Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4
по дисциплине “Алгоритмы и структуры данных”
Базовые задачи

Выполнила:
Шевченко Д. П.,
группа P3230

Преподаватели:
Косяков М.С.
Тараканов Д. С.

Санкт-Петербург
2024

Задача M. Цивилизация

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <string>
#include <algorithm>

struct Cell {
    int x, y, dist;

    Cell(int x, int y, int dist = 0) : x(x), y(y), dist(dist) {}
};

struct CompareDist {
    bool operator()(const Cell &a, const Cell &b) {
        return a.dist > b.dist;
    }
};

bool isValid(int x, int y, int N, int M) {
    return (x >= 0 && x < N && y >= 0 && y < M);
}

void
processCell(std::priority_queue<Cell, std::vector<Cell>, CompareDist> &queue,
std::vector<std::vector<int>> &distance,
            std::vector<std::string> &lines, int n, int m) {
    auto cell = queue.top();
    queue.pop();

    std::vector<int> dx = {-1, 0, 1, 0};
    std::vector<int> dy = {0, 1, 0, -1};

    for (int i = 0; i < 4; ++i) {
        int new_x = cell.x + dx[i];
        int new_y = cell.y + dy[i];
        if (isValid(new_x, new_y, n, m) && lines[new_x][new_y] != '#') {
            int new_distance = cell.dist + (lines[new_x][new_y] == '.' ? 1 :
2);
            if (new_distance < distance[new_x][new_y]) {
                distance[new_x][new_y] = new_distance;
                queue.emplace(new_x, new_y, new_distance);
            }
        }
    }
}

std::string findPath(std::vector<std::vector<int>> &distance,
std::vector<std::string> &lines, int x_start, int y_start,
                     int x_finish, int y_finish, int n, int m) {
    std::string result;
    int x = x_finish, y = y_finish;
    std::vector<int> dx = {-1, 0, 1, 0};
    std::vector<int> dy = {0, 1, 0, -1};
    std::string direction = "NESW";

    while (x != x_start || y != y_start) {
        for (int i = 0; i < 4; i++) {
            int previous_x = x - dx[i];
            int previous_y = y - dy[i];
            if (isValid(previous_x, previous_y, n, m) &&
```

```
                distance[previous_x][previous_y] + (lines[x][y] == '.' ? 1 :
2) == distance[x][y]) {
                result += direction[i];
                x = previous_x;
                y = previous_y;
                break;
            }
        }
    }
    std::reverse(result.begin(), result.end());
    return result;
}

int main() {
    int n, m, x_start, y_start, x_finish, y_finish;
    std::cin >> n >> m >> x_start >> y_start >> x_finish >> y_finish;
    --x_start;
    --y_start;
    --x_finish;
    --y_finish;

    std::vector<std::string> lines(n);
    for (int i = 0; i < n; i++) {
        std::cin >> lines[i];
    }

    std::vector<std::vector<int>> distance(n, std::vector<int>(m, INT_MAX));
    distance[x_start][y_start] = 0;

    std::priority_queue<Cell, std::vector<Cell>, CompareDist> queue;
    queue.emplace(x_start, y_start);

    while (!queue.empty()) {
        processCell(queue, distance, lines, n, m);
    }

    if (distance[x_finish][y_finish] == INT_MAX) {
        std::cout << -1 << std::endl;
    } else {
        std::cout << distance[x_finish][y_finish] << std::endl;
        std::string result = findPath(distance, lines, x_start, y_start,
x_finish, y_finish, n, m);
        std::cout << result << std::endl;
    }

