Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Яндекс.Контест

Выполнил:

Студент группы P3211

Кривоносов Е.Д.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2021

Задача №M «Цивилизация»

Пояснение к примененному алгоритму:

Чтобы решить данную задачу я воспользовался алгоритмом Дейкстры. В процессе решения задачи мне нужно найти кратчайшие пути от одной вершины графа до всех остальных. Начинал решение с конца (от точки в которую нужно прийти) и шёл в самое начало. Вокруг карты сделал “ограждение”, чтобы в последствии не пришлось придумывать сложную логику для обработки на краях карты координат. Варианты, куда могут пойти только 4, поэтому нужно было ещё избегать стояния на месте и диагональных переходов.

Сложность алгоритма:
O(n*m)

Код:

```
#include <iostream>
#include <string.h>
#include <vector>
#include <set>

using namespace std;

string create_string(int size){
    string line = "";
    while (size != 0){
        line += "X";
        size--;
    }
    return line;
}

void solve(){
    int n, m, x0, y0, x, y;
    cin >> n >> m >> x0 >> y0 >> x >> y;

    vector <string> karta;
    string line;
    string answer = "";

    // Считываем карту и ставим ограждение
    karta.push_back(create_string(m + 2));
    while (cin >> line){
        karta.push_back('X' + line + 'X');
    }
    karta.push_back(create_string(m + 2));

    vector <vector <int>> timer (n + 2, vector <int>(m + 2));
    vector <vector <char>> way (n + 2, vector <char>(m + 2));
    set <pair <int, pair <int, int>>> position;

    // Заполняем массив времени карты
    for (int i = 0; i < n + 2; i ++){
        for (int j = 0; j < m + 2; j++){
            if (karta[i][j] == '#' || karta[i][j] == 'X'){
                timer[i][j] = -1;
```

```
                continue;
            }
            timer[i][j] = 1006010;
        }
    }

    // Устанавливаем начальные позиции и обнулаяем таймер
    timer[x][y] = 0;
    position.insert({0, {x, y}});

    // Идем в обратную сторону от конца к началу (так эффективнее проходить
лабиринт)
    while (!position.empty()){
        pair <int, int> pos = position.begin() -> second;
        position.erase(position.begin());

        for (int i = -1; i <= 1; i++){
            for (int j = -1; j <= 1; j++){
                // Проверяем, что мы не идём по диагонали и не стоим, а также
нет ли на следующем шаге воды или ограждения
                if ((i != 0 && j != 0) || (i == 0 && j == 0) || karta[pos.first
+ i][pos.second + j] == '#' || karta[pos.first + i][pos.second + j] == 'X'){
                    continue;
                }
                // Подсчитываем сколько в зависимости куда мы идём, сколько
вреени мы затрачиваем
                if (timer[pos.first + i][pos.second + j] >
timer[pos.first][pos.second] + (karta[pos.first][pos.second] == 'W') + 1){
                    timer[pos.first + i][pos.second + j] =
timer[pos.first][pos.second] + (karta[pos.first][pos.second] == 'W') + 1;
                    position.insert({timer[pos.first + i][pos.second + j],
{pos.first + i, pos.second + j}});

                    if (i == -1){
                        way[pos.first + i][pos.second + j] = 'S';
                    } else if (j == 1){
                        way[pos.first + i][pos.second + j] = 'W';
                    } else if (i == 1){
                        way[pos.first + i][pos.second + j] = 'N';
                    } else if (j == -1){
                        way[pos.first + i][pos.second + j] = 'E';
                    }
                }
            }
        }
    }

    if (timer[x0][y0] == 1006010){
        cout << -1 << endl;
    } else {
        cout << timer[x0][y0] << endl;
        while (!(x0 == x) || !(y0 == y)){
                answer += way[x0][y0];
            if (way[x0][y0] == 'S'){
                x0++;
            } else if (way[x0][y0] == 'W'){
                y0--;
            } else if (way[x0][y0] == 'N'){
                x0--;
            } else if (way[x0][y0] == 'E'){
                y0++;
            }
```

```
        }
        cout << answer << endl;
    }
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    solve();
    return 0;
}
```

Задача №N «Свинки-копилки»

Пояснение к примененному алгоритму:

Для решения данной задачи я воспользовался обходом в глубину или кратко dfs. Ведь он прост в реализации и эффективен, чтобы проверить весь граф от начала и до конца. Вершины – это копилки, а ребра – это ключи. Чтобы получить ключ от всех свинок нам достаточно сломать одну копилку. Если встречается копилка, в которой не хранится ни один ключ, тогда ей будет соответствовать вершина, в которую не входит ни одно ребро и из которой существует путь по ребрам, ведущий к какому-либо циклу. Разбивать их нет смысла, ведь мы можем их открыть, получив доступ из цикла.
А ответом будет служить это кол-во циклов в графе, где у каждой вершины существует только 1 ребро.

