Факультет программной инженерии и компьютерной техники

Алгоритмы и структуры данных

Алгоритмы на графах

Преподаватели: Косяков Михаил Сергеевич, Тараканов Денис Сергеевич

Выполнил: Кульбако Артемий Юрьевич

Р3212

# 1080. Раскраска карты

#include <vector>

#include <queue>

#include <iostream>

using namespace std;

struct Country {

    int color = -1;*// -1 - не посещен, 0 - красный, 1 - синий*

    vector<int> borders;

};

void BFSAndColoring(Country\* *countries*, int *st*) {

    queue<int> q;

    q.push(st);

    countries[st].color = 0;

    while (!q.empty()) {

        int v = q.front();

        q.pop();

        for (int i = 0; i < countries[v].borders.size(); i++) {

            int to = countries[v].borders[i];

            if (countries[v].color == countries[to].color) {

                cout << "-1";

                exit(0);

            }

            if (countries[to].color == -1) {

                countries[to].color = (countries[v].color == 0 ? 1 : 0);

                q.push(to);

            }

        }

    }

}

int main() {

    int n;

    cin >> n;

    Country lands[n + 1];

    for (int i = 1; i <= n; i++) {

        int e = -1;

        while (e != 0) {

            cin >> e;

            if (e != 0) {

                lands[i].borders.push\_back(e);

                lands[e].borders.push\_back(i);

            }

        }

    }

    for (int i = 1; i <= n; i++) {

        if (lands[i].color == -1) BFSAndColoring(lands, i);

        cout << lands[i].color;

    }

    return 0;

}

Необходимо реализовать алгоритм раскраски графа в 2 цвета так, чтобы смежные вершины не могли быть раскрашены в один цвет. Создадим структуру данных Country, которая будет содержать данные о своём цвете и номера стран, с которыми имеет границы. На вход алгоритма подаём заполненный невзвешенный граф и номер стартовой вершины . На каждом шаге красим одну вершину в определённый цвет в зависимости от того, возможно ли это по условию. В классической реализации необходимо иметь также массив used[], который содержит состояние прохода вершины, т.е. была ли она посещена или нет, чтоб исключить добавления её в очередь при наличии циклов в графе. В данной задаче от этого можно избавиться, условно покрасив множество в третий цвет до прохода в ширину (-1 как-раз может имитировать этот цвет). Сделать это нужно в цикле, так-как граф может быть несвязанным, а значит одного прохода может оказаться недостаточно. Запустив алгоритм от каждой непокрашенной вершины, мы точно покроим всем граф.

**Визуализация работы на входном примере:**

1. Заполненная карта.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | № | 1 | 2 | 3 | | 1 |  | x |  | | 2 | x |  | x | | 3 |  | x |  | | |  |  |  |  | | --- | --- | --- | --- | | **color:** | 0 | -1 | -1 | | |  | | --- | | **queue:** | |
| 1. Итерации: | | |
| |  |  |  |  | | --- | --- | --- | --- | | № | 1 | 2 | 3 | | 1 |  | x |  | | 2 | x |  | x | | 3 |  | x |  | | |  |  |  |  | | --- | --- | --- | --- | | **color:** | 0 | -1 | -1 | | |  |  | | --- | --- | | **queue:** | 1 | |
|  | | |
| |  |  |  |  | | --- | --- | --- | --- | |  | 1 | 2 | 3 | | 1 |  | x |  | | 2 | x |  | x | | 3 |  | x |  | | |  |  |  |  | | --- | --- | --- | --- | | **color:** | 0 | 1 | -1 | | |  |  | | --- | --- | | **queue:** | 2 | |
|  | | |
| |  |  |  |  | | --- | --- | --- | --- | |  | 1 | 2 | 3 | | 1 |  | x |  | | 2 | x |  | x | | 3 |  | x |  | | |  |  |  |  | | --- | --- | --- | --- | | **color:** | 0 | 1 | 0 | | |  |  | | --- | --- | | **queue:** | 3 | |

# 1450. Российские газопроводы

#include <iostream>

#include <string>

#include <vector>

using namespace std;

struct GasPipeline {

    int from, to, price;

};

int main() {

    int n, m;

    cin >> n >> m;

    GasPipeline wires[m];

    int res[n \* n];

    fill\_n(res, n \* n, -1);

    for (int i = 0; i < m; i++) cin >> wires[i].from >> wires[i].to >> wires[i].price;

    int s, f;

    cin >> s >> f;

    res[s] = 0;

    for (int i = 0; i < n - 1; i++)

        for (int j = 0; j < m; j++)

            if (res[wires[j].from] != -1 && res[wires[j].to] < res[wires[j].from] + wires[j].price)

