Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3233
Богатов Александр Сергеевич

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2022

Задача M: Цивилизация

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int n, m, x0, y0, xe, ye;
int mark = 0;
int maximum_dist = 2147483647;

class Block {
    public:
        int id;

        int x;
        int y;
        int weight;
        int mark;
        bool visited;

        Block (int id, int x, int y, int weight, bool visited) {
              this->id = id;
              this->x = x;
              this->y = y;
              this->weight = weight;
              this->visited = visited;
              this->mark = maximum_dist;
           }
};


class compare
{
    public:
        bool operator()(Block* a,Block* b) {
            return a->mark < b->mark;
        }
};

vector<vector<Block*>> field(1000, vector<Block*>(1000, new Block(-1,
-1, -1, -1, true)));

void path_by_dijkstra(int x, int y) {
    priority_queue<Block*, vector<Block*>, compare> ord;
    field[x][y]->mark = 0;
    ord.push(field[x][y]);
    while(!ord.empty()) {
        x = ord.top()->x;
        y = ord.top()->y;
        ord.pop();
        if (field[x][y]->visited)
            continue;
        field[x][y]->visited = 1;

        if (x > 0 && !field[x-1][y]->visited && field[x-1][y]->weight
!= -1) {
```

```
            if (field[x][y]->mark + field[x-1][y]->weight < field[x-
1][y]->mark)
                field[x-1][y]->mark = abs(field[x][y]->mark) +
field[x-1][y]->weight;
            if (field[x-1][y]->mark > 0)
                field[x-1][y]->mark = -field[x-1][y]->mark;
            ord.push(field[x-1][y]);
        }

        if (x < n-1 && !field[x+1][y]->visited && field[x+1][y]-
>weight != -1) {
            if (field[x][y]->mark + field[x+1][y]->weight <
field[x+1][y]->mark)
                field[x+1][y]->mark = abs(field[x][y]->mark) +
field[x+1][y]->weight;
            if (field[x+1][y]->mark > 0)
                field[x+1][y]->mark = -field[x+1][y]->mark;
            ord.push(field[x+1][y]);
        }

        if (y > 0 && !field[x][y-1]->visited && field[x][y-1]->weight
!= -1) {
            if (field[x][y]->mark + field[x][y-1]->weight <
field[x][y-1]->mark)
                field[x][y-1]->mark = abs(field[x][y]->mark) +
field[x][y-1]->weight;
            if (field[x][y-1]->mark > 0)
                field[x][y-1]->mark = -field[x][y-1]->mark;
            ord.push(field[x][y-1]);
        }

        if (y < m-1 && !field[x][y+1]->visited && field[x][y+1]-
>weight != -1) {
            if (field[x][y]->mark + field[x][y+1]->weight <
field[x][y+1]->mark)
                field[x][y+1]->mark = abs(field[x][y]->mark) +
field[x][y+1]->weight;
            if (field[x][y+1]->mark > 0)
                field[x][y+1]->mark = -field[x][y+1]->mark;
            ord.push(field[x][y+1]);
        }
    }
}

void trace_by_dijkstra(int x, int y)
{
    if (!field[x][y]->mark)
        return;
    mark = field[x][y]->mark;
    char direction;
    if (x > 0 && mark > field[x-1][y]->mark) {
        direction = 'S';
        mark = field[x-1][y]->mark;
    }
    if (x < n-1 && mark > field[x+1][y]->mark) {
        direction = 'N';
        mark = field[x+1][y]->mark;
    }
```

```
    if (y > 0 && mark > field[x][y-1]->mark) {
        direction = 'E';
        mark = field[x][y-1]->mark;
    }
    if (y < m-1 && mark > field[x][y+1]->mark) {
        direction = 'W';
        mark = field[x][y+1]->mark;
    }
    if (direction == 'S') {
        trace_by_dijkstra(x - 1, y);
    } else if (direction == 'N') {
        trace_by_dijkstra(x + 1, y);
    } else if (direction == 'E') {
        trace_by_dijkstra(x, y - 1);
    } else if (direction == 'W') {
        trace_by_dijkstra(x, y + 1);
    }
    cout << direction;
}

int get_weight(char terrain) {
    switch (terrain) {
        case '.':
            return 1;
        case 'W':
            return 2;
        default:
            return -1;
    }
}

int main() {

    cin >> n >> m >> x0 >> y0 >> xe >> ye;

    int counter = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            char terrain;
            cin >> terrain;
            int weight = get_weight(terrain);
            field[i][j] = new Block(counter, i, j, weight, false);
            counter++;
        }
    }
    x0--;
    y0--;
    xe--;
    ye--;

    path_by_dijkstra(x0, y0);

    for (int i = 0 ; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (field[i][j]->mark < 0)
                field[i][j]->mark = -field[i][j]->mark;
        }
```

```
    }

    if (field[xe][ye]->mark == maximum_dist) {
        cout << -1;
    } else {
        cout << field[xe][ye]->mark << endl;
        trace_by_dijkstra(xe, ye);
    }

    cout << endl;
    return 0;
}
```

Пояснение к примененному алгоритму:

Используя алгоритм Дейкстры мы формируем кратчайшие пути от изначального пункта до остальных точек, а затем проходим по вычисленному пути от конечного пункта до начального и выводим путь. Так как для выбора оптимальной вершины используется приоритетная очередь (на основе бинарной пирамиды), сложность алгоритма - O(E log V).

Задача N: Свинки-Копилки

