Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P3219

Билобрам Д. А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

Задача №M «Цивилизация»

Пояснение к примененному алгоритму:

Карту мира интерпретируется как взвешенный граф. У соседних клеток на карте мира есть ребра, а длина ребра зависит от того, клетку какого типа она соединяет. Ребра, которые соединяют клетку с водяным рельефом, имеют бесконечную длину. Далее достаточно найти кратчайший путь в графе из стартовой вершины к той, где нужно построить город. Для определения кратчайшего пути используется алгоритм Дейкстры. Для доказательства корректности моего алгоритма необходима рассмотреть и доказать корректность алгоритма Дейкстры:

Дан взвешенный граф G = (V,E) с весовой функцией w : E → R+. Требуется найти кратчайшие пути от вершины-источника s ∈ V до всех остальных вершин.
Для всех вершин v ∈ V устанавливаем минимальное расстояние в бесконечность dist(v) ←∞. Минимальное расстояние от s до s известно и равно 0, dist(s) ← 0. Добавляем s в приоритетную (по кратчайшему расстоянию dist[i]) очередь Q и пока в Q есть вершины на каждой итерации получаем вершину u с минимальным кратчайшим расстоянием (ту что стоит в начале очереди), расстояние dist[u] до каждой такой вершины u на каждой итерации уже является кратчайшим от s, остается только удалить эту вершину из Q и пересчитать соседние вершины v, если v ∈ Q и dist(u) + w(u,v) < dist(v), то: dist(v) ← dist(u) + w(u,v) и добавляем (dist[v], v) в Q.

Доказательство корректности алгоритма Дейкстры через индукцию:

Алгоритм основывается на том, что когда мы выбираем и удаляем вершину u из Q с кратчайшим из кратчайших расстояний от s, мы можем быть уверенны что dist[u] уже точно является кратчайшим путем от s до u. Докажем по индукции:

• База индукции: Для начальной вершины s, dist(s) = 0, что очевидно является длиной кратчайшего пути от s до s.

• Индукционное предположение: Пусть для некоторых вершин v1,v2,...,vk, которые уже были удалены из Q, выполнено, что dist(vi) равны длинам кратчайших путей от s до vi для всех i = 1,2,...,k.

• Индукционный шаг: Рассмотрим следующую вершину u, которая удаляется из Q. Пусть P — кратчайший путь от s до u. Пусть v — предпоследняя вершина на пути P перед u. Тогда длина пути P равна dist(v)+w(v,u). По предположению индукции, dist(v) — длина кратчайшего пути от s до v.

Так как u выбирается из Q с минимальным значением dist(u), это означает, что для всех вершин z ∈ Q dist(z) ≥ dist(u). Поскольку путь через v был бы альтернативным путем к u через вершину, которая была удалена раньше u, dist(u) не могло бы быть меньше, чем dist(v)+ w(v,u).

Следовательно, dist(u) действительно является длиной кратчайшего пути от s до u.

Оценка сложности по времени: O(V*logV)

Перебираем V вершин и каждую добавляем в очередь через priority_queue.push() за logV.

Оценка сложности по памяти: O(V*M)