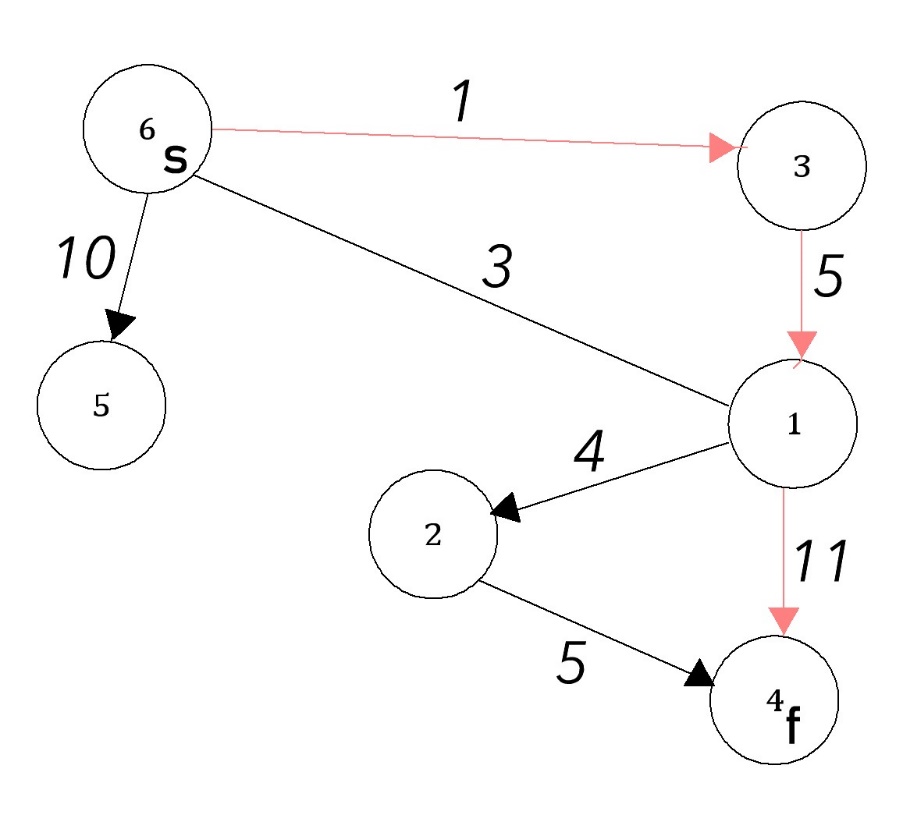
                res[wires[j].to] = res[wires[j].from] + wires[j].price;

    if (res[f] != -1) cout << res[f];

    else cout << "No solution";

    return 0;

}

Задача на алгоритм Дейкстры, с той лишь разницей, что нужно вывести не минимальный путь от до , а максимальный. Выделим память под массив res, который будет содержать путь от выделенной вершины до всех остальных вершин взвешенного ориентированного графа. Изначально res[s] = 0 — это значит, что путь из в пока ещё = 0, но необходимо для старта алгоритма, а для остальных вершин это длина равна заведомо невозможному минимальному числу . Для облегчения понимания кода создадим структуру GasPipeline, содержащую ребро графа и его вес, чтобы в явном виде не хранить массив предков являющейся предпоследней в длиннейшем пути до вершины . Найдём для узла самое большое ребро из узлов , и пересчитаем стоимость соседей этого узла. Повторим так для каждого, и в конце получим элементов res[f], который будет содержать сумму рёбер от до (если , значит оказалась непосещённой, следовательно до неё не добраться из ). Доказательство алгоритма таково: для вершины имеем res[s] = 0, что и является длиной кратчайшего пути до неё. Пусть теперь это утверждение выполнено для всех предыдущих итераций, т.е. всех уже помеченных вершин; докажем, что оно не нарушается после выполнения текущей итерации. Пусть — вершина, которую алгоритм собирается пометить на текущей итерации. , где - длине кратчайшего пути до . Рассмотрим кратчайший путь до вершины . Этот путь можно разбить на два пути: , состоящий только из помеченных вершин, и - остальная часть пути. , ведь на одной из прошлых итераций мы перебирали вершины из . Вследствие не отрицательности стоимостей рёбер длина кратчайшего пути не превосходит длины кратчайшего пути до вершины . Учитывая, что получаем: . Поскольку и — вершины непомеченные, то так как на текущей итерации была выбрана именно вершина , а не вершина , то получаем: .