#LeetCode 505

https://leetcode.com/problems/the-maze-ii/description/

Yifeng Zeng

#Description

505. The Maze II

#Idea Report

Basically this is a search problem in a graph. The ball can move like a rook in chess. We can search for any position this ball will be, and calculate the distance from start. Then if the ball finally reach to the destination, we compare the distances and return the minimum.

Code

```
// Time Limit Exceeded
class Solution {
    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        if (maze == null || maze.length == 0 || maze[0].length == 0) {
            return -1;
        }

        int rows = maze.length;
        int cols = maze[0].length;
        int[][] visited = new int[rows][cols];
        int[] shortest = new int[1];
        shortest[0] = Integer.MAX_VALUE;

        shortestDistance(maze, start, destination, visited, 0, shortest);

        return shortest[0] == Integer.MAX_VALUE ? -1 : shortest[0];
    }

    private void shortestDistance(int[][] maze,
                                  int[] start,
                                  int[] destination,
                                  int[][] visited,
                                  int distance,
                                int[] shortest) {
        if (start[0] == destination[0] && start[1] == destination[1]) {
            shortest[0] = Math.min(shortest[0], distance);
            return;
        }
        List<int[]> nextLocations = findNextLocations(maze, start, visited);
        for (int[] nextLocation : nextLocations) {
            if (visited[nextLocation[0]][nextLocation[1]] == 0) {
                visited[nextLocation[0]][nextLocation[1]] = 1;
                shortestDistance(maze, nextLocation, destination, visited,
                    distance + calculateDistance(start, nextLocation),
                    shortest);
                visited[nextLocation[0]][nextLocation[1]] = 0;
            }

        }
    }


    private List<int[]> findNextLocations(int[][] maze, int[] start,
                                          int[][] visited) {
        List<int[]> nextLocations = new ArrayList<>();
        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};

        for (int i = 0; i < dx.length; i++) {
            int[] nextLocation =
                findNextLocation(maze, start[0], start[1], dx[i], dy[i]);
            nextLocations.add(nextLocation);
        }
        return nextLocations;
    }


    private int[] findNextLocation(int[][] maze, int x, int y, int dx, int dy) {
        int[] nextLocation = new int[]{x, y};

        while (isValid(maze, x, y)) {
            nextLocation[0] = x;
            nextLocation[1] = y;
            x += dx;
            y += dy;
        }
        return nextLocation;
    }


    private boolean isValid(int[][] maze, int x, int y) {
        if (0 <= x && x < maze.length && 0 <= y && y < maze[0].length) {
            return maze[x][y] == 0;
        }
        return false;
    }


    private int calculateDistance(int[] start, int[] end) {
        if (start[0] == end[0]) {
            return Math.abs(start[1] - end[1]);
        }
        return Math.abs(start[0] - end[0]);
    }
}
```

The above DFS code exceeds the time limit, because we search all the posibilities. But some of the posibilities might not need to be searched because the ball might travel to a wrong direction. We use a distances[][] matrix to save the minimum distance from start point for each possible location. If we search to a certain location but the current distance is larger than the minimum distance in the record (in distances[][]), we just stop the meaningless searching because that path is no longer the optimal path.

Code

```
// AC
class Solution {
    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        if (maze == null || maze.length == 0 || maze[0].length == 0) {
            return -1;
        }

        int rows = maze.length;
        int cols = maze[0].length;
        int[][] distances = new int[rows][cols];
        for (int[] dist : distances) {
            Arrays.fill(dist, Integer.MAX_VALUE);
        }
        distances[start[0]][start[1]] = 0;

        Deque<int[]> q = new LinkedList<>();
        q.offer(start);

        while (!q.isEmpty()) {
            int[] cur = q.poll();
            List<int[]> nextLocations = findNextLocations(maze, cur);
            for (int[] nextLocation : nextLocations) {
                int distance = calculateDistance(cur, nextLocation)
                    + distances[cur[0]][cur[1]];
                if (distances[nextLocation[0]][nextLocation[1]] > distance) {
                    distances[nextLocation[0]][nextLocation[1]] = distance;
                    q.offer(nextLocation);
                }
            }
        }

        return distances[destination[0]][destination[1]] == Integer.MAX_VALUE ?
            -1 : distances[destination[0]][destination[1]];
    }

    // reuse the following methods from DFS
    private List<int[]> findNextLocations(int[][] maze, int[] start) {
        List<int[]> nextLocations = new ArrayList<>();
        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};

        for (int i = 0; i < dx.length; i++) {
            int[] nextLocation =
                findNextLocation(maze, start[0], start[1], dx[i], dy[i]);
            nextLocations.add(nextLocation);
        }
        return nextLocations;
    }


    private int[] findNextLocation(int[][] maze, int x, int y, int dx, int dy) {
        int[] nextLocation = new int[]{x, y};

        while (isValid(maze, x, y)) {
            nextLocation[0] = x;
            nextLocation[1] = y;
            x += dx;
            y += dy;
        }
        return nextLocation;
    }


    private boolean isValid(int[][] maze, int x, int y) {
        if (0 <= x && x < maze.length && 0 <= y && y < maze[0].length) {
            return maze[x][y] == 0;
        }
        return false;
    }


    private int calculateDistance(int[] start, int[] end) {
        if (start[0] == end[0]) {
            return Math.abs(start[1] - end[1]);
        }
        return Math.abs(start[0] - end[0]);
    }
```

```
}
```

#Summary

- Record the temporary optimal when searching a global optimal is a way to speed up (memorize search).
- Use different function to modularize the program is a good practice.
- Note:
    - Use this for four direction movement:
    - int[] dx = {0, 0, 1, -1};
    - int[] dy = {1, -1, 0, 0};

#Follow up

In the DFS method, I used a distances[][] to memorize the search and change the int[] to class Point and finally got accepted.