```
#include <iostream>
#include <vector>
using namespace std;

class Node {
    public:
        string color;
        Node() {
            this->color = "WHITE";
        }
};


void DFS_visit(vector<vector<int>>& graph, int s, vector<Node>&
visit_map, int& t) {
    visit_map[s].color = "GRAY";
    for (int i = 0; i < graph[s].size(); ++i) {
        int v = graph[s][i];
        if (visit_map[v].color == "WHITE") {
            DFS_visit(graph, v, visit_map, t);
        } else if (visit_map[v].color == "GRAY") t++;
    }
    visit_map[s].color = "BLACK";
}

void DFS(vector<vector<int>>& graph, vector<Node>& visit_map, int&
counter) {
    for (int i = 0; i < graph.size(); ++i) {
        if (visit_map[i].color == "WHITE")
            DFS_visit(graph, i, visit_map, counter);
    }
}
```

```
int main() {

    int n;
    int result = 0;

    cin >> n;

    vector<vector<int>> pig_graph;
    pig_graph.resize(n);
    vector<Node> vis(n, *(new Node()));

    for (int i = 0; i < n; i++) {
        int key;
        cin >> key;
        pig_graph[i].push_back(--key);
    }

    DFS(pig_graph, vis, result);

    cout << result << endl;
    return 0;
}
```

Пояснение к примененному алгоритму:

Наименьшее число разбитых копилок определяется числом циклов в графе, где ребро от вершины к другой вершине равносильно тому, что копилка 1 содержит ключ к копилке 2. Число циклов можем вычислить алгоритмом поиска в глубину, увеличивая счетчик циклов каждый раз когда встречаемся при обходе с серой вершиной. Сложность поиска в глубину по графу - O(V + E).

Задача O: Долой списывание!

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

using namespace std;

class Node {
    public:
        int color;
        int idx;
        Node() {
            this->color = 0;
        }
};


bool BFS_bipartite(vector<vector<int>> graph, int s) {
    vector<Node> visited(graph.size(), *(new Node()));
    visited[s].idx = 1;
    if (visited[s].color == 0)
        visited[s].color = 1;
    queue<int> ord;
    ord.push(s);
    while(!ord.empty()) {
        int current = ord.front();
        ord.pop();
```

```
            for (int i = 0; i < graph[current].size(); i++) {
                int v = graph[current][i];
                if (visited[v].color == 0) {
                    visited[v].color = -visited[current].color;
                    ord.push(v);
                }
                if (visited[v].color == visited[current].color)
                    return false;
            }
        }
        return true;
}

int main()
{
    int n, m;

    cin >> n >> m;

    vector<vector<int>> cheat_graph;
    cheat_graph.resize(n);
    vector<Node> vis(n, *(new Node()));

    for (int i = 0; i < m; i++) {
        int lk1;
        int lk2;
        cin >> lk1 >> lk2;
        --lk1;
        --lk2;
        cheat_graph[lk1].push_back(lk2);
              cheat_graph[lk2].push_back(lk1);
    }

    for (int i = 0; i < n; i++) {
        if (!BFS_bipartite(cheat_graph, i)) {
            cout << "NO" << endl;
            return 0;
        }
    }

    cout << "YES" << endl;

    return 0;
}
```

Пояснение к примененному алгоритму:

По условию задачи, искомый нами граф должен быть двудольным – поэтому будем проходиться по графу, начиная с разных вершин, методом поиска в ширину, раскрашивая вершины в два разных цвета. Если мы встретимся при обходе со связанными вершинами одного цвета – граф не двудольный, а значит ответ на задачу – нет. Сложность поиска в ширину по графу - O(V + E).

Задача P: Авиаперелеты

```
#include <iostream>
#include <vector>
using namespace std;
```

```
#define LIMIT 1000000000;

struct Node {
    int color;
};


void DFS_visit(bool direction, vector<vector<int>>& graph, int s,
vector<Node>& visit_map, int& t, int& length) {
    visit_map[s].color = 1;
    t++;
    for (int i = 0; i < graph[s].size(); ++i) {
        int v;
        if (!direction)
            v = graph[s][i];
        else v = graph[i][s];
        if (visit_map[i].color == 0 && v <= length) {
            DFS_visit(direction, graph, i, visit_map, t, length);
        }
    }
}

bool DFS_is_connected_check(vector<vector<int>>& graph, int& n, int
length) {
    int counter = 0;
    int counter_backwards = 0;
    vector<Node> vmap(n, {0});
    DFS_visit(false, graph, 0, vmap, counter, length);
    vector<Node> vmap_backwards(n, {0});
    if (counter == n) {
        DFS_visit(true, graph, 0, vmap_backwards, counter_backwards,
length);
        return (counter == counter_backwards);
    }
    return false;
}

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];
        }
    }

    int left = 0;
    int right = LIMIT;
    while (right > left) {
        int middle = (left + right) / 2;
        if (DFS_is_connected_check(graph, n, middle))
            right = middle;
```

```
            else left = middle + 1;
        }
        cout << right << endl;
        return 0;
}
```

Пояснение к примененному алгоритму:

Здесь необходимо построить полносвязный ориентированный граф с минимальными путями из вершины в другие вершины. Граф полносвязный, если в нем от каждой вершины V можно добраться до остальных вершин, и если в графе с развернутыми в другую сторону ребрами из всех вершин графа есть путь в вершину V. Будем проверять эти условия для различных значения размера бака и найдем подходящий размер методом бинарного поиска.