует нашему, то граф не двудольный и такой граф нельзя раскрасить в два цвета

Здесь подводным камнем может стать то, что граф может быть несвязным. Такое происходит, если есть не посещенные вершины. Для них по отдельности вызовем обход, поэтому есть проверка с массивом color, которая по умолчанию имеет значение -1.

Сложность алгоритма O(V + E), если граф связный