```
#include <iostream>
#include <vector>
#include <climits>
#include <cassert>
#include <tuple>

int simulate_game(std::vector<std::tuple<int, int>> min, std::vector<std::tuple<int,
 int>> max, std::tuple<int, int> starting_pos, bool stupid) {
    int num_moves = 0;
    while(std::get<1>(starting_pos) != 0) {
        if(num_moves++ % 2 == 0) {
            starting_pos = min[std::get<1>(starting_pos)];
        } else {
            starting_pos = max[std::get<1>(starting_pos)];
        }
    }

    return num_moves;
}

void testcase(){

    // Number of positions
    int n; std::cin >> n;
    // Number of transistions
    int m; std::cin >> m;

    // Starting point of red meeple
    int r; std::cin >> r;
    // Starting point of black meeple
    int b; std::cin >> b;


    // Adjacency list to represent transitions
    std::vector<std::vector<int>> adjacency_list(n, std::vector<int>(0 , -1));
    for(int i = 0; i < m; ++i) {
        int u; std::cin >> u;
        int v; std::cin >> v;
        u = n - u;
        v = n - v;
        adjacency_list[u].push_back(v);
    }

    // Store quickest transition (dist, next position)
    std::vector<std::tuple<int, int>> min_dist_to_target(n, std::make_tuple(-1, -1))
;
    // Store longest transition (dist, next position)
    std::vector<std::tuple<int, int>> max_dist_to_target(n, std::make_tuple(-1, -1))
;
    // Distance to target from target is 0
    min_dist_to_target[0] = std::make_tuple(0, 0);
    max_dist_to_target[0] = std::make_tuple(0, 0);
    for(int i = 1; i < n; ++i) {
        std::tuple<int, int> min_dist = std::make_tuple(INT_MAX, -1);
        std::tuple<int, int> max_dist = std::make_tuple(-1, -1);
        // For every transition from u to v
        for(int transition = 0; transition < adjacency_list[i].size(); ++transition)
 {
            int j = adjacency_list[i][transition];
            // Compute best and worst choice of transition
            int min_dist_via_j = 1 + std::get<0>(max_dist_to_target[j]);
            if (min_dist_via_j < std::get<0>(min_dist)) {
                min_dist = std::make_tuple(min_dist_via_j, j);
            }
            int max_dist_via_j = 1 + std::get<0>(min_dist_to_target[j]);
            if (max_dist_via_j > std::get<0>(max_dist)) {
                max_dist = std::make_tuple(max_dist_via_j, j);
            }
        }
        min_dist_to_target[i] = min_dist;
        max_dist_to_target[i] = max_dist;
    }
    // Simulate a game - move red and black meeple through their respecitve sequence
s
    // Count the number of moves made by each
    int dist_from_start_r = std::get<1>(min_dist_to_target[n - r]) + 1;
    std::tuple<int, int> r_pos = std::make_tuple(dist_from_start_r, n - r);
    int r_moves = simulate_game(min_dist_to_target, max_dist_to_target, r_pos, false
);

    int dist_from_start_b = std::get<1>(min_dist_to_target[n - b]) + 1;
    std::tuple<int, int> b_pos = std::make_tuple(dist_from_start_b, n - b);
    int b_moves = simulate_game(min_dist_to_target, max_dist_to_target, b_pos, true)
;

    // The winner is the one with less moves
    // If the number of moves is equal
    if (r_moves < b_moves) {
        // Sherlock wins
        std::cout << "0" << std::endl;
    } else if (r_moves > b_moves){
        // Moriarty wins
        std::cout << "1" << std::endl;
    } else {
        // Special case: number of moves is equal
        // The meeples move 2 times in a row, but staggered over each move, like thi
s
        // 1. r b
        // 2. b r
        // 3. r b
        // 4. b r
        // So if the number of moves is even, then the black meeple will end on the
target first
        // and if the number of moves is odd, then the red will win.
        std::cout << 1 - r_moves % 2 << std::endl;
    }

}

int main(){
    int t; std::cin >> t;

    for(int i = 0; i < t; ++i) {
        testcase();
    }
}
```