Code

```
class Solution {
    // DFS AC
    class Point{
        int x;
        int y;
        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }

    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        if (maze == null || maze.length == 0 || maze[0].length == 0) {
            return -1;
        }

        int rows = maze.length;
        int cols = maze[0].length;
        int[][] distances = new int[rows][cols];
        for (int[] dist : distances) {
            Arrays.fill(dist, Integer.MAX_VALUE);
        }
        distances[start[0]][start[1]] = 0;
        shortestDistance(maze, new Point(start[0], start[1]), distances);

        return distances[destination[0]][destination[1]] == Integer.MAX_VALUE ?
            -1 : distances[destination[0]][destination[1]];
    }

    private void shortestDistance(int[][] maze,
                                  Point cur,
                                  int[][] distances) {
        List<Point> nextLocations = findNextLocations(maze, cur);
        for (Point nextLocation : nextLocations) {
            int nextDistance = distances[cur.x][cur.y]
                               + calculateDistance(cur, nextLocation);
            if (distances[nextLocation.x][nextLocation.y] > nextDistance) {
                distances[nextLocation.x][nextLocation.y] = nextDistance;
                shortestDistance(maze, nextLocation, distances);
            }
        }
    }


    private List<Point> findNextLocations(int[][] maze, Point cur) {
        List<Point> nextLocations = new ArrayList<>();
        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};

        for (int i = 0; i < dx.length; i++) {
            Point nextLocation =
                findNextLocation(maze, cur.x, cur.y, dx[i], dy[i]);
            nextLocations.add(nextLocation);
        }
        return nextLocations;
    }


    private Point findNextLocation(int[][] maze, int x, int y, int dx, int dy) {
        Point nextLocation = new Point(x, y);

        while (isValid(maze, x, y)) {
            nextLocation.x = x;
            nextLocation.y = y;
            x += dx;
            y += dy;
        }
        return nextLocation;
    }


    private boolean isValid(int[][] maze, int x, int y) {
        if (0 <= x && x < maze.length && 0 <= y && y < maze[0].length) {
            return maze[x][y] == 0;
        }
        return false;
    }


    private int calculateDistance(Point start, Point end) {
        if (start.x == end.x) {
            return Math.abs(start.y - end.y);
        }
        return Math.abs(start.x - end.x);
    }
}
```

#Follow up

The same DFS code but using int[] instead of Point class, I thought I got time limit exceeded, but got accepted when I submit again.

Code

```
class Solution {

    // DFS AC
    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        if (maze == null || maze.length == 0 || maze[0].length == 0) {
            return -1;
        }

        int rows = maze.length;
        int cols = maze[0].length;
        int[][] distances = new int[rows][cols];
        for (int[] dist : distances) {
            Arrays.fill(dist, Integer.MAX_VALUE);
        }
        distances[start[0]][start[1]] = 0;

        shortestDistance(maze, start, distances);

        return distances[destination[0]][destination[1]] == Integer.MAX_VALUE ?
            -1 : distances[destination[0]][destination[1]];
    }

    private void shortestDistance(int[][] maze,
                                  int[] cur,
                                  int[][] distances) {
        List<int[]> nextLocations = findNextLocations(maze, cur);
        for (int[] nextLocation : nextLocations) {
            int nextDistance = distances[cur[0]][cur[1]]
                + calculateDistance(cur, nextLocation);
            if (distances[nextLocation[0]][nextLocation[1]] > nextDistance) {
                distances[nextLocation[0]][nextLocation[1]] = nextDistance;
                shortestDistance(maze, nextLocation, distances);
            }
        }
    }


    private List<int[]> findNextLocations(int[][] maze, int[] start) {
        List<int[]> nextLocations = new ArrayList<>();
        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};

        for (int i = 0; i < dx.length; i++) {
            int[] nextLocation =
                findNextLocation(maze, start[0], start[1], dx[i], dy[i]);
            nextLocations.add(nextLocation);
        }
        return nextLocations;
    }

    private int[] findNextLocation(int[][] maze, int x, int y, int dx, int dy) {
        int[] nextLocation = new int[]{x, y};

        while (isValid(maze, x, y)) {
            nextLocation[0] = x;
            nextLocation[1] = y;
            x += dx;
            y += dy;
        }
        return nextLocation;
    }

    private boolean isValid(int[][] maze, int x, int y) {
        if (0 <= x && x < maze.length && 0 <= y && y < maze[0].length) {
            return maze[x][y] == 0;
        }
        return false;
    }

    private int calculateDistance(int[] start, int[] end) {
        if (start[0] == end[0]) {
            return Math.abs(start[1] - end[1]);
        }
        return Math.abs(start[0] - end[0]);
    }
}
```