1. We employed the A* search algorithm. The specific implementation process is as follows: We utilized a priority queue, using the heap queue implemented by heapq to store the nodes to be expanded. The nodes are sorted according to f_score = g_score + heuristic. g_score is a dictionary used to record the actual movement cost from the starting point to each node. came_from is also a dictionary used to record the parent node and the movement action of the node. came_from is finally used to trace back the path. We also wrote two functions to handle the movement of the red frog, namely handle_step_move, which detects the adjacent LILY_PAD grids, updates the cost and the path. handle_jump_move checks whether the jumping condition is met, where the intermediate grid is the blue frog and the target grid is the LILY_PAD. The process terminates when any target grid is visited, and then the final route is generated by backtracking.

Time complexity: O(b^d)

Space complexity: O(b^d)

2.The heuristic in the code calculates the minimum value of the current cell to the maximum value of the row and column differences among all target cells. This heuristic ensures that the actual cost will not be exceeded because each move can reduce at most one row or one column. Therefore, A* can guarantee to find the shortest path.

3.If six frogs are moved simultaneously, the problem will transform from a single-path planning to a multi-agent path planning. The complexity will increase significantly. Secondly, conflict issues need to be handled. The movement routes of different frogs may conflict with each other. For our current move function, conflict detection is necessary. We need to check whether the target square is occupied by other red frogs. Secondly, we can plan paths separately for each red frog, then compare whether the paths conflict with each other, and modify the routes to resolve the conflicts.