Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №1

по «Алгоритмам и структурам данных»

Блок 4

Выполнил:
Студент группы P3216
Билошийцкий Михаил Владимирович

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

Санкт-Петербург
2024

M. Цивилизация

Решение:

Эта программа решает задачу о нахождении оптимального маршрута для поселенца в игре "Цивилизация". Изначально считываются размеры карты, начальная и конечная позиции, а также сама карта. Затем используется поиск в ширину (BFS) для нахождения кратчайшего пути от начальной до конечной точки.

Во время обхода обновляются расстояния до каждой клетки и сохраняется направление движения до неё. После обхода построен оптимальный путь от конечной до начальной точки, используя сохранённые направления. Программа эффективно использует пространство и время за счёт BFS и минимизации памяти при хранении данных о клетках карты.

Временная сложность алгоритма: O(NxM)
Сложность алгоритма по памяти: O(NxM)

Дополнительные пояснения в коде:

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

// Максимальное расстояниеы
#define INT32_MAX 2147483647

// Клетки
#define FIELD '.'
#define FOREST 'W'
#define WATER '#'

// Длинна пути
#define FIELD_DUR 1
#define FOREST_DUR 2

// Направления
#define UP 'N'
#define DOWN 'S'
#define LEFT 'W'
#define RIGHT 'E'

// Есть ли x, y на полеы
```

```cpp
bool in_field(int32_t x, int32_t y, int32_t n, int32_t m) {
    return x >= 0 && x < n && y >= 0 && y < m;
}

// Возвращает длинну пути у клетки
int get_dur(char cell) {
    return cell == FIELD ? FIELD_DUR : FOREST_DUR;
}

// Переводит направление в обратное
char reverse_direction(char dir) {
    if (dir == UP) return DOWN;
    if (dir == DOWN) return UP;
    if (dir == LEFT) return RIGHT;
    if (dir == RIGHT) return LEFT;
    return '\0';
}

int main(int argc, char** argv) {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int32_t n, m, start_x, start_y, dest_x, dest_y;
    cin >> n >> m >> start_x >> start_y >> dest_x >> dest_y;

    // Приводим к 0 индексации
    start_x--;
    start_y--;
    dest_x--;
    dest_y--;

    // Игровое поле
    vector<vector<char>> field(n, vector<char>(m));
    // Матрица расстояний
    vector<vector<int32_t>> dist(n, vector<int32_t>(m, INT32_MAX));
    // Матрица направлений
    vector<vector<char>> prev_dir(n, vector<char>(m, '\0'));

    // Ввод поля
```

```
for (int32_t i = 0; i < n; i++) {
    for (int32_t j = 0; j < m; j++) {
        cin >> field[i][j];
    }
}

// Очередь клеток для обработки
queue<pair<int32_t, int32_t>> q;
q.push({start_x, start_y});
dist[start_x][start_y] = 0;

// Измненения направлений
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};
char dir[] = {UP, DOWN, LEFT, RIGHT};

while (!q.empty()) {
    auto [x, y] = q.front();
    q.pop();

    for (int i = 0; i < 4; ++i) {
        // Координатый новой клетки
        int nx = x + dx[i];
        int ny = y + dy[i];

        // Если клетка в пределах поля и это не вода
        if (in_field(nx, ny, n, m) && field[nx][ny] != WATER) {
            // Рассчёт расстояния
            int new_dist = dist[x][y] + get_dur(field[nx][ny]);
            // Если меньше, то запоминаем
            if (new_dist < dist[nx][ny]) {
                dist[nx][ny] = new_dist;
                prev_dir[nx][ny] = dir[i];
                q.push({nx, ny});
            }
        }
    }
}
```