Сложность алгоритма:
O(n+m)

Код:

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

using namespace std;

void dfs (int i, int i_cur, vector<vector<int>> &piggy_bank, vector<bool> &used,
vector<bool> &broke){
    if (used[i]){
        return;
    }
    used[i] = true;
    for (int u : piggy_bank[i]){
        if (u != i_cur){
            dfs(u, i_cur, piggy_bank, used, broke);
            broke[i] = false;
        }
    }
}
```

```
void solve(){
    int n, key, i = 0;

    cin >> n;

    vector<vector<int>> piggy_bank(n);
    vector<bool> used(n, false);
    vector<bool> broke(n, true);

    while (cin >> key){
        piggy_bank[--key].push_back(i);
        i++;
    }

    for (int i = 0; i < n; i++){
        if (!used[i]){
            dfs(i, i, piggy_bank, used, broke);
        }
    }

    cout << count(broke.begin(), broke.end(), true);
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    solve();
    return 0;
}
```

Задача №O «Долой списывание!»

Пояснение к примененному алгоритму:

Аналогично, как и в прошлой задаче я использовал dfs (обход графа в глубину). Заполнив граф парами, когда один или другой даёт списывать. И проверял их обмены, возможно ли разделить их на 2 группы.

Сложность алгоритма:
O(n+m)

Код:

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

using namespace std;

bool dfs (int i, vector <vector <int>> &para, int *trade, bool start = true){
    trade[i] = !start;
    bool answer;
    for (auto p : para[i]){
        if (trade[p] == -1){
            answer = dfs(p, para, trade, !start);
        } else if (trade[i] == trade[p]){
            return false;
```

```
        }
    }
    return answer;
}

void solve(){
    int n, m;
    cin >> n >> m;
    bool answer;
    vector <vector <int>> para (n);
    int trade[n];
    memset(trade, -1, sizeof(trade));

    while (m != 0){
        int p1, p2;
        cin >> p1 >> p2;
        para[--p1].push_back(--p2);
        para[p2].push_back(p1);
        m--;
    }

    int i = 0;
    while (n != 0){
        if (trade[i] == -1){
            answer = dfs(i, para, trade);
        }
        i++;
        n--;
    }

    if (answer){
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    solve();
    return 0;
}
```

Задача №P «Авиаперелёты»

Пояснение к примененному алгоритму:

И снова наш любимый алгоритм dfs (обход в глубину). Чтобы летать из одного города в другой, наш граф должен быть сильно связанный, а значит выполнять условия:

1) Из вершины vertex существует путь во все остальные
2) Из любой вершины существует путь в vertex

Если эти условия выполняются для хотя бы 1 вершины, то для всех остальных они тоже выполняются. Значит с помощью обхода в глубину мы определяем сильно

связанность графа. А исходный минимальный объём бака самолеты мы можем найти бинарным поиском. Как делали это в задаче про Коров и стойла.

Сложность алгоритма:
O((n+m)*log(1e9))

Код:

```
#include <iostream>
#include <string.h>
#include <vector>
#include <set>

using namespace std;

void dfs(int vertex, int n, bool direction, vector<bool> &visited,
vector<vector<bool>> &g_check){
    visited[vertex] = true;
    for (int i = 0; i < n; i++){
        if (direction){
            if (g_check[i][vertex] && !visited[i]) {
                dfs(i, n, direction, visited, g_check);
            }
        } else {
            if (g_check[vertex][i] && !visited[i]){
                dfs(i, n, direction, visited, g_check);
            }
        }
    }
}

bool check_connectivity(int n, vector<bool> &visited){
    for (int i = 0; i < n; i++){
        if (visited[i]){
            continue;
        } else {
            return false;
        }
    }
    return true;
}

void solve(){
    int n;
    cin >> n;
    vector<vector<int>> graph(n, vector<int> (n));
    vector<vector<bool>> g_check(n, vector<bool>(n));
    vector<bool> visited;
    int i = 0, j = 0, oil;

    while (cin >> oil){
        graph[i][j] = oil;
        j++;
        if (j == n){
            i++;
            j = 0;
        }
    }

    int l = 0, r = 1000000000;
```

```
    while (l != r){
        int mid = (l + r) / 2;
        visited = vector<bool>(n, false);
        for (int i = 0; i < n; i++){
            for (int j = 0; j < n; j++){
                g_check[i][j] = graph[i][j] <= mid;
            }
        }

        dfs(0, n, 0, visited, g_check);

        bool connectivity = false;
        if (check_connectivity(n, visited)){
            visited = vector<bool>(n, false);

            dfs(0, n, 1, visited, g_check);

            if (!check_connectivity(n, visited)){
                connectivity = true;
            }
        } else connectivity = true;


        if (connectivity){
            l = mid + 1;
        } else {
            r = mid;
        }
    }

    cout << l << endl;
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    solve();
    return 0;
}
```