    return 0;
}
```

Пояснение:
Используем алгоритм Дейкстры для нахождения пути с минимальной стоимостью от начальной точки до конечной.
Создается двумерный вектор, который хранит в себе минимальное расстояние от начальной точки до каждой точки на карте. Потом из очереди берется клетка с наименьшим расстоянием. Для каждой соседней клетки проверяется, можно ли туда пройти, нет ли там воды. Если можно, то вычисляется новое расстояние до этой клетки. Если новое расстояние меньше текущего, то обновляется расстояние в векторе и клетка добавляется в очередь. Когда найдем минимальное расстояние, то восстанавливаем путь, идя от конечной точки к начальной и проверяя, откуда пришли в каждую клетку.
Сложность: O((n+m) log m), где n - количество клеток на карте, а m - количество возможных переходов между клетками.

Задача N. Свинки-копилки

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

void dfs_method(int i, std::vector<int> &vis, std::vector<std::vector<int>>
&lines) {
    for (int j = 0; j < lines[i].size(); j++) {
        int s = lines[i][j];
        if (!vis[s]) {
            vis[i] = true;
            dfs_method(s, vis, lines);
        }
    }
}

int main() {
    int line;
    int n;
    std::cin >> n;
    std::vector<int> vis(n + 1, false);
    std::vector<std::vector<int>> lines(n + 1);
    for (int i = 1; i <= n; i++) {
        std::cin >> line;
        lines[i].push_back(line);
        lines[line].push_back(i);
    }
    int result = 0;
    for (int i = 0; i <= n; i++) {
        if (!vis[i]) {
            result++;
            dfs_method(i, vis, lines);
        }
    }
    std::cout << result - 1;


    return 0;
}
```

Пояснение:
В данной задаче я использовала поиск в глубину. Записывала в вектор номер копилки и номер, в которой лежит ключ от нее. После чего нужно проверить для каждой копилки, есть ли она в посещенных вершинах, если нет, то делаем посещенной ее и все копилки, которые с ней связаны.
Сложность поиска в глубину: m – количество всех вершин, n – количество всех ребер O(n+m), так как обходим всех соседей всех вершин, кроме тех, которые уже посещали.

Задача O. Долой списывание!

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>

bool flag = true;

void dfs_method(int i, bool visit, std::unordered_map<int, bool> &groups,
std::vector<std::vector<int>> &lines, std::vector<bool> &vis) {
    vis[i] = true;
    groups[i] = visit;
    for (int j = 0; j < lines[i].size(); j++) {
        int s = lines[i][j];
        if (!vis[s]) {
            dfs_method(s, !visit, groups, lines, vis);
        } else if (groups[s] == visit) {
            flag = false;
            std::cout << "NO" << std::endl;
            exit(0);
        }
    }
}

int main() {
    int n, m;
    std::cin >> n >> m;
    int left, right;
    std::vector<bool> vis(n+1);
    std::unordered_map<int, bool> groups;
    std::vector<std::vector<int>> lines(n+1);
    for (int i = 0; i < m; i++) {
        std::cin >> left >> right;
        lines[left].push_back(right);
        lines[right].push_back(left);
    }
    for (int i = 0; i < n; i++) {
        if (!vis[i]) {
            dfs_method(i, true, groups, lines, vis);
        }
    }
    if (flag) {
        std::cout << "YES" << std::endl;
    }
    return 0;
}
```

Пояснение:
В данной задаче граф двудольный, использую обход в глубину для проверки графа на двудольность. Помечаю вершины разным «цветом», чтобы разделить их на две группы, перемещаясь по ребрам. Если две смежные оказываются одного цвета, то граф не двудольный. Если все вершины посещены и все смежные разных цветов, то граф двудольный.
Сложность: m – количество всех вершин, n – количество всех ребер
O(n+m), так как обходим всех соседей всех вершин, кроме тех, которые уже посещали.

Задача P. Авиаперелеты

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

int main() {
    int n;
    std::cin >> n;
    int result = 0;
    std::vector<std::vector<int>> graph(n, std::vector<int>(n));
    int val;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            std::cin >> val;
            graph[i][j] = val;
        }
    }

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                int new_val = std::max(graph[i][k], graph[k][j]);
                graph[i][j] = std::min(graph[i][j], new_val);
            }
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result = std::max(result, graph[i][j]);
        }
    }

    std::cout << result;

    return 0;
}
```

Пояснение: будем использовать алгоритм Флойда-Уоршалла, который находит кратчайшие пути между всеми парами вершин в графе.
Для каждой пары городов находим минимальный расход топлива. Новое значение элементов графа равно минимуму из старого значения элемента в графе и максимума из graph[i][k] и graph[k][j]. Для того чтобы, если самолет может сэкономить топливо, сделав остановку в городе, то он будет это делать.
После того как все кратчайшие пути найдены, находим максимальное значение в графе. Это будет оптимальным размером бака, так как он должен быть подходящим для самого длинного пути.
Временная сложность алгоритма составляет O(n^3), где n - количество городов.
Пространственная сложность - O(n^2).