Код алгоритма:

```
#include <iostream>
#include <queue>
#include <vector>
#include <map>
#include <cmath>
#include <string>

using namespace std;

int main()
{

    int n, m, x_start, y_start, x_finish, y_finish;

    cin >> n >> m >> x_start >> y_start >> x_finish >> y_finish;

    vector<vector<float>> graph(n, vector<float>(m));
    vector<vector<float>> dist(n, vector<float>(m, INFINITY));
    vector<vector<float>> visited(n, vector<float>(m, false));
    vector<vector<pair<int, int>>> path(n, vector<pair<int, int>>(m, {-1, -
1}));

    map<char, float> field_types = {{'.', 1}, {'W', 2}, {'#', INFINITY}};

    string str;
    for (int i = 0; i < n; i++)
    {
        cin >> str;
        for (int j = 0; j < m; j++)
        {
            graph[i][j] = field_types.at(str[j]);
        }
    }

    priority_queue<pair<float, pair<int, int>>, vector<pair<float,
pair<int, int>>>, greater<pair<float, pair<int, int>>>> vertex_to_take;
    dist[x_start - 1][y_start - 1] = 0;

    vertex_to_take.push({dist[x_start - 1][y_start - 1], {x_start - 1,
y_start - 1}});

    while (!vertex_to_take.empty())
    {
        pair<int, int> vertex = vertex_to_take.top().second;
        vertex_to_take.pop();
        visited[vertex.first][vertex.second] = true;

        vector<pair<int, int>> connected;
        if (vertex.first > 0)
        {
            connected.push_back({vertex.first - 1, vertex.second});
        }
        if (vertex.first < n - 1)
```

```
        {
            connected.push_back({vertex.first + 1, vertex.second});
        }
        if (vertex.second > 0)
        {
            connected.push_back({vertex.first, vertex.second - 1});
        }
        if (vertex.second < m - 1)
        {
            connected.push_back({vertex.first, vertex.second + 1});
        }

        float new_dist;
        for (auto connected_vert : connected)
        {
            new_dist = dist[vertex.first][vertex.second] +
graph[connected_vert.first][connected_vert.second];
            if (!visited[connected_vert.first][connected_vert.second] &&
                dist[connected_vert.first][connected_vert.second] >
new_dist)
            {
                dist[connected_vert.first][connected_vert.second] =
new_dist;
                vertex_to_take.push({new_dist, {connected_vert.first,
connected_vert.second}});
                path[connected_vert.first][connected_vert.second] = vertex;
            }
        }
    }

    map<pair<int, int>, string> dirs = {{{1, 0}, "S"}, {{-1, 0}, "N"}, {{0,
1}, "E"}, {{0, -1}, "W"}};
    string str_path = "";
    auto vertex = make_pair(x_finish - 1, y_finish - 1);
    while (vertex != make_pair(x_start - 1, y_start - 1))
    {

        auto vertex_from = path[vertex.first][vertex.second];
        pair<int, int> dir = {vertex.first - vertex_from.first,
vertex.second - vertex_from.second};
        if (vertex_from == make_pair(-1, -1))
        {
            cout << "-1";
            return 0;
        }

        str_path = dirs[dir] + str_path;

        vertex = vertex_from;
    }
    cout << dist[x_finish - 1][y_finish - 1] << "\n";
    cout << str_path;
}
```

Задача №N «Свинки-копилки»

Пояснение к примененному алгоритму:

Копилки можно интерпретировать как вершины графа, а наличие ключа от i-ой копилки в k-ой копилке, ориентированным ребром k -> I; Мой алгоритм сводится к тому, чтобы найти количество таких компонент связанности, что в них есть такая вершина v, начав с которой можно добраться до любой другой вершины из этой компоненты связанности, и помимо того если добавить любую другую вершину в эту компоненту связанности, то такой вершины v уже не найдётся (то есть эта компонента связанности самая крупная). Дак вот количество таких компонент и будет являться ответом на вопрос, так как при разбитии определенной копилки из каждой компоненты, мы получим доступ ко всем копилкам.

Поиск таких компонент осуществляется следующим образом: из каждой вершины графа запускаю dfs (поиск в глубину) и составляется множество всех вершин, которые можно достичь начав с данной (разбив её). Далее если эта компонента не является подкомпонентой уже найденных, я удаляю все компоненты которые являются подкомпонентами данной, и добавляю её в множество искомых компонент. Таким образом мы гарантированно получим крупнейшие компоненты связанности этого графа перебором, и их количество будет ответом.

Оценка сложности по времени: O(V*E)

Перебор V вершин и запуск dfs для каждой за E+V.

