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 | |

# 1160. Network

#include <iostream>

#include <algorithm>

using namespace std;

struct Cable {

    int from, to, length;

    bool operator< (const Cable& *other*) {

        return length < other.length;

    }

};

class DisjointSetUnion {

    private: pair<int, int>\* nodes;

    public:

        DisjointSetUnion(int *size*) {

            nodes = new pair<int, int>[size + 1];

            for (int i = 1; i <= size; i++) nodes[i] = {0, i};

        }

        int findRoot(int *a*) {

            if (a != nodes[a].second) nodes[a].second = findRoot(nodes[a].second);

            return nodes[a].second;

        }

        void merge(int *a*, int *b*) {

            if (nodes[a].first > nodes[b].first) nodes[b].second = a;

            else {

                nodes[a].second = b;

                if (nodes[a].first == nodes[b].first) nodes[b].first++;

            }

        }

};

int main() {

    int n, m, maxLength = 0;

    cin >> n >> m;

    Cable cables[m];

    for (int i = 0; i < m; i++) {

        int a, b, l;

        cin >> a >> b >> l;

        cables[i] = {a, b, l};

    }

    sort(cables, cables + m);*//передаём сортировке указатели на начало и конец массива*

    DisjointSetUnion\* dsu = new DisjointSetUnion(n);

    for (int i = 0; i < m; i++) {

        int from = cables[i].from;

        int to = cables[i].to;

        if (dsu->findRoot(from) != dsu->findRoot(to)) {

            if (cables[i].length > maxLength) maxLength = cables[i].length;

            cables[i].length \*= -1;

            dsu->merge(dsu->findRoot(from), dsu->findRoot(to));

        }

    }

    cout << maxLength << "\n" << n - 1 << "\n";

    for (Cable c: cables) if (c.length <= 0) cout << c.from << " " << c.to << "\n";

    return 0;

}

Задача о нахождении минимального оставного дерева в графе, где вершинами будут выступать хабы, а провода - рёбрами графа. Необходимо пройти через все вершины так, чтобы сумма рёбер была минимальной. На выбор для реализации есть три известных алгоритма: Прима, Краскала и Буровки. Реализуем Краскала. Принцип работы таков: алгоритм изначального помещает все ребра графа в собственное дерево, затем объединяя на два некоторых дерева ребром . Изначально все рёбра сортируются по весу в порядке неубывания - если у текущего ребра его концы принадлежат разным поддеревьям, то эти поддеревья объединяются. По окончании перебора все окажутся принадлежащими одному поддереву, и ответ найден.

|  |  |
| --- | --- |
| **Изображение** | **Описание** |
|  | Ребра **AD** и **CE** имеют минимальный вес, равный 5. Произвольно выбирается ребро **AD** (выделено на рисунке). В результате получаем множество выбранных вершин (**A**, **D**). |
|  | Теперь наименьший вес, равный 5, имеет ребро **CE**. Так как добавление **CE** не образует цикла, то выбираем его в качестве второго ребра. Так, как это ребро не имеет общих вершин с имеющимся множеством выбранных вершин (**A**, **D**), получаем второе множество выбранных вершин (**C**, **E**) |
|  | Аналогично выбираем ребро **DF**, вес которого равен 6. При этом не возникает ни одного цикла, так как не существует (среди невыбранных) ребра, имеющего обе вершины из одного (**A**, **D**, **F**) или второго (**C**, **E**) множества выбранных вершин. |
|  | Следующие ребра — **AB** и **BE** с весом 7. Произвольно выбирается ребро **AB**, выделенное на рисунке. Тем самым вершина **B** добавляется к первому множеству выбранных вершин (**A**, **B**, **D**, **F**). Невыбранное ранее ребро **BD** выделено красным, так как его вершины входят в множество выбранных вершин (**A**, **B**, **D**, **F**), а, следовательно, уже существует путь (зелёный) между **B** и **D** (если бы это ребро было выбрано, то образовался бы цикл **ABD**).  Следующее ребро может быть выбрано только после нахождения всех циклов. |
|  | Аналогичным образом выбирается ребро **BE**, вес которого равен 7. Так как это ребро имеет вершины в обоих множествах выбранных вершин (**A**, **B**, **D**, **F**) и (**C**, **E**), эти множества объединяются в одно (**A**, **B**, **C**, **D**, **E**, **F**). На этом этапе красным выделено гораздо больше ребер, так как множества выбранных вершин, а, следовательно, и множества выбранных рёбер объединились. Теперь **BC** создаст цикл **BCE**, **DE** создаст цикл **DEBA**, и **FE** сформирует цикл **FEBAD**. |
|  | Алгоритм завершается добавлением ребра **EG** с весом 9. Количество выбранных вершин (**A**, **B**, **C**, **D**, **E**, **F**, **G**) = 7 соответствует количеству вершин графа. [Минимальное остовное дерево](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BD%D0%B8%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BE%D1%81%D1%82%D0%BE%D0%B2%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) построено. |

Cable отвечает за удобное хранение рёбер. Их сортировка потребует операций, если использовать алгоритм из стандартной библиотеки C++. Затем нужно проверять соединяет ли ребро две различных компоненты связности графа. Эффективно реализовать это можно с помощью структуры данных DisjointSetUnion (Система непересекающихся множеств), создадим отдельный класс для работы с ней. Объединять компоненты связности будем методом merge(int a, int b), а проверять, находятся ли разные вершины графа в разных деревьях (т.е. компонентах связности) методом findRoot(int a). Для описания множества используется номер вершины, являющейся корнем соответствующего дерева. Для определения, принадлежат ли два элемента к одному и тому же множеству, для каждого элемента нужно найти корень соответствующего дерева (поднимаясь вверх пока это возможно) и сравнить эти корни. Объединяются множества так: пусть нам нужно объединить множества с корнями . Просто присвоим , (где - массив, хранящий прямого предка для каждой вершины), тем самым подвесив всё дерево к корню дерева . Можно заметить, что при такой реализации при постепенном объединении деревьев глубина будет расти вплоть до . Применим несколько оптимизаций к структуре:

# Ранги вершин: будем хранить для всех деревьев текущую глубину, и при объединении подвешивать дерево с меньшей глубиной к корню дерева с большей глубиной (в действительности, ранг вершины описывают не точную глубину дерева, а её верхнюю границу, но это не играет роли).

# Сжатие путей: при поиске корня заданной вершины будем переподвешивать её за найденный корень. К примеру, мы вызвали findRoot для вершины, которую отделяют от корня дерева пять других вершин. Рекурсивный вызов функции обойдёт каждую из них, и найдёт корень. На выходе из каждого рекурсивного вызова обновим текущую вершину только что найденным корнем. Таким образом, все пять вершин теперь будут подвешены напрямую к корню.

P.S. В моей реализации объединим массивы и массив рангов в массив объектов pair.

# В конце выведем ответ. Максимальную длину рёбра можно было получить во время прохода по массиву рёбер, количество рёбер, соединяющих все точки будет равняться , для любого количества точек (это можно заметить эмпирическим путём), а также выведем все рёбра, у которых длина , так как во время вышеописанного цикла, необходимо отмечать уже использованные в графе рёбра - для этого можно просто менять знак веса ребра на противоположный.

# 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: .

