Задача.

Для того, чтобы ноги персонажа не пересекались при движении, необходимо, чтобы они одновременно не сделали передвижение на противоположные клетки — например, одновременно с нижней на верхнюю и с правой на левую.

Для того, чтобы исключить эти ситуации, добавим дополнительную проверку в часть кода, которая отвечает за обработку только корректных конфигураций поля — заведем ассоциативный массив, который будет указывать нам на то, какая клетка является противоположной для данной и будем проверять, сходил ли пользователь так или нет.

```
double play_DDR(std::string gameFile) {
    std::ifstream gf(gameFile);
    const int SIZE = 5; // всего 5 плиток
    int x[SIZE] = {-1, 0, 0, 0, 1}; // координаты плиток
    int y[SIZE] = {0, -1, 0, 1, 0}; // кодирование: 0 - Left, 1 - Down, 2 -
Center, 3 - Up, 4 - Right
    auto dist = [&x, &y](int i, int j) {
        return sqrt( (x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])*(y[i]-y[j]) ); //
евклидово расстояние между
    };
    int N;
    gf >> N;
    std::vector< std::vector< std::vector<double> > > E(N+1, std::vector<
std::vector<double> >(5, std::vector<double>(5, INFINITY)));
    // E[i][A][B] - стоимость(расстояние) конфигцрации, при котогой правая нога
стоит на A, левая на B на i-той игровой позиции
    // изначально положим бесконечность в каждую конфигурацию
    std::vector< std::vector<bool> > game(N+1, std::vector<bool>(5, false));
    // данные игры. game[i][C] == True -> на i-той позиции нога должна
находиться на C, False -> не обязана
    for(int i = 0; i < N; i++) {
        int q;
        gf >> q;
        for(int j = 0; j < q; j++) { // считывает параметры игры
            char position;
            gf >> position;
            switch (position) {
                case 'U':
                    game[i][3] = true;
                    break;
                case 'D':
                    game[i][1] = true;
                    break;
                case 'L':
                    game[i][0] = true;
                    break;
                case 'R':
                    game[i][4] = true;
                    break;
            }
        }
    }
    std::map<int, int> against;
    against[3] = 1;
    against[1] = 3; // противоположные клетки
    against[0] = 4;
    against[4] = 0;
    against[5] = 5;
    E[0][2][2] = 0.0; // Изначально стоим в центре. Расстояние равно 0
```

```
    for(int i = 0; i < N; i++) { // начинаем расчитывать расстояния конфигураций
        for(int C = 0; C < SIZE; C++) {
            for(int D = 0; D < SIZE; D++) { // Перебираем все конфигурации на текущей позиции
                bool incorrect = false;
                for(int game_conf = 0; game_conf < SIZE; game_conf++) {
                    if(game[i][game_conf] and !( game_conf == C or game_conf == D ) ){ // проверяем, является ли данная конфигурация допустимой с точки зрения игры
                        incorrect = true;
                    }
                    if(i >= 1 and game[i][game_conf] and game[i-1][against[game_conf]]) {
                        incorrect = true; // проверка на противоположную клетку
                    }
                }
                if(incorrect) continue;  // пропускаем все неподходящие конфигурации
                for(int A = 0; A < SIZE; A++){
                    for(int B = 0; B < SIZE; B++) { // начитаем вычисление каждой конфигурации на следующей позиции
                        double cost = std::min(dist(A, C) + dist(B, D), // у нас есть только два варианта смены конфигцрации
                                               dist(A, D) + dist(B, C)); // или A->C и B->D или наоборот
                        E[i+1][C][D] = std::min(E[i+1][C][D], E[i][A][B]+cost); // выбираем минимальный путь
                    }
                }
            }
        }
    }
    double result = INFINITY;
    for(int A = 0; A < SIZE; A++) {
        for(int B = 0; B < SIZE; B++) {
            result = std::min(result, E[N][A][B]); // Выбираем минимальный на последней позиции
        }
    }
    return result;
}
```