Оценка сложности по памяти: O(V+E)

Код алгоритма:

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

using namespace std;

int main()
{
    int n;
    cin >> n;

    vector<vector<int>> graph(n + 1);
    for (int i = 1; i <= n; ++i)
    {
        int keyLocation;
        cin >> keyLocation;
        graph[keyLocation].push_back(i);
```

```
    }

    set<set<int>> components;

    for (int i = 1; i <= n; ++i)
    {
        vector<bool> visited(n + 1, false);
        set<int> new_component;
        vector<int> stack;
        stack.push_back(i);
        while (!stack.empty())
        {
            int vertex = stack.back();
            stack.pop_back();
            if (!visited[vertex])
            {
                new_component.insert(vertex);
                visited[vertex] = true;
                for (int connected_vertex : graph[vertex])
                {
                    if (!visited[connected_vertex])
                    {
                        stack.push_back(connected_vertex);
                    }
                }
            }
        }

        bool uniq = true;
        for (auto it = components.begin(); it != components.end();)
        {
            const set<int> &comp = *it;
            if (includes(comp.begin(), comp.end(), new_component.begin(),
new_component.end()))
            {
                uniq = false;
                break;
            }
            if (includes(new_component.begin(), new_component.end(),
comp.begin(), comp.end()))
            {
                it = components.erase(it);
                components.insert(new_component);
                uniq = false;
            }
            else
            {
                ++it;
            }
        }
        if (uniq)
        {
            components.insert(new_component);
        }
    }

    cout << components.size() << endl;
```

```
    return 0;
}
```

Задача №О «Долгой списывание!»

Пояснение к примененному алгоритму:

Учеников можно интерпретировать как вершины графа, а то, что v-ый учеников передавал записку u-ому ученику, как наличие ребра (v, u). Тогда задача разделить учеников на две группы так, чтобы любая передача записок осуществлялась между учениками разных групп, сводится к раскраске графа так, чтобы для каждой вершины все её соседи были другого цвета. Мой алгоритм заключается в том, чтобы попробовать разукрасить граф начиная с каждой вершины, обходя всю компоненту связанности через bfs. Если в одной из компонент при попытке покрасить вершину в один цвет, нашлась соседняя вершина с таким цветом, то разделить учеников на две группы так, чтобы обмен записками осуществлялся только между этими группами - невозможно.

Для доказательства корректности докажем:

1. Алгоритм корректно проверяет двудольность для одной компоненты связности.

Рассмотрим одну компоненту связности графа и начнем с произвольной вершины v.

• База индукции: Начальная вершина v красится в цвет -1.

• Индукционный шаг: Пусть вершина u покрашена в цвет c на iм шаге BFS. Тогда все её соседи, которые ещё не покрашены, будут покрашены в цвет -c на следующем шаге BFS. Если на каком-либо шаге BFS обнаруживается, что сосед имеет тот же цвет, что и текущая вершина, это означает, что граф не двудольный, так как две смежные вершины имеют одинаковый цвет. Если обход BFS завершился без обнаружения конфликта, это значит, что все смежные вершины имеют противоположные цвета.

Следовательно, алгоритм корректно проверяет двудольность для одной компоненты связности.

Оценка сложности по времени: O(V*E)

Перебор V вершин и запуск bfs из каждой за V+E.

Оценка сложности по памяти: O(V)

Код алгоритма:

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

using namespace std;

int main()
{

    int n, m;
    cin >> n >> m;

    map<int, vector<int>> graph;

    int first, second;
    for (int i = 0; i < m; i++)
    {
        cin >> first >> second;
        if (first == second)
        {
            cout << "NO";
            return 0;
        }
        if (graph.find(first) != graph.end())
        {
            graph[first].push_back(second);
        }
        else
        {
            graph[first] = {second};
        }
        if (graph.find(second) != graph.end())
        {
            graph[second].push_back(first);
        }
        else
        {
            graph[second] = {first};
        }
    }

    for (int i = 1; i < n + 1; i++)
    {
        vector<int> color(n + 1);
        queue<pair<int, int>> queue;
        vector<int> visited(n + 1);
        queue.push({i, -1});

        while (!queue.empty())
        {
            int vertex = queue.front().first;
            int current_id = -queue.front().second;
            queue.pop();
            visited[vertex] = true;
```

```
            for (int conn_vert : graph[vertex])
            {
                if (color[conn_vert] == current_id)
                {
                    cout << "NO";
                    return 0;
                }
            }

            if (color[vertex] == 0)
            {
                color[vertex] = current_id;
            }
            for (int conn_vert : graph[vertex])
            {
                if (!visited[conn_vert])
                {
                    queue.push({conn_vert, current_id});
                }
            }
        }
    }

    cout << "YES";
}
```

