Университет ИТМО

Факультет программной инженерии и компьютерной техники

# Лабораторная работа №4

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3232

Чмурова М. В.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

**Задача №M «Цивилизация»**

Пояснение к примененному алгоритму:

Для решения задачи «Цивилизация» используется подход, при котором `array_of_visited` представляется схемой, которая по окончании работы алгоритма будет хранить наименьший путь до каждой точки, пока не будет достигнута искомая точка. Для того, чтобы заполнить этот массив используется приоритетная очередь для обхода всех точек, в которую добавляются точки, в которые возможно пойти из текущей рассматриваемой. Для этого рассматриваются все направления вокруг этой точки (вверх, вниз, вправо и влево)

Для того, чтобы получить путь, который мы прошли до искомой точки необходимо рекурсивно пройтись по нашей схеме путей в обратную сторону, пока мы не вернемся из конечной точки в начальную.

Сложность алгоритма:

O(n * m * log(n * m)), где n – количество строк, m – количество столбцов

Код алгоритма:

```
#include <iostream>
#include <tuple>
#include <vector>
#include <string>
#include <queue>
#include <set>
#include <list>

using namespace std;

struct Element {
    int intValue;
    int idValue;
};
bool operator< (const Element &v1, const Element &v2){
    return v1.intValue > v2.intValue;
}

int main() {

    int n, m, x, y, x_dest, y_dest;
    cin >> n >> m >> x >> y >> x_dest >> y_dest;

    string map_of_game[n];
```

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

    int start = (x - 1) * m + (y - 1);
    int end = (x_dest - 1) * m + (y_dest - 1);

    // хранение информации о том, какой элемент обходить следующим
    priority_queue<Element> order_of_traversing;
    order_of_traversing.push({0, start});

    // хранение значения оптимальных путей
    Element array_of_visited[n*m];
    for (int i = 0; i < n*m; i++) {
        array_of_visited[i] = {-1, -1};
    }

    while(!order_of_traversing.empty()) {

        auto current_element = order_of_traversing.top();
        order_of_traversing.pop();

        if (current_element.idValue == end) {
            break;
        }

        // для хранения точек, которые получили
        multiset<Element> found_elements;
        int index_current = current_element.idValue;

        int x_current = current_element.idValue / m;
        int y_current = current_element.idValue % m;

        if (map_of_game[x_current][y_current] == '#') {
            continue;
        }

        // вверх
        if (x_current - 1 >= 0 && map_of_game[x_current-1][y_current] !=
'#') {
            if (map_of_game[x_current-1][y_current] == 'W') {
                found_elements.insert({2, index_current - m});
            } else {
                found_elements.insert({1, index_current - m});
            }
        }

        // вниз
        if (x_current + 1 < n && map_of_game[x_current + 1][y_current] !=
'#') {
            if (map_of_game[x_current+1][y_current] == 'W') {
                found_elements.insert({2, index_current + m});
            } else {
```

```
                found_elements.insert({1, index_current + m});
            }
        }

        // влево
        if (y_current - 1 > 0 && map_of_game[x_current][y_current - 1] !=
'#') {
            if (map_of_game[x_current][y_current-1] == 'W') {
                found_elements.insert({2, index_current - 1});
            } else {
                found_elements.insert({1, index_current - 1});
            }
        }

        // вправо
        if (y_current + 1 < m && map_of_game[x_current][y_current + 1] !=
'#') {
            if (map_of_game[x_current][y_current+1] == 'W') {
                found_elements.insert({2, index_current + 1});
            } else {
                found_elements.insert({1, index_current + 1});
            }
        }

        for (auto element : found_elements) {
            int length_of_way = current_element.intValue +
element.intValue;
            if (array_of_visited[element.idValue].intValue == -1 ||
array_of_visited[element.idValue].intValue > length_of_way) {
                order_of_traversing.push({length_of_way,
element.idValue});
                array_of_visited[element.idValue].intValue =
length_of_way;
                array_of_visited[element.idValue].idValue =
current_element.idValue;
            }
        }
    }

    cout << array_of_visited[end].intValue << endl;
    if (array_of_visited[end].intValue == -1) return 0;

    list<string> letters;
    while (end != start) {
        int differences = end - array_of_visited[end].idValue;
        if (differences == -m) {
            letters.push_front("N");
        } else if (differences == m) {
            letters.push_front("S");
        } else if (differences == -1) {
            letters.push_front("W");
        } else if (differences == 1) {
            letters.push_front("E");
```

