LeetCode 773

https://leetcode.com/problems/sliding-puzzle/description/

Yifeng Zeng

Description

773. Sliding Puzzle

On a 2x3 board, there are 5 tiles represented by the integers 1 through 5, and an empty square represented by 0.

A move consists of choosing 0 and a 4-directionally adjacent number and swapping it.

The state of the board is solved if and only if the board is [[1,2,3],[4,5,0]].

Given a puzzle board, return the least number of moves required so that the state of the board is solved. If it is impossible for the state of the board to be solved, return -1.

Examples:

```
Input: board = [[1,2,3],[4,0,5]]
Output: 1
Explanation: Swap the 0 and the 5 in one move.

Input: board = [[1,2,3],[5,4,0]]
Output: -1
Explanation: No number of moves will make the board solved.

Input: board = [[4,1,2],[5,0,3]]
Output: 5
Explanation: 5 is the smallest number of moves that solves the board.
An example path:
After move 0: [[4,1,2],[5,0,3]]
After move 1: [[4,1,2],[0,5,3]]
After move 2: [[0,1,2],[4,5,3]]
After move 3: [[1,0,2],[4,5,3]]
After move 4: [[1,2,0],[4,5,3]]
After move 5: [[1,2,3],[4,5,0]]
```

```
Input: board = [[3,2,4],[1,5,0]]
Output: 14
```

Note:

- board will be a 2 x 3 array as described above.
- board[i][j] will be a permutation of [0, 1, 2, 3, 4, 5].

Idea Report

The board is a 2 x 3 array, and we are moving '0' up/down/left/right one direction at a time to search if we can get to the final state [[1,2,3],[4,5,0]]. And we are looking for the least number of moves, so we can think of this as a shortest path problem in a graph. But during the search, how can we store which board state are have already searched? The intuitive idea is to hash the state, for example if my current board is [[A,B,C],[D,E,F]], we can use a number A * 100000 + B * 10000 + C * 1000 + D * 100 + E * 10 + F to represent the current state, and use a hash set of Integer to save the states as visited. But during the search, every board state has to be transfered to the hash number, which may not be very effecient. We can directly convert the board to a single String. So our target is "123450", and we can use a hash set of String to store the states as visited. But how do we transfer from one state to another, namely, how do we move the '0'? Since this is a 2 x 3 array converted to a 1-D String, the original index [0,0], [0,1], [0,2], [1,0], [1,1], [1,2] is now mapped to index [0],[1],[2],[3],[4],[5]. So if current index is i, then after we move '0' up/down/left/right, the new index is i - 3, i + 3, i - 1, i + 1 respectively. But we cannot move from [0,2] to [1,0] or from [1,0] to [0,2], so if i == 2, next index == 3, or i == 3 next index == 2, we need to skip it. And because we are looking for the lease number of moves, we can use BFS.

Code:

```
class Solution {
    // BFS AC
    public int slidingPuzzle(int[][] board) {
        String target = "123450";
        StringBuilder start = new StringBuilder();
        for (int[] b : board) {
            for (int num : b) {
                start.append(num + "");
            }
        }
```

```
        Deque<String> q = new LinkedList<>();
        Set<String> visited = new HashSet<>();
        q.offer(start.toString());
        visited.add(start.toString());

        int dist = 0;
        while (!q.isEmpty()) {
            int size = q.size();
            for (int s = 0; s < size; s++) {
                String cur = q.poll();
                if (cur.equals(target)) {
                    return dist;
                }
                offerNext(q, visited, cur);
            }
            dist++;
        }

        return -1;
    }

    private void offerNext(Deque<String> q, Set<String> visited, String cur) {
        char[] chars = cur.toCharArray();
        int index = cur.indexOf('0');
        // move: +-1, +-3
        int[] dx = {1, -1, 3, -3};
        for (int i = 0; i < dx.length; i++) {
            int nextIndex = index + dx[i];
            if (nextIndex < 0 || nextIndex >= cur.length()
                || index == 2 && nextIndex == 3
                || index == 3 && nextIndex == 2) {
                continue;
            }

            swap(chars, index, nextIndex);
            String next = new String(chars);
            if (!visited.contains(next)) {
                visited.add(next);
                q.offer(next);
            }
            swap(chars, index, nextIndex);
        }
    }

    private void swap(char[] chars, int i, int j) {
        char ch = chars[i];
        chars[i] = chars[j];
        chars[j] = ch;
    }
}
```

Summary

- Shortest path search problem in undirected graph using BFS.
- Key point is to find a way to represent each state of the board.
- Follow up is using a OOP style for the implementation.

Follow up

We can construct a State class to that we can all the board states that we have searched. Because we will use a hash set to save all the searched states, so we need to override the hashCode() and equals() method of the State class.

```
class Solution {
    // AC
    class State {
        private int[][] board;
        private int x;
        private int y;
        private int moves;
        private final int[] dx = {0, 0, 1, -1};
        private final int[] dy = {1, -1, 0, 0};

        public State(int[][] board) {
            this.board = new int[board.length][board[0].length];
            for (int r = 0; r < board.length; r++) {
                for (int c = 0; c < board[0].length; c++) {
                    this.board[r][c] = board[r][c];
                }
            }
        }

        @Override
        public int hashCode() {
            int res = 0;
            for (int r = 0; r < board.length; r++) {
                for (int c = 0; c < board[0].length; c++) {
                    res = res * 10 + board[r][c];
                }
            }
            return res;
        }
```

```java
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof State)){
            return false;
        }
        State s = (State) obj;
        for (int r = 0; r < board.length; r++) {
            for (int c = 0; c < board[0].length; c++) {
                if (board[r][c] != s.board[r][c]) {
                    return false;
                }
            }
        }
        return true;
    }

    public boolean isSolved() {
        return hashCode() == 123450;
    }

    public List<State> getNextStates() {
        List<State> nextStates = new ArrayList<>();
        int[] cur = findCurrentZero();
        for (int i = 0; i < dx.length; i++) {
            int r = cur[0] + dx[i];
            int c = cur[1] + dy[i];
            if (isValid(r, c)) {
                swap(cur[0], cur[1], r, c);
                nextStates.add(new State(board));
                swap(cur[0], cur[1], r, c);
            }
        }
        return nextStates;
    }

    private void swap(int r0, int c0, int r, int c) {
        int temp = board[r0][c0];
        board[r0][c0] = board[r][c];
        board[r][c] = temp;
    }

    private boolean isValid(int r, int c) {
        if (0 <= r && r < board.length && 0 <= c && c < board[0].length) {
            return true;
        }
        return false;
    }

    private int[] findCurrentZero() {
        for (int r = 0; r < board.length; r++) {
            for (int c = 0; c < board[0].length; c++) {
```

```
                    if (board[r][c] == 0) {
                        return new int[]{r, c};
                    }
                }
            }
            return null;
        }
    }

    public int slidingPuzzle(int[][] board) {
        State cur = new State(board);
        Deque<State> q = new LinkedList<>();
        Set<State> visited = new HashSet<>();
        q.offer(cur);
        visited.add(cur);

        int dist = 0;
        while (!q.isEmpty()) {
            int size = q.size();
            for (int s = 0; s < size; s++) {
                cur = q.poll();
                if (cur.isSolved()) {
                    return dist;
                }
                for (State next : cur.getNextStates()) {
                    if (!visited.contains(next)) {
                        visited.add(next);
                        q.offer(next);
                    }
                }
            }
            dist++;
        }

        return -1;
    }
}
```