Задача №P «Авиаперелёты»

Пояснение к примененному алгоритму:

Мое решение заключается в использовании модифицированного алгоритма Флайда-Уоршелла для нахождения минимально возможного бака. Создаётся двумерный массив graph[][] в котором элемент graph[i][j] означате минимально возможный бак для перемещения из вершины i в вершину j без возможности лететь через другие вершины. Затем я использую двумерный массив dp_max_fuel_K на каждой итерации k от 0 до n высчитывая минимально возможный бак для полета из вершины i в вершину j при этом с возможностью пролетать через города 0, …, k. Для расчета минимально возможного бака на K-ой итерации dp_max_fuel_K[i][j] нужно выбирать min(dp_max_fuel_(K-1)[i][j], max(dp_max_fuel[i][k], dp_max_fuel[k][j])), то есть минимум между минимальным нужным размером бака при пролете от i с возможностью через вершины 0,…,k до j и между большим значением минимального бака который потребуется при пролете от i до k (вполне возможно с пролетом через 0,…,k-1) и минимальным значением бака который потребуется для пролета из k до j (вполне возможно с пролетом через 0,…,k-1).

Доказательство корректности

1. Для любой пары городов i и j, значение dp_max_fuel[i][j] на каждом шаге алгоритма представляет собой минимально возможный максимальный расход топлива для перелета между этими городами с учетом всех промежуточных городов.

Докажем это утверждение по индукции.

• База индукции: Рассмотрим начальный этап алгоритма, когда промежуточные вершины отсутствуют (т.е. k = 0). В этом случае матрица dp_max_fuel равна исходной матрице graph, где dp_max_fuel[i][j] — это расход топлива на прямой перелет из города i в город j. Очевидно, что в этом случае значения dp_max_fuel[i][j] корректны и представляют собой прямые пути без промежуточных дозаправок.

• Индукционное предположение: Пусть для некоторого k матрица dp_max_fuel содержит минимально возможные максимальные значения топлива для перелета между всеми парами городов i и j, с учетом промежуточных городов из множества {0,1,...,k − 1}.

• Индукционный шаг: Докажем, что при добавлении новой промежуточной вершины k матрица dp_max_fuel продолжает содержать минимально возможные максимальные значения топлива для перелета между всеми парами городов i и j. Для каждой пары городов i и j обновляется значение dp_max_fuel[i][j] по следующей формуле:

dp_max_fuel[i][j] = min(dp_max_fuel[i][j],max(dp_max_fuel[i][k],dp_max_fuel[k][j]))

Это означает, что мы сравниваем текущее значение dp_max_fuel[i][j] с максимальным расходом топлива на пути, который проходит через вершину k. Если новый путь через вершину k уменьшает максимальный расход топлива для пары городов i и j, то значение dp_max_fuel[i][j] обновляется.

Таким образом, на каждом шаге мы гарантируем, что для всех пар i и j значение dp_max_fuel[i][j] остается минимально возможным максимальным расходом топлива с учетом всех промежуточных городов до k.

Оценка сложности по времени: O(V^3)

Тройной чикл, в котором для каждого возможного города в маршруте k=0…V, просчитывался минимальный нужный бак для путешествия из i=0…V города в j=0…V.

Оценка сложности по памяти: O(V^2)

Матрица dp_max_fuel динамически высчитываемого минимального бака.