```
        }
        end = array_of_visited[end].idValue;
    }

    for (string& letter : letters) {
        cout << letter;
    }
    cout << endl;

    return 0;
}
```

**Задача №N «Свинки-копилки»**

Пояснение к примененному алгоритму:

Для решения этой задачи используется алгоритм поиска в глубину, при котором мы проходимся по каждой вершине графа и помечаем ее как пройденную, чтобы больше не заходить в нее. Таким образом, возможно пройтись по каждой вершине графа по одному разу.

Для подсчета количества копилок, которые нужно разбить, мы увеличиваем счетчик каждый раз, когда обращаемся к вершине, которая было уже посещена до этого. После этого необходимо пометить эту вершину другим значением, например – 2, чтобы больше не учитывать ее лишний раз.

Сложность алгоритма:

O(n + k), где n – количество вершин графа, k – количество ребер графа

Код алгоритма:

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

using namespace std;

int depth_first_search(pair<int, list<int>>* graph, int i) {

    int counter = 0;

    // вершина становится посещенной в первый раз, поэтому присваиваем 1
```

```
    graph[i].first = 1;

    for (int element : graph[i].second) {
        // если связанная вершина уже посещалась 1 раз, то добавляем
counter
        if (graph[element].first == 1) {
            return 1;
        }
        // если еще ни разу не вызывалась, то рекурсивно вызываем метод
depth_first_search()
        if (graph[element].first == 0) {
            counter += depth_first_search(graph, element);
        }
    }

    // присваиваем значению посещения 2, чтобы больше не считать эту
вершину так как она посещена и уже посчитана
    graph[i].first = 2;

    return counter;
}

int main() {

    // счетчик для результата
    int counter = 0;

    // значение количества ключей в копилке
    int key_of_graph;

    // количество вершин графа (копилок)
    int n;

    // считывание
    cin >> n;

    // считывание в каждую вершину графа ключи для каких графов она имеет
    pair<int, list<int>> graph[n];
    for (int i = 0; i < n; i++) {
        cin >> key_of_graph;
        graph[key_of_graph - 1].second.push_back(i);
    }

    for (int i = 0; i < n; i++) {

        // если связанную вершину графа еще не посещали, то вызываем
поиск в глубину
        if (graph[i].first == 0) {
            counter += depth_first_search(graph, i);
        }
    }

    cout << counter << endl;
```

```
}
```

**Задача №O «Долой списывание!»**

Пояснение к примененному алгоритму:

Для проверки возможности разделения учеников на две группы необходимо воспользоваться тем же способом поиска в глубину, чтобы посетить каждого «ученика» (вершину графа) по одному разу. При этом каждому ученику необходимо присвоить цифру (например, 1), и, если они передавали друг другу записки – второму ученику необходимо присвоить другое значение (например, 2).

Если же выясниться, что значение ученику уже было присвоено и оно такое же, как у ученика, который передавал записку – значит разделить учеников невозможно. Если же всем ученикам присвоятся подходящие значения, то разделить учеников можно.

Сложность алгоритма:

O(n + m), где n – количество вершин (учеников), m – количество ребер (пар)

Код алгоритма:

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

using namespace std;

bool depth_first_search(pair<int, list<int>>* graph, int i, int
visited_group) {

    graph[i].first = visited_group;

    for (int element : graph[i].second) {
        if (graph[element].first == 0) {
            if (!depth_first_search(graph, element, (visited_group ==
2) ? 1 : 2)) {
                return false;
            }
        }
        if (graph[element].first == visited_group) {
            return false;
        }
    }
```

```
    return true;
}

int main() {

    // количество лкшат
    int n;
    // количество пар лкшат
    int m;

    bool res = true;

    cin >> n;
    cin >> m;

    int number_of_lk_1;
    int number_of_lk_2;
    pair<int, list<int>> graph[n];
    for (int i = 0; i < m; i++) {
        cin >> number_of_lk_1;
        cin >> number_of_lk_2;
        graph[number_of_lk_1 - 1].second.push_back(number_of_lk_2 - 1);
        graph[number_of_lk_2 - 1].second.push_back(number_of_lk_1 - 1);
    }

    for (int i = 0; i < n; i++) {
        if (!res) {
            break;
        }
        if (graph[i].first == 0) {
            res = depth_first_search(graph, i, 1);
        }
    }

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

}
```