```
    // Время, выводится в результат
    int32_t time = dist[dest_x][dest_y];
    if (time != INT32_MAX) {
        // Реверс пути
        string path = "";
        int x = dest_x;
        int y = dest_y;
        while (x != start_x || y != start_y) {
            char direction = prev_dir[x][y];
            path = direction + path;
            int dx, dy;
            if (direction == UP) { dx = 1; dy = 0; }
            else if (direction == DOWN) { dx = -1; dy = 0; }
            else if (direction == LEFT) { dx = 0; dy = 1; }
            else if (direction == RIGHT) { dx = 0; dy = -1; }
            x += dx;
            y += dy;
        }
        cout << time << endl;
        cout << path << endl;
    } else {
        cout << -1 << endl;
    }
    return 0;
}
```

N. Свинки-копилки

Эта программа решает задачу о минимальном количестве копилок, которые нужно разбить, чтобы получить доступ ко всем ключам. Она использует подход на основе графов. Вначале считывается количество копилок и их связи с ключами. Затем строится граф, где вершины представляют копилки, а рёбра - связи между копилками и ключами. Для определения минимального количества копилок, которые нужно разбить, используется обход графа в глубину.

После обхода графа выявляются независимые компоненты связности. Каждая независимая компонента требует разбития хотя бы одной копилки. Количество таких компонент определяет минимальное количество копилок, которые нужно разбить.

Временная сложность алгоритма: O(N)
Сложность алгоритма по памяти: O(N)

Дополнительные пояснения в коде:

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

#define WHITE 0
#define BLACK 1

// Покрас графа, если прошли, то красим в чёрный
void colorize(vector<unordered_set<int32_t>>& graph, vector<int32_t>& colors,
int32_t start) {
    colors[start] = BLACK;
    for (int32_t neighbor : graph[start]) {
        if (colors[neighbor] == WHITE) {
            colorize(graph, colors, neighbor);
        }
    }
}

int main(int argc, char** argv) {
    int32_t n;
    cin >> n;
    // Граф с копилками
    vector<unordered_set<int32_t>> graph(n);
    // Цвета вершин графа с копилками
    vector<int32_t> colors(n, WHITE);
    // Ввод данных
    for (int32_t i = 0; i < n; i++) {
        int32_t key;
        cin >> key;
        key--;
        // Фишка решения: ребро двунаправленное
        graph[i].insert(key);
        graph[key].insert(i);
    }
    int result = 0;
```

```
    for (int32_t i = 0; i < n; i++) {
        if (colors[i] == WHITE) {
            // Как только появляется непокрашенная вершина, то result++
            // так как это говорит о новом независимом компоненте связности
            // как минимум одну копилку, чтобы доставть ключ и открыть остальыне
            result++;
            colorize(graph, colors, i);
        }
    }
    cout << result << endl;
    return 0;
}
```

O. Долой списывание!

Программа решает задачу о проверке возможности разделения лкшат на две группы так, чтобы обмен записками осуществлялся только между разными группами. Она использует алгоритм проверки двудольности графа. Вначале считывается количество лкшат и пары лкшат, обменивающихся записками. Затем строится граф, где вершины - лкшаты, а рёбра - обмены записками. Для каждой вершины выполняется проверка на двудольность графа с этой вершиной в качестве начальной. Если хотя бы одна проверка не пройдена, то граф не может быть разделён на две группы с учётом условий задачи.

Временная сложность алгоритма: O(N+M)
Сложность алгоритма по памяти: O(N+M)

Дополнительные пояснения в коде:

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

#define WHITE -1
#define BLACK 0

// Функция для определения, является ли граф двудольным, начиная с заданной вершины
bool is_bipartite(vector<vector<int32_t>>& graph, int32_t start) {
    int32_t n = graph.size();
    vector<int32_t> colors(n, WHITE);
```

```
    queue<int32_t> q;

    q.push(start);
    // Помечаем стартовую вершину чёрной
    colors[start] = BLACK;

    while (!q.empty()) {
        int32_t curr = q.front();
        q.pop();

        for (int32_t neighbor : graph[curr]) {
            if (colors[neighbor] == WHITE) { // Если сосед не посещён
                // Красим его в противоположный цвет текущей вершины
                colors[neighbor] = 1 - colors[curr];
                q.push(neighbor);
            } else if (colors[neighbor] == colors[curr]) { // Если сосед имеет
тот же цвет, что и текущая вершина
                // Граф не является двудольным
                return false;
            }
        }
    }

    // Если цветовка прошла успешно, граф двудолен
    return true;
}

int main(int argc, char** argv) {
    int32_t n, m;
    cin >> n >> m;

    vector<vector<int32_t>> graph(n);

    // Считываем пары вершин, между которыми есть связи
    for (int32_t i = 0; i < m; i++) {
        int32_t u, v;
        cin >> u >> v;
        --u;
        --v;
```

```
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // Флаг, определяющий, можно ли разделить граф на две группы
    bool can_be_devided = true;
    for (int32_t i = 0; i < n; i++) {
        // Если вершина не изолированная и граф не двудолен
        if (graph[i].size() > 0 && !is_bipartite(graph, i)) {
            can_be_devided = false;
            break;
        }
    }

    // Выводим результат
    cout << (can_be_devided ? "YES" : "NO") << endl;

    return 0;
}
```