Код алгоритма:

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

using namespace std;

int main() {
    int n;
    cin >> n;

    vector<vector<int>> graph(n, vector<int>(n));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> graph[i][j];
        }
    }


    vector<vector<int>> dp_max_fuel = vector(graph);

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i == j || i == k || j == k) continue;

                dp_max_fuel[i][j] = min(dp_max_fuel[i][j],
max(dp_max_fuel[i][k], dp_max_fuel[k][j]));

            }
        }
    }
    int m = 0;

    for (int i = 0; i < n; i++) {
        m = max(m, *max_element(dp_max_fuel[i].begin(),
dp_max_fuel[i].end()));
    }

    cout << m;
}
```

Задача №1160 «Network»

Пояснение к примененному алгоритму:

Мой алгоритм решает задачу нахождения минимального максимального кабеля в сети хабов с помощью алгоритма Прима, который используется для построения минимального остовного дерева. Начальная вершина выбирается произвольно (хаб 1), и все рёбра, соединяющие её с другими вершинами, добавляются в приоритетную очередь (множество pool_of_borders). Для каждой новой вершины, добавляемой в минимальное остовное дерево, обновляются рёбра в очереди, исключая те, которые соединяют уже подключённые вершины. При добавлении нового ребра обновляется максимальная длина кабеля, если она больше текущего максимума.

Алгоритм корректно минимизирует максимальную длину кабеля. (по индукции):

• База индукции: Рассмотрим начальный шаг алгоритма, когда выбирается первый хаб и все его соединения добавляются в приоритетную очередь. Очевидно, что в этот момент максимальная длина кабеля минимальна, так как мы рассматриваем только одно ребро.

• Индукционное предположение: Пусть на k-м шаге алгоритма, когда уже добавлены k хабов, максимальная длина кабеля минимальна среди всех возможных остовных деревьев, содержащих эти k хабов.

• Индукционный шаг: Рассмотрим следующий шаг, на котором добавляется (k+1)-й хаб. Алгоритм выбирает ребро с минимальной длиной, соединяющее новый хаб с уже включёнными в дерево вершинами. Таким образом, на каждом шаге минимизируется максимальная длина добавленного ребра.

Пусть Tk — минимальное остовное дерево, содержащее k хабов, построенное алгоритмом. На шаге k +1 добавляется ребро с минимальной длиной ℓ_{min} , соединяющее новый хаб с уже включёнными в дерево вершинами.

Максимальная длина ребра в Tk+1 будет $\max(\ell_{max}^{(k)}, \ell_{min})$, где $\ell_{max}^{(k)}$, — максимальная длина ребра в Tk. Так как алгоритм всегда выбирает минимальное возможное ребро, максимальная длина кабеля в Tk+1 будет минимальной среди всех возможных остовных деревьев, содержащих k +1 хабов.

Оценка сложности по времени: O(E*logE)

Обработки E ребер и вставка/удаление ребер в set за logE.

Оценка сложности по памяти: O(E+V)

Код алгоритма:

```
#include <iostream>
#include <vector>
#include <map>
#include <set>

using namespace std;

int main() {

    int n, m;
    cin >> n >> m;

    map<int, vector<pair<int, int>>> hubs; // hub, hub_to, length_to

    for (int i = 1; i < n+1; i++) {
        hubs[i] = {};
    }

    int hub, hub_to, length;
    for (int i = 0; i < m; i++) {

        cin >> hub >> hub_to >> length;
        hubs[hub].push_back({hub_to, length});
        hubs[hub_to].push_back({hub, length});
    }

    vector<bool> is_connected(n+1, false);

    set<pair<int, pair<int, int>>> pool_of_borders; // length_to, hub,
hub_to
    is_connected[1] = true;
    for (auto border : hubs[1]) {
        pool_of_borders.insert({border.second, {1, border.first}});
    }

    int max_length = 0;
    vector<pair<int, int>> network;
    for (int i = 1; i < n; i++) {

        for (auto it = pool_of_borders.begin(); it !=
pool_of_borders.end(); ) {
            const auto hub_to = (*it).second.second;
            if (!is_connected[hub_to]) {
                is_connected[hub_to] = true;
                max_length = max(max_length, (*it).first);
                network.push_back({(*it).second.first,
(*it).second.second});
                for (auto border : hubs[hub_to]) {
                    if (!is_connected[border.first]) {
                        pool_of_borders.insert({border.second, {hub_to,
border.first}});
                    }
                }
```

```
            }
            it = pool_of_borders.erase(it);
        }

    }

    cout << max_length << "\n" << network.size() << "\n";
    for (auto el : network) {
        cout << el.first << " " << el.second << "\n";
    }

}
```

Задача №1329 «Галактическая история»

Пояснение к примененному алгоритму:

Этот алгоритм решает задачу определения, находится ли веха A в поддереве с корнем в вехе B или наоборот, для дерева исторических вех. Алгоритм основан на использовании обхода дерева в глубину (dfs) и временных меток, которые помогают определить, находится ли одна вершина в поддереве другой.

Доказательство корректности алгоритма:
1. Правильность меток времени.

Временные метки time_in и time_out, присвоенные вершинам во время обхода дерева, корректно отражают вход и выход из каждой вершины в рамках поддерева.

• При входе в вершину u устанавливается метка time_in[u].

• Алгоритм затем рекурсивно обходит все дочерние вершины u.

• После завершения обработки всех дочерних вершин устанавливается метка time_out[u].

• Таким образом, для любой вершины v, находящейся в поддереве вершины u, выполнено time_in[u]<time_in[v] и time_out[u]>time_out[v].

2. Определение поддерева с помощью временных меток.

Вершина A является корнем поддерева, содержащего вершину B, если и только если time_in[A]≤time_in[B] и time_out[A]≥time_out[B].

• Если вершина B находится в поддереве вершины A, то time_in[A] будет меньше или равно time_in[B], и time_out[A] будет больше или равно time_out[B].

• Если выполнены условия time_in[A]≤time_in[B] и time_out[A]≥ time_out[B], это означает, что вершина B была посещена после входа в вершину A и обработана до выхода из вершины A. Следовательно, вершина B находится в поддереве вершины A.

• Аналогично, если выполнены условия time_in[B] ≤ time_in[A] и time_out[B]≥time_out[A], вершина A находится в поддереве вершины B.

Оценка сложности по времени: O(V+L)

Выполнение dfs за N и обработка каждого из L запроса за 1.

Оценка сложности по памяти: O(V+L)

Дерево с V вершинами и V-1 ребрами а так же L запросов.

Код алгоритма:

```
#include <iostream>
#include <vector>
#include <map>
#include <set>

using namespace std;

int main()
{
    int n;
    cin >> n;

    map<int, vector<int>> tree;
    map<int, bool> visited;
    int root = -1;

    int id, parent_id;
    for (int i = 0; i < n; i++)
    {
        cin >> id >> parent_id;
        visited[id] = false;
        if (parent_id == -1)
        {
            root = id;
        }
        else
        {
            tree[parent_id].push_back(id);
        }
    }

    map<int, int> time_in;
    map<int, int> time_out;
    vector<pair<int, bool>> stack;

    stack.push_back({root, true});
    int time = 0;

    while (!stack.empty())
    {
        int node = stack.back().first;
        bool entering = stack.back().second;
```

```
            stack.pop_back();

            time++;
            if (entering)
            {
                time_in[node] = time;
                stack.push_back({node, false});
                visited[node] = true;
                for (int child : tree[node])
                {
                    if (!visited[child])
                    {
                        stack.push_back({child, true});
                    }
                }
            }
            else
            {
                time_out[node] = time;
            }
        }

        int l;
        cin >> l;
        int a, b;
        string result = "";

        for (int i = 0; i < l; i++)
        {
            cin >> a >> b;
            if (time_in[a] <= time_in[b] && time_out[a] >= time_out[b])
            {
                result += "1\n";
            }
            else if (time_in[b] <= time_in[a] && time_out[b] >= time_out[a])
            {
                result += "2\n";
            }
            else
            {
                result += "0\n";
            }
        }

        cout << result;

        return 